Verification, Validation, and Testing Techniques

Osman Balci
Professor

Department of Computer Science
Virginia Polytechnic Institute and State University (Virginia Tech)
Blacksburg, VA 24061, USA

https://manta.cs.vt.edu/balci

Copyright © Osman Balci

Verification, Validation, and Testing (VV&T) Techniques

- More than 100 techniques exist for M/S VV&T.
- The Figure on the next slide shows a taxonomy of more than 75 VV&T techniques applicable for M/S VV&T.
- The taxonomy classifies the VV&T techniques into four primary categories: informal, static, dynamic, and formal.
- A primary category is further divided into secondary categories as shown in italics.
- The use of mathematical and logic formalism by the techniques in each primary category increases from informal to formal.
- Likewise, the complexity also increases as the primary category becomes more formal.

A Taxonomy of VV&T Techniques

Verification, Validation, and Testing Techniques

Informal

- Audit
- Desk Checking
- Documentation Checking
- Face Validation
- Inspections
- Reviews
- Turing Test
- Walkthroughs

Static

- Cause-Effect Graphing
- *Control Analysis*
  - Calling Structure Analysis
  - Concurrent Process Analysis
  - Control Flow Analysis
  - State Transition Analysis
- *Data Analysis*
  - Data Dependency Analysis
  - Data Flow Analysis
- Fault/Failure Analysis
- *Interface Analysis*
  - Model Interface Analysis
  - User Interface Analysis
- Semantic Analysis
- Structural Analysis
- Symbolic Evaluation
- Syntax Analysis
- Traceability Assessment

Dynamic

- Acceptance Testing
- Alpha Testing
- Assertion Checking
- Beta Testing
- Bottom-Up Testing
- Comparison Testing
- *Compliance Testing*
  - Authorization Testing
  - Performance Testing
  - Security Testing
  - Standards Testing
- Debugging
- *Execution Testing*
  - Execution Monitoring
  - Execution Profiling
  - Execution Tracing
- Fault/Failure Insertion Testing
- Field Testing
- Functional (Black-Box) Testing
- Graphical Comparisons
- *Interface Testing*
  - Data Interface Testing
  - Model Interface Testing
  - User Interface Testing
- Object-Flow Testing
- Partition Testing
- Predictive Validation
- Product Testing
- Regression Testing
- Sensitivity Analysis
- *Special Input Testing*
  - Boundary Value Testing
  - Equivalence Partitioning Testing
  - Extreme Input Testing
  - Invalid Input Testing
  - Real-Time Input Testing
  - Self-Driven Input Testing
  - Stress Testing
  - Trace-Driven Input Testing
- *Statistical Techniques*
- *Structural (White-Box) Testing*
  - Branch Testing
  - Condition Testing
  - Data Flow Testing
  - Loop Testing
  - Path Testing
  - Statement Testing
- Submodel/Module Testing
- Symbolic Debugging
- Top-Down Testing
- Visualization/Animation

Formal

- Induction
- Inductive Assertions
- Inference
- Lambda Calculus
- Logical Deduction
- Predicate Calculus
- Predicate Transformation
- Proof of Correctness

Informal and Formal VV&T Techniques

- **Informal VV&T techniques** are called informal because the tools and approaches used rely heavily on human reasoning and subjectivity without stringent mathematical formalism.
  - The "informal" label does not imply any lack of structure or formal guidelines for the use of the techniques.
  - In fact, these techniques are applied using well structured approaches under formal guidelines and they can be very effective especially during the early stages of the M&S life cycle.
- **Formal VV&T techniques** are based on mathematical proof of correctness.
  - If attainable, proof of correctness is the most effective means of M/S V&V.

Static and Dynamic VV&T Techniques

- **Static VV&T techniques** are concerned with accuracy assessment on the basis of characteristics of the static model design and source code.
  - Static techniques do not require computer execution of the model, but mental execution can be used.
- **Dynamic VV&T techniques** require model execution and are intended for evaluating M/S based on execution behavior.
  - Many dynamic VV&T techniques require model instrumentation.
  - The insertion of additional code (probes or stubs) into the executable model or submodel (module) for the purpose of collecting information about model behavior during execution is called **model instrumentation**.
  - Probe locations are determined manually or automatically based on static analysis of model structure.

Informal VV&T Techniques

- Audit
- Desk Checking
- Documentation Checking
- Face Validation
- Inspections
- Reviews
- Turing Test
- Walkthroughs

Audit

- Audit seeks to determine through investigation the adequacy of the overall development process with respect to established practices, standards, and guidelines.
- Audit is accomplished through a mixture of meetings, observations, and examinations.
- Audit is performed by a single Auditor.
- Auditing can consist of other audits, reviews, and even some testing.
- Audit is carried out on a periodic basis.
- Audit also seeks to establish traceability within the development process. Given an error in a part of the model, the error should be traceable to its source via its audit trail.

Desk Checking

- Very first step in VV&T and is particularly useful for the early stages of development.
- Most traditional means for analyzing a program by hand while sitting at one's desk.
- Desk checking becomes much more effective if it is conducted by another person or group of people. Commonly the developer becomes blinded to his or her own mistakes.

Documentation Checking

- Deals with the assessment of the quality of all aspects of documentation other than the quality of its content.
- The following document quality characteristics are assessed:
  - Accessibility
  - Accuracy (which is measured in terms of Verity and Validity)
  - Completeness
  - Clarity (which is measured in terms of Unambiguity and Understandability)
  - Maintainability
  - Portability
  - Readability

Face Validation

- The project team members, potential users of the model, people knowledgeable about the system under study, based on their estimates and intuition, subjectively compare model and system behaviors under identical input conditions and judge whether the model and its results are reasonable.
- Face Validation is useful as a preliminary approach to validation.

Inspections

- Inspections are conducted by a team of four to six members for any model development work product such as requirements specification, design specification, or code.
- For example, in the case of design inspection, the team consists of:
  - *Moderator:* manages the inspection team and provides leadership
  - *Reader:* narrates the model design and leads the team through it
  - *Recorder:* produces a written report of detected faults
  - *Designer:* is the representative of the development team which created the model design
  - *Implementer:* translates the model design into code
  - *Tester:* SQA group representative

Inspections

- An inspection goes through five distinct phases:
  1. *Overview:* In phase I, the designer gives an overview of the module design to be inspected. The module characteristics such as purpose, logic and interfaces are introduced and related documentation is distributed to all participants to study.
  2. *Preparation:* In phase II, the team members prepare individually for the inspection by examining the documents in detail. The moderator arranges the inspection meeting with an established agenda and chairs it in phase III.
  3. *Inspection:* In phase III, the reader narrates the module design documentation and leads the team through it. The inspection team is aided by a checklist of queries during the fault finding process. The objective is to find and document the faults, not to correct them. The recorder prepares a report of detected faults immediately after the meeting.

Inspections

- An inspection goes through five distinct phases:
  4. *Rework:* In phase IV, the designer resolves all faults and problems specified in the written report.
  5. *Follow-up:* In the final phase, the moderator ensures that all faults and problems have been resolved satisfactorily. All changes must be examined carefully to ensure that no new errors have been introduced as a result of a fix.

Reviews

- The review is conducted in a similar manner as the inspection and walkthrough except in the way the team members are selected.
- The review team also involves managers.
- The review is intended to:
  - give management and the sponsor evidence that the model development process is being carried out according to stated requirements and project objectives.
  - evaluate the model in light of development standards, guidelines, and specifications.
- As such, the review is a higher level technique than the inspections and walkthroughs.

Reviews

- Each review team member examines the model documentation prior to the review.
- The team then meets to evaluate the model relative to specifications and standards, recording defects and deficiencies.
- The result of the review is a document portraying the events of the meeting, deficiencies identified, and review team recommendations.
- Appropriate action may then be taken to correct any deficiencies.
- As opposed to inspections and walkthroughs, which concentrate on correctness assessment, reviews seek to ascertain that tolerable levels of quality are being attained.

Turing Test

- The Turing Test is based on the subject matter expert (SME) knowledge about the system under study.
- The SMEs are presented with two sets of output data obtained, one from the model and one from the system, under the same input conditions.
- Without identifying which one is which, the SMEs are asked to differentiate between the two.
- If they succeed, they are asked how they were able to do it.
- Their response provides valuable feedback for correcting model representation.
- If they cannot differentiate, our confidence in model validity is increased.

Walkthroughs

- Also called Structured Walkthroughs
- The Walkthrough Team consists of a moderator, model developer, and 3 to 6 other members.
- Except the model developer, all other members should not be directly involved in the model development effort.

Walkthroughs

- Roles of the team members in a structured walkthrough:
  - *Presenter:* most often is the model developer.
  - *Coordinator:* Organizes, moderates, and follows up the walkthrough activities. Is usually from the SQA department.
  - *Scribe:* Documents the events of the meeting.
  - *Maintenance Oracle:* Considers long-term implications.
  - *Standards Bearer:* Concerned with adherence to standards.
  - *User Representative:* Reflects the needs and concerns of the sponsor.
  - *Other Reviewers:* (e.g., auditors)
- The presenter provides test data and leads the team through a manual simulated execution of the model system.
- The test data are walked through the system.
- The purpose is to encourage discussions. Most errors are discovered by questioning the developer's decisions at various stages.

Static VV&T Techniques

- Cause-Effect Graphing
- *Control Analysis*
  - Calling Structure Analysis
  - Concurrent Process Analysis
  - Control Flow Analysis
  - State Transition Analysis
- *Data Analysis*
  - Data Dependency Analysis
  - Data Flow Analysis
- Fault/Failure Analysis
- *Interface Analysis*
  - Model Interface Analysis
  - User Interface Analysis
- Semantic Analysis
- Structural Analysis
- Symbolic Evaluation
- Syntax Analysis
- Traceability Assessment

Cause-Effect Graphing

- Assists accuracy assessment by addressing the question of "what causes what in the model representation?"
- It is performed by first identifying causes and effects in the problem domain being represented and by examining if they are accurately reflected in the model specification.
- As many causes and effects as possible are listed and the semantics are expressed in a cause-effect graph.
- The graph is annotated to describe special conditions or impossible situations.
- Once the cause-effect graph has been constructed, a decision table is created by tracing back through the graph to determine combinations of causes which result in each effect.
- The decision table is then converted into test cases with which the model is tested.

Control Analysis: Calling Structure Analysis

- is used to assess model accuracy by identifying who calls who and who is called by who.
- The “who” could be a module, procedure, subroutine, function, or a method in an object-oriented model.
- In an object-oriented model, inaccuracies such as the ones given below can be revealed by analyzing which methods invoke a method and by which methods a method is invoked.
  - Inaccuracies caused by message passing.
  - Inaccuracies caused by invoking an invalid method.
  - Inaccuracies caused by calling an object which does not exist.

Control Analysis: Concurrent Process Analysis

- model accuracy is assessed by analyzing the overlap or concurrency of model components executed in parallel or as distributed.
- Such analysis can reveal synchronization problems such as deadlocks.

Control Analysis: Control Flow Analysis

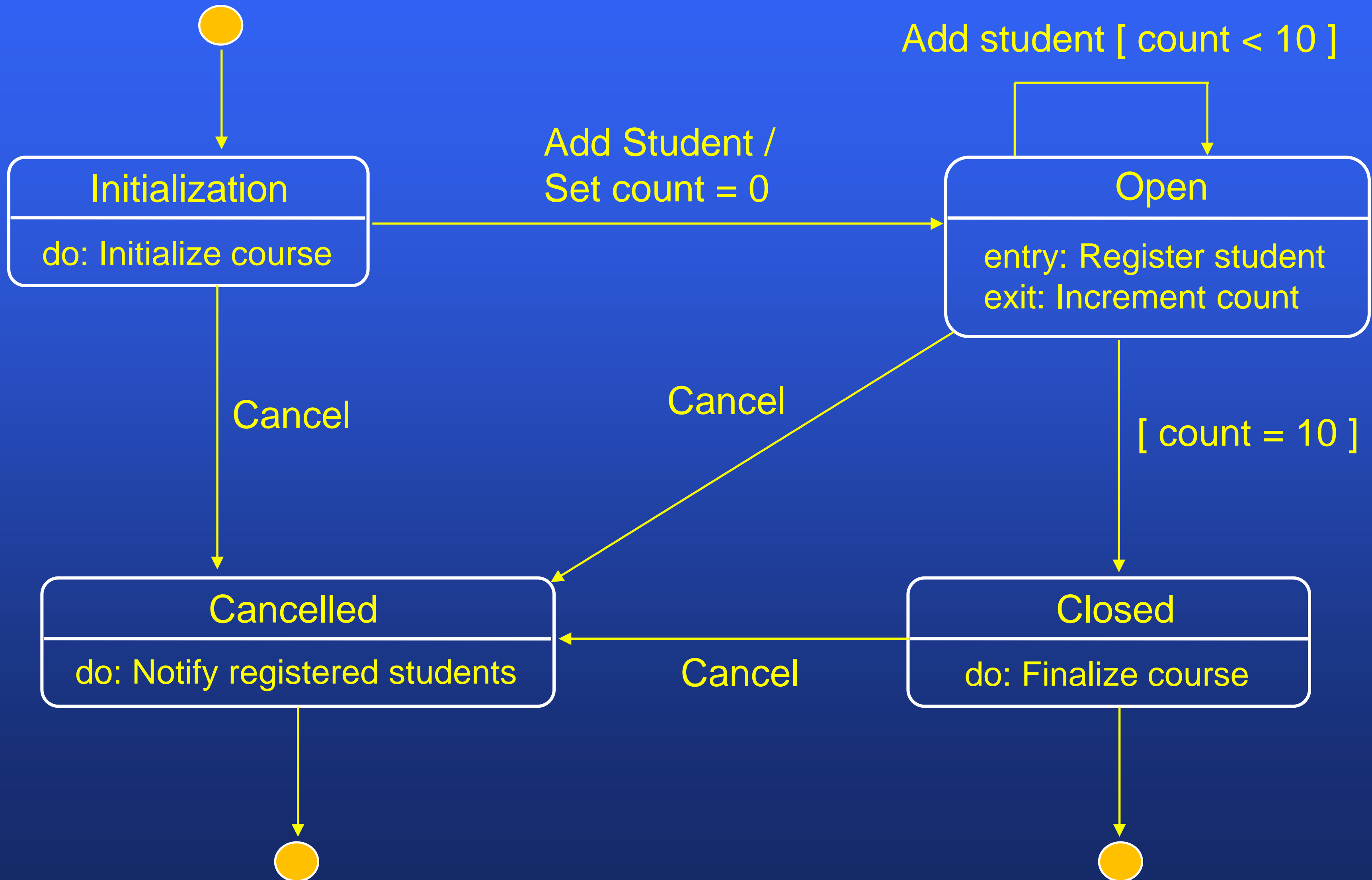
- This technique requires the development of a graph of the model where conditional branches and model junctions are represented by nodes and the model segments between such nodes are represented by links.
- A node of the model graph usually represents a logical junction where the flow of control changes, while an edge represents towards which junction it changes.
- This technique examines sequences of control transfers and is useful for identifying incorrect or inefficient constructs within model representation.

Control Analysis: State Transition Analysis

- This technique requires the identification of a finite number of states the model execution goes through.
- A state transition diagram is created showing how the model transitions from one state to another.
- model accuracy is assessed by analyzing the conditions under which a state change occurs.
- The analysis deals with questions such as
  - Have all states been defined?
  - Can all the states be reached?
  - Can all the states be left?
  - Does each state respond properly to all possible conditions?

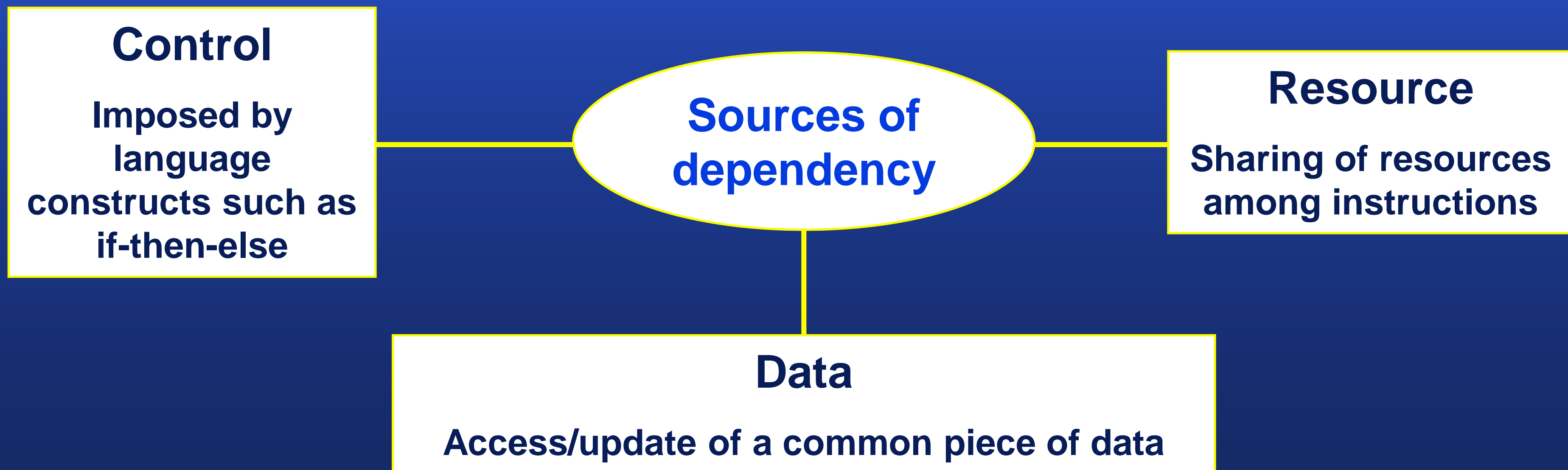
Control Analysis: State Transition Analysis

Example State Transition Diagram

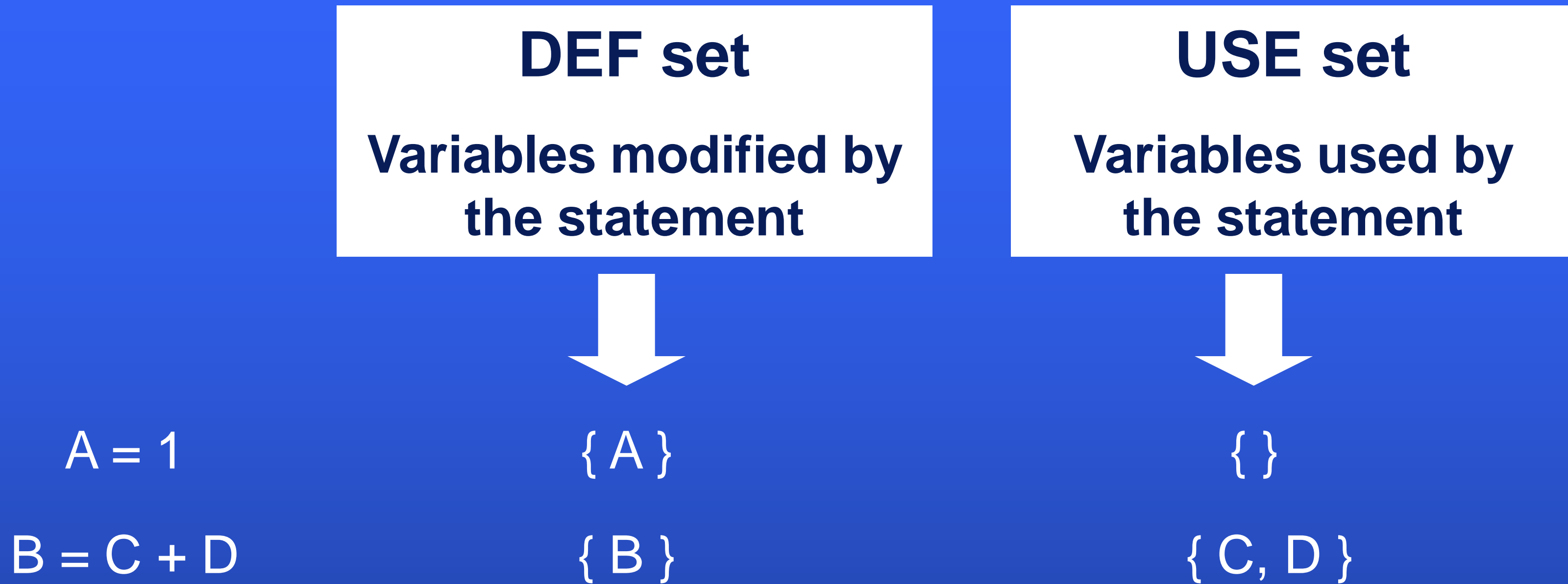


Data Analysis: Data Dependency Analysis

- Program statements / procedures are not generally independent.
- Involves the determination of what variables depend on what other variables.
- In parallel and distributed applications, data dependency knowledge is critical for assessing the accuracy of process synchronization.



Data Analysis: Data Dependency Analysis



Data Dependence is largely a function of DEF and USE sets.

Flow Dependence

Intersection of DEF(S1) and USE(S2) is non-empty

S1: A = B + C

S2: D = A +10

Output Dependence

Intersection of USE(S1) and DEF(S2) is empty

S1: A = B + C

S2: A = A - D

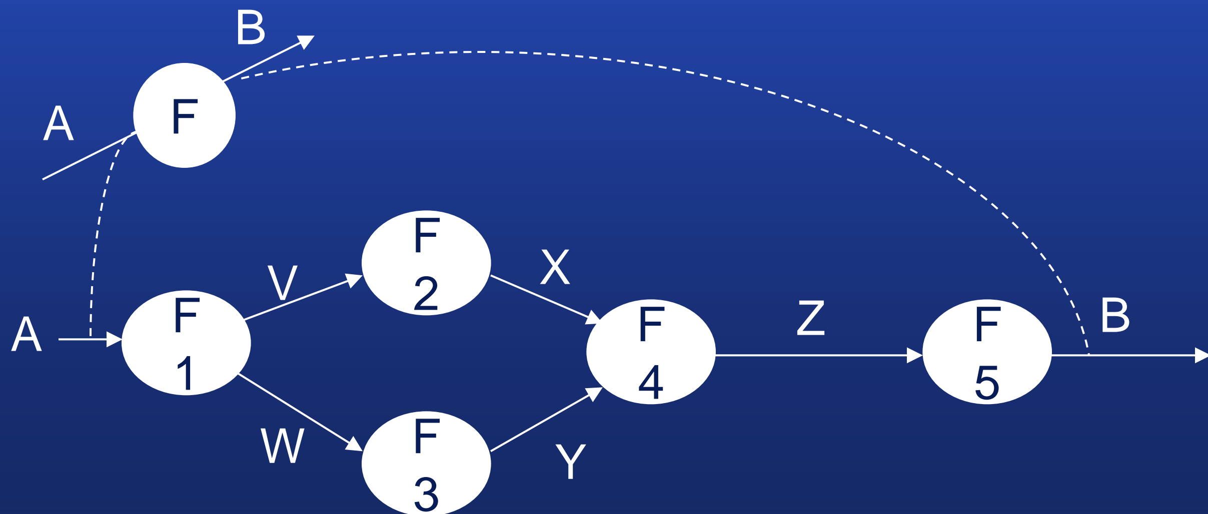
Data Analysis: Data Dependency Analysis

- Most programs involve some array structure / dependence.
- Notion of dependence needs to be extended to handle array subscripts
  - Do i
  - A [2 * i + 1] = B [i] + ……….. (1)
  - D[i] = A [2 * i] ………………. (2)
- Can (1) and (2) be done in parallel?
- Statements 1 and 2 have flow dependence if array A is considered as a single entity.
- Identifying data dependencies helps in program transformations for parallelism.
- A way to overcome the dependencies is found so that the program is suitable for parallel execution.

Data Analysis: Data Flow Analysis

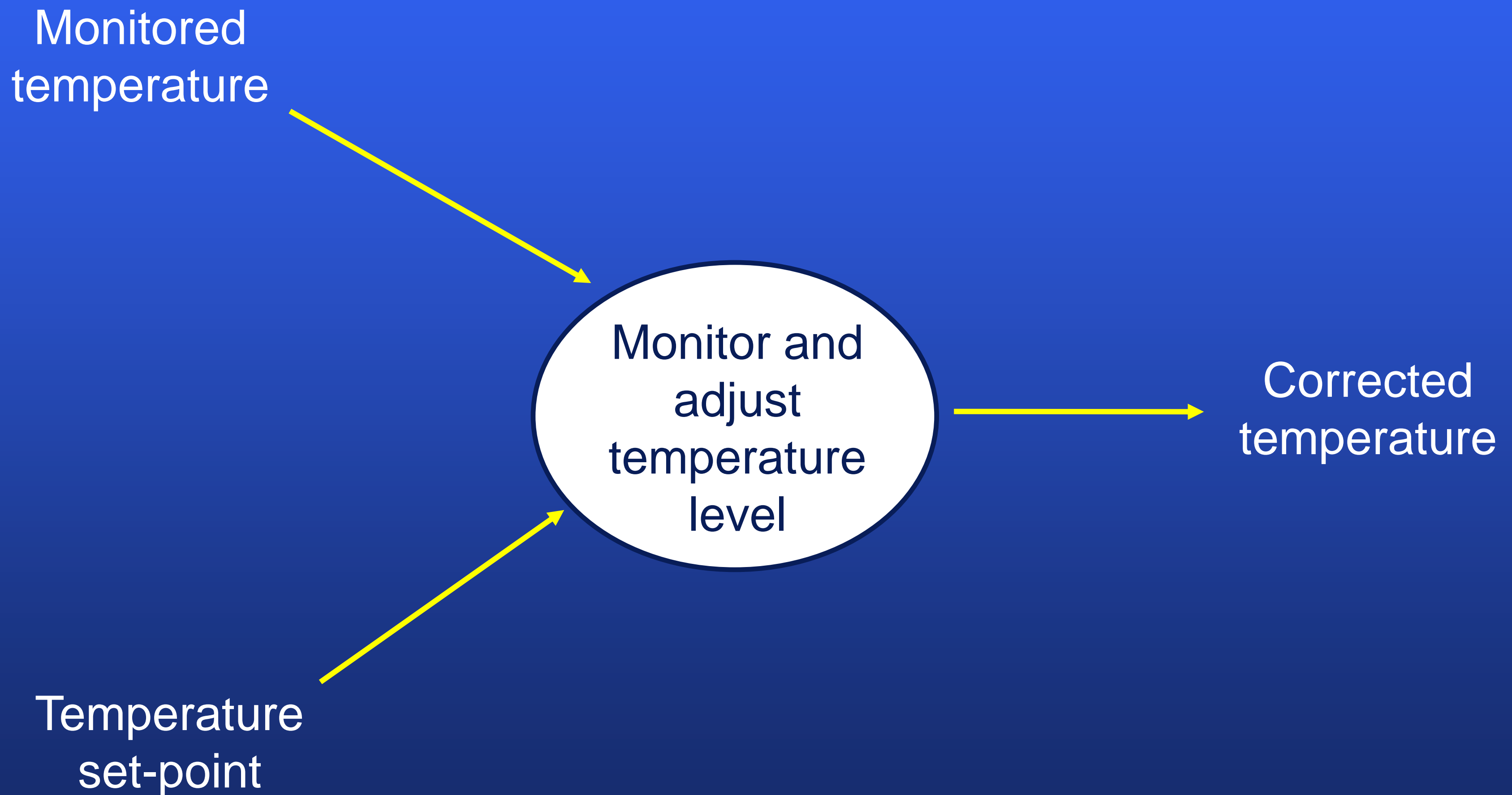
- It is easier to analyze programs, if you know where all the data in the program is defined, used, and altered.
- A data flow graph is constructed to aid data flow analysis.
- Data flow analysis can be used to
  - Detect undefined or un-referenced variables
  - Track maximum and minimum values
  - Track data dependencies and data transformations

As information moves through flows in the model, it is modified by a series of transformations.



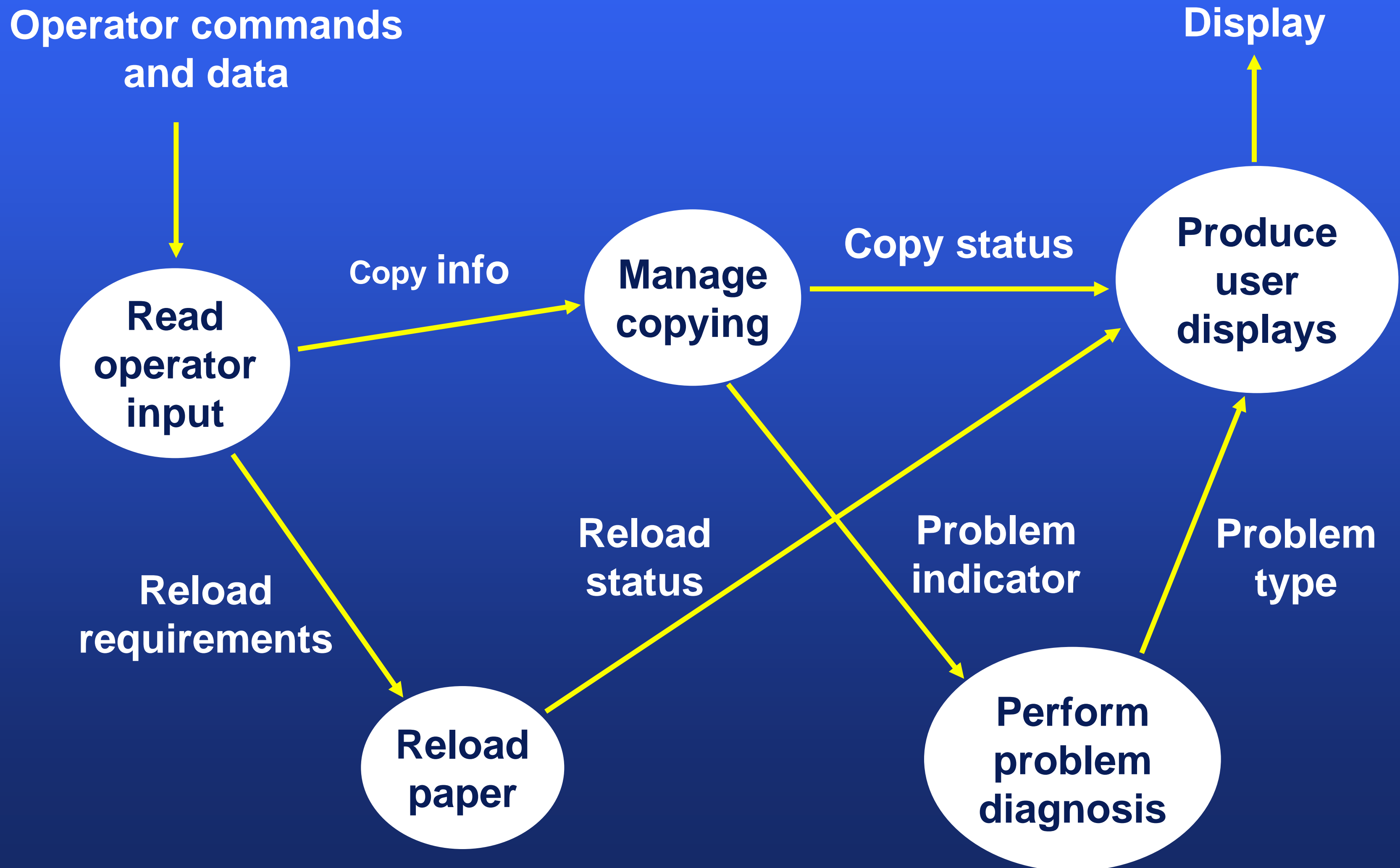
Data Analysis: Data Flow Analysis

An example of data flow in a thermostat



Data Analysis: Data Flow Analysis

An example of data flow in a photocopier software



Fault/Failure Analysis

- Fault refers to incorrect model component.
- Failure refers to incorrect behavior of a model component.
- This analysis uses model input-output transformation descriptions to identify how the model might logically fail.
- The model design specification is examined to determine if any failure-mode possibilities could logically occur and in what context and under what conditions.
- Failure Analysis is the evaluation of defect patterns to learn process or product weakness.
- Failure Analysis includes not only analyzing model defects but also brainstorming the root causes of those defects and incorporating what we learn into training and process changes so that defects won’t occur again.

Interface Analysis

- The interface analysis category consists of model interface analysis and user interface analysis.
- These techniques are especially useful for VV&T of interactive and distributed simulations.
- Model Interface Analysis is conducted to examine the (sub)model-to-(sub)model interface and determine if the interface structure and behavior are sufficiently accurate.
- User Interface Analysis is conducted to examine the user-model interface and determine if it is human engineered so as to prevent occurrences of errors during the user's interactions with the model.
  - It is also used to assess how accurately the interface is integrated with the simulation model.
  - This technique is particularly useful for accuracy assessment of interactive simulation models used for training purposes.

Semantic Analysis

- Is conducted by the programming language compiler and attempts to determine the programmer’s intent in writing the code.
- The compiler informs the programmer about what is specified in the source code so that the programmer can verify that the true intent is accurately reflected.
- The compiler generates a wealth of information to help the programmer determine if the true intent is accurately translated into the executable code:
  - Symbol Tables: describe the elements or symbols that are manipulated in the model, function declarations, type and variable declarations, scoping relationships, interfaces, dependencies, etc.

Semantic Analysis

- **Cross-reference Tables**: describe called versus calling modules (where each data element is declared, referenced and altered), duplicate data declarations (how often and where occurring) and unreferenced source code.
- **Subroutine Interface Tables**: describe the actual interfaces of the caller and the called.
- **Maps**: relate the generated runtime code to the original source code.
- **"Pretty Printers"** or **Source Code Formatters**: provide reformatted source listing on the basis of its syntax and semantics, clean pagination, highlighting of data elements and marking of nested control structures.

Structural Analysis

- Is used to examine the model structure and to determine if it adheres to structured principles.
- It is conducted by constructing a control flow graph of the model structure and examining the graph for anomalies, such as
  - multiple entry and exit points,
  - excessive levels of nesting within a structure, and
  - questionable practices such as the use of unconditional branches.

Symbolic Evaluation

- Is used to assess model accuracy by exercising the model using symbolic values rather than actual data values for input.
- It is performed by feeding symbolic inputs into the module and producing expressions for the output which are derived from the transformation of the symbolic data along model execution paths.

Symbolic Evaluation

Consider, for example, the following function:

```
function jobArrivalTime(arrivalRate,currentClock,randomNumber)
    lag = -10
    Y = lag * currentClock
    Z = 3 * Y
    if Z < 0 then
        arrivalTime = currentClock - log(randomNumber)/arrivalRate
    else
        arrivalTime = Z - log(randomNumber)/arrivalRate
    end if
    return arrivalTime
end jobArrivalTime
```

In symbolic execution, lag is substituted in Y resulting in Y = -10*currentClock. Substituting again, we find Z = -30*currentClock.

Since currentClock is always zero or positive, an error is detected that Z will never be greater than zero.

Syntax Analysis

- Is carried on by the programming language compiler to assure that the mechanics of the language are applied correctly.
- What is Syntax?
  - Textual representation of a language
- Why do Syntax Analysis?
  - To construct a parse tree that determines if the text of the programming language is well formed according to the grammar rules of the programming language.
- Syntactically erroneous code in Pascal:
  - a := a * (b + c * d ;
- Syntactically correct code in Pascal:
  - a := a * (b + c * d);

Traceability Assessment

- Used to match, one to one, the elements of one form of the model to another.
- For example, the elements of the system and requirements specification are matched one to one to the elements of the model design specification.
- Unmatched elements may reveal either unfulfilled requirements or unintended design functions.
- Traceability must be performed throughout the model development life cycle.
- Benefits:
  - Align the changing business needs with the model developed
  - Reduces risk by capturing knowledge that's vital to the project's success
  - Supports process improvement

Example Traceability

#	Reference	Requirement Description	Test Method	Team	Test Objectives	References to SRS Rev B
1	CEI 3.1.3.1 a)-1	Be able to command exposure time between 0.1 and 10 sec	R8	SW	Inspect Detailed Design Specification to check that the times have been implemented.	¤ 3.2.2.3.1 , Table 11
2	CEI 3.1.3.2 a)-a-1-iii	Extract event location, time, and energy	D5	SW	Verify pixel location, time, and energy in timed- exposure faint event mode during SW integration tests, using "image loader" and test images (from CD-R set).	¤ 3.2.2.3.1 , Table 11 ¤ 3.2.2.3.21 , Table 20

Dynamic VV&T Techniques

- Acceptance Testing
- Alpha Testing
- Assertion Checking
- Beta Testing
- Bottom-Up Testing
- Comparison Testing
- *Compliance Testing*
  - Authorization Testing
  - Performance Testing
  - Security Testing
  - Standards Testing
- Debugging
- *Execution Testing*
  - Execution Monitoring
  - Execution Profiling
  - Execution Tracing
- Fault/Failure Insertion Testing
- Field Testing
- Functional (Black-Box)Testing
- Graphical Comparisons
- *Interface Testing*
  - Data Interface Testing
  - Model Interface Testing
  - User Interface Testing
- Object-Flow Testing
- Partition Testing
- Predictive Validation
- Product Testing
- Regression Testing
- Sensitivity Analysis
- *Special Input Testing*
  - Boundary Value Testing
  - Equivalence Partitioning Testing
  - Extreme Input Testing
  - Invalid Input Testing
  - Real-Time Input Testing
  - Self-Driven Input Testing
  - Stress Testing
  - Trace-Driven Input Testing
- *Statistical Techniques*
- *Structural (White-Box) Testing*
  - Branch Testing
  - Condition Testing
  - Data Flow Testing
  - Loop Testing
  - Path Testing
  - Statement Testing
- Submodel/Module Testing
- Symbolic Debugging
- Top-Down Testing
- Visualization/Animation

Acceptance Testing

- Used either by the client organization, by the developer's SQA group in the presence of client representatives, or by an independent contractor hired by the client after the model is officially delivered and before the client officially accepts the delivery.
- The model is operationally tested by using the actual hardware and actual data to determine whether all requirements specified in the legal contract are satisfied.

Alpha Testing

- Refers to the operational testing of the alpha version of the complete model at an in-house site which is not involved with the model development.
- Users selected within the model development company are asked to conduct Alpha Testing, in a way independent of the development department.

Assertion Checking

- An assertion is a statement that should hold true as the model executes.
- Assertion Checking is a verification technique used to check what *is* happening against what the model engineer *assumes* is happening so as to guard model execution against potential errors.
- The assertions are placed in various parts of the model to monitor model execution. They can be inserted to hold true
  - globally—for the whole model;
  - regionally—for some modules;
  - locally—within a module; or
  - at entry and exit of a module.

Assertion Checking

- Consider, for example, the following pseudo-code:

```
Base := Hours * PayRate;
Gross := Base * (1 + BonusRate);
```

- In just these two simple statements, several assumptions are being made. It is assumed that

  Hours, PayRate, Base, BonusRate and Gross are all ≥ 0

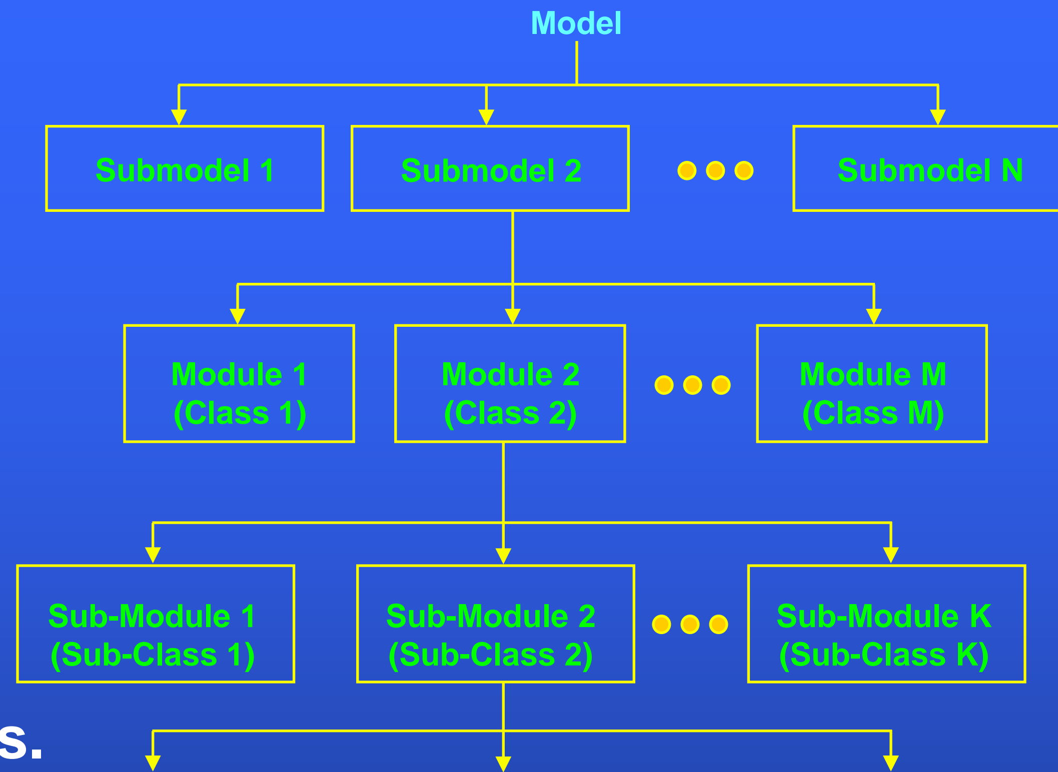
- The following asserted code can be used to prevent execution errors due to incorrect values inputted by the user:

```
Assert Local (Hours ≥ 0 and PayRate ≥ 0 and BonusRate ≥ 0);
Base := Hours * PayRate;
Gross := Base * (1 + BonusRate);
```

Beta Testing

- Refers to the operational testing of the beta version of the complete model at a “beta” user site under realistic field conditions.
- Experienced, knowledgeable users are selected as the Beta Testers.
- Beta Testers generally agree to report the bugs they discover.

Bottom-up Testing



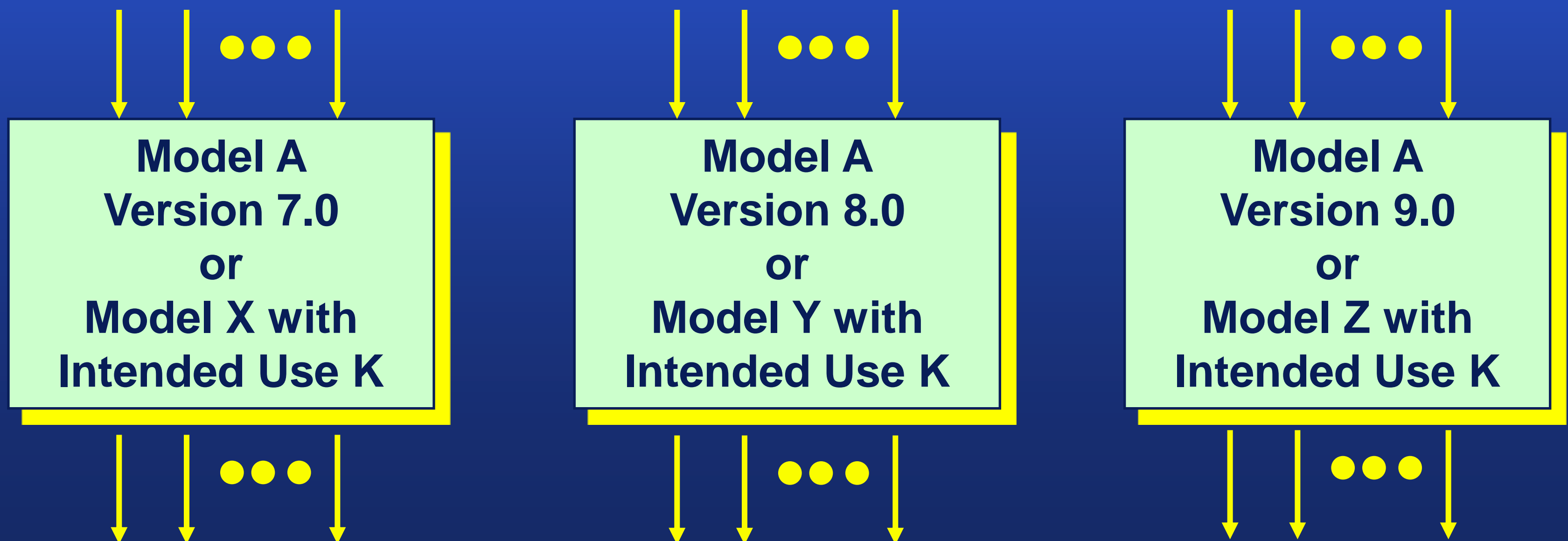
- Testing is conducted in the same pattern as bottom-up development.
- Begin testing the modules at the lowest level and then proceed to testing the higher levels.
- As each module is completed, it is thoroughly tested. When the modules comprising a level have been coded and tested, the modules are integrated and integration testing is performed. This process is repeated until the entire model has been integrated and tested.
- The integration of completed modules need not wait for all “same level” modules to be completed. Module integration and testing can be, and often is, performed incrementally.

Bottom-up Testing

- **Advantages:**
  - Encourages extensive testing at the module level.
  - Since most well-structured model consists of many modules, there is much to be gained by bottom-up testing.
  - The smaller the module and more limited its function, the easier and more complete its testing can be.
  - Particularly attractive for testing distributed systems.
- **Disadvantages:**
  - The need for individual module drivers to test the modules. These drivers, called test harnesses, simulate the calling of the module and pass data necessary to exercise the module.
  - Developing harnesses for every module can be quite large.
  - The harnesses may themselves contain errors.
  - Faces the same cost and complexity issues as does top-down testing.

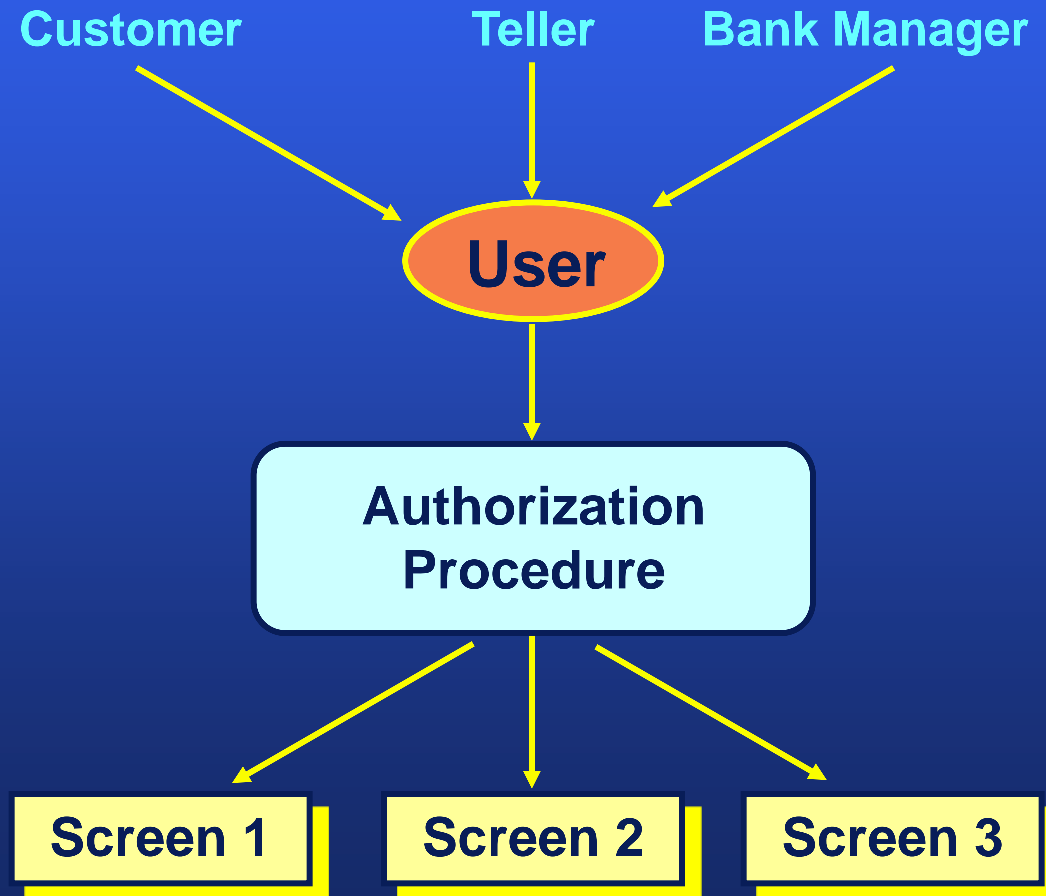
Comparison Testing

- **Comparison Testing** (also known as **back-to-back testing**) is used when more than one version of the same model or more than one model with the same intended use is available for testing.
- Different versions of the same model or different models with the same intended use are executed with the same input data and outputs are compared for accuracy assessment.



Compliance Testing: Authorization Testing

- Used to test how accurately and properly different levels of access authorization are implemented in the model and how properly they comply with the established rules and regulations.
- Testing procedure:
  - Log on as a type of authorized user and test the model behavior.
  - Record the results.
  - Match the results with rules and regulations set by the client.



Compliance Testing: Performance Testing

- Used to test whether
  - all run-time performance characteristics are measured and evaluated with sufficient accuracy, and
  - all established performance requirements are satisfied.
- Accomplished in parallel with Stress Testing and often requires hardware and/or model instrumentation.
- For example:
  - If a transaction is required to take place within a specified time, this test performs that transaction and verifies whether the timing requirement is met.
  - If a system interacts with a large database, this test can be used to examine the speed and accuracy of the data retrieval.

Compliance Testing: Security Testing

- Used to test whether all security procedures are correctly and properly implemented in using the M/S.
- For example, the test can be conducted by attempting to penetrate into the M/S while it is available for use and break into classified components such as secure databases.
- Security testing is applied to substantiate the accuracy and evaluate the adequacy of the protective procedures and countermeasures.

Compliance Testing: Standards Testing

- Used to substantiate that the model is developed with respect to the required standards, procedures, and guidelines.
- Any set of standards can be chosen as the basis for testing.

Debugging

- Testing reveals the presence of errors, debugging finds them and removes them.
- Involves the following steps:
  1. Test the model to reveal an error if any
  2. If error found, locate the source of error
  3. Determine the needs for correcting the error
  4. Make the correction
  5. Retest to ensure successful modification (because a change correcting an error may create another error)
- Isolating the true source of the error (step 2) may be very difficult. Frequently, what may appear to be the source of the error is but an extension of a deeper problem.

Execution Testing: Execution Monitoring

- Provides low-level information about activities and events which took place during execution.
- May provide information about:
  - number of times a certain message is executed,
  - how long it took to perform a certain task,
  - how many times a page fault occurred, etc.
- Requires model instrumentation

Execution Testing: Execution Profiling

- Provides high-level information (a profile) about activities and events which took place during execution.
- May provide information about:
  - how many times a submodel is executed,
  - how many times a global variable is referenced,
  - how many times a source line is executed, etc.
- Gives its results directly in terms of the source definition
- Requires model instrumentation

Execution Testing: Execution Tracing

- Revealing errors by “watching” the line-by-line execution activity of the model.
- Often associated with interpretive languages that offer source level tracing by simply displaying the source statement being interpreted at the given moment.
- In compiled languages, execution tracing can be facilitated via model instrumentation.

Fault/Failure Insertion Testing

- Fault refers to incorrect model component.
- Failure refers to incorrect behavior of a model component.
- Used to deliberately insert a kind of fault or a kind of failure into the model and observe whether the model produces the invalid behavior as expected.
- Unexplained behavior may reveal errors in model representation.

Field Testing

- Field Testing places the model in an operational situation for the purpose of collecting as much information as possible for model validation.
- It is especially useful for validating models of military combat systems.
- Although it is usually difficult, expensive and sometimes impossible to devise meaningful field tests for complex systems, their use wherever possible helps both the project team and decision makers to develop confidence in the model.

Functional (Black-Box) Testing

- Synonyms for black-box testing include:
  - Behavioral testing
  - Functional testing
  - Opaque-box testing
  - Closed-box testing
- Black-box testing treats the model as a “black-box” and it does not explicitly use knowledge of the internal structure.
- Black-box testing is usually described as focusing on testing functional requirements.

Functional (Black-Box) Testing

Deals with the verification and validation of the input-output transformation of the model (module).

- The concern is not what is in the box (i.e., model); rather, what is produced by the box for a given set of inputs.
- The generated test data are inputted into the model (module) and its output is examined to verify and validate the input-output transformation.
- It is virtually impossible to test all inputs to the model.
- Test data may be generated as: (a) minimum and/or maximum values of input variables, (b) randomly generated values, and (c) containing invalid input values deliberately.
- Determining if the test set is complete is the main drawback to black-box testing. Law of large numbers does not apply!

Graphical Comparisons

- Graphical Comparisons is a subjective, inelegant and heuristic, yet quite practical approach especially useful as a preliminary approach to model VV&T.
- The graphs of values of model variables over time are compared with the graphs of values of system variables to investigate characteristics such as
  - similarities in periodicities,
  - skewness,
  - number and location of inflection points,
  - logarithmic rise and linearity,
  - phase shift,
  - trend lines, and
  - exponential growth constants.

Interface Testing: Data Interface Testing

- Used to assess the accuracy of data inputted into the model or outputted from the model during execution.
- All data interfaces are examined to substantiate that all aspects of data input/output are correct.
- This form of testing is particularly important for those model the inputs of which are read from a database and/or the results of which are stored into a database for later analysis.
- The model's interface to the database is examined to ensure correct importing and exporting of data.

Interface Testing: Model Interface Testing

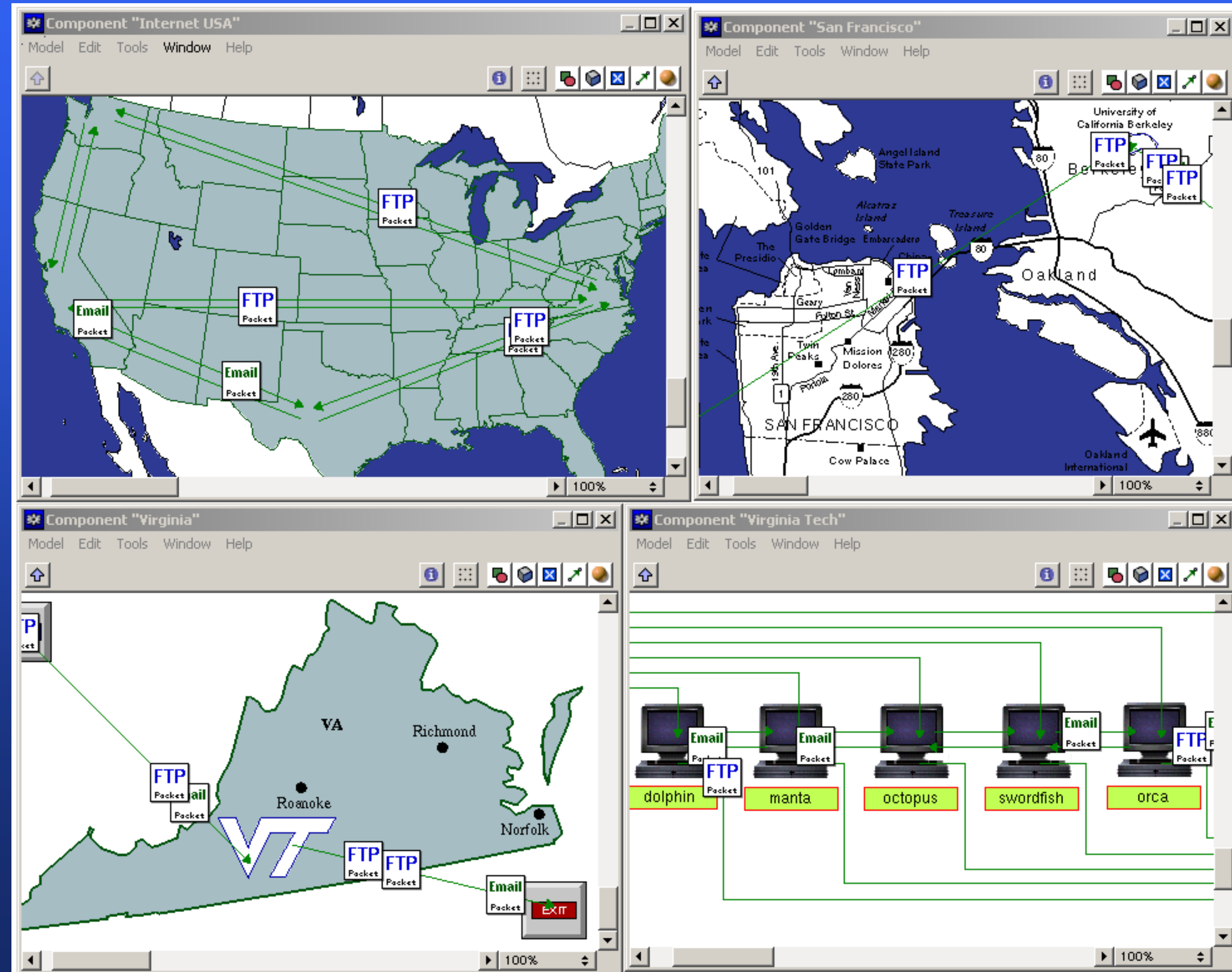
- Used to detect errors created as a result of module-to-module interface or invalid assumptions about the interfaces.
- It is assumed that each model component (module) is individually tested and found to be sufficiently accurate before model interface testing begins.
- This form of testing deals with how well the modules are integrated with each other and is particularly useful for object-oriented and distributed model.
- Under the object-oriented paradigm, objects:
  - are created with public and private interfaces,
  - interface with other objects through message passing,
  - are reused with their interfaces, and
  - inherit the interfaces and services of other objects.

Interface Testing: User Interface Testing

- Used to detect model representation errors created as a result of user-model interface errors or invalid assumptions about the interfaces.
- User interface testing deals with the assessment of the interactions between the user and the model.
- The user interface is examined from low level ergonomic aspects to instrumentation and controls and from human factors to global considerations of usability and appropriateness for the purpose of identifying potential errors.

Object-Flow Testing

- Object-flow testing is similar to transaction-flow testing and thread testing.
- It is used to assess model accuracy by way of exploring the life cycle of an object during model execution.



Partition Testing

- Used for testing the model with the test data generated by analyzing the model's functional representatives (partitions).
- It is accomplished by:
  1. decomposing both model specification and implementation into functional representatives (partitions),
  2. comparing the elements and prescribed functionality of each partition specification with the elements and actual functionality of corresponding partition implementation,
  3. deriving test data to extensively test the functional behavior of each partition, and
  4. testing the model by using the generated test data.

Predictive Validation

- Predictive Validation requires past input and output data of the system being modeled.
- The model is driven by past system input data and its forecasts are compared with the corresponding past system output data to test the predictive ability of the model.

Product Testing

- Product testing is conducted by the development organization after all modules are successfully integrated (as demonstrated by the interface testing) and before the acceptance testing is performed by the client.
- No contractor wants to be in a situation where the product (model) fails the acceptance test.
- Product testing serves as a means of getting prepared for the acceptance testing.
- As such, the QA group must perform product testing and make sure that all requirements specified in the legal contract are satisfied before delivering the model to the client organization.

Regression Testing

- Seeks to ensure that correcting errors and/or making changes to the model do not create other errors.
- Accomplished by retesting the modified model with a set of previous test cases used.
- Successful regression testing requires planning.
- A plan for performing regression testing must be incorporated in the overall management of model development.

Sensitivity Analysis

- Sensitivity Analysis is performed by systematically changing the values of model input variables and parameters over some range of interest and observing the effect upon model behavior.
- Unexpected effects may reveal invalidity.
- The input values can also be changed to induce errors to determine the sensitivity of model behavior to such errors.
- Sensitivity analysis can identify those input variables and parameters to the values of which model behavior is very sensitive.
- Then, model validity can be enhanced by assuring that those values are specified with sufficient accuracy.

Special Input Testing: Boundary Value Testing

- Used to test model accuracy by using test cases on the boundaries of input equivalence classes.
- A model's input domain can usually be divided into classes of input data (known as equivalence classes) which cause the model to function the same way.
- Example: A DBMS design specification indicates that the DBMS must be able to handle any number of records from 1 through 32,000.
  - Equivalence class 1: Less than 1 record
  - Equivalence class 2: From 1 through 32,000 records
  - Equivalence class 3: More than 32,000 records

Special Input Testing: Boundary Value Testing

Example Test Cases

- Test case 1: 0 records — Member of equivalence class 1 & adjacent to boundary value
- Test case 2: 1 record — Boundary value
- Test case 3: 2 records — Adjacent to boundary value
- Test case 4: 16,000 records — Member of equivalence class 2
- Test case 5: 31,999 records — Adjacent to boundary value
- Test case 6: 32,000 records — Boundary value
- Test case 7: 32,001 records — Member of equivalence class 3 & adjacent to boundary value

Special Input Testing: Equivalence Partitioning Testing

- This technique partitions the model input domain into equivalence classes in such a manner that a test of a representative value from a class is assumed to be a test of all values in that class.
- Example: A DBMS design specification indicates that the DBMS must be able to handle any number of records from 1 through 32,000.
  - Equivalence class 1: Less than 1 record
  - Equivalence class 2: From 1 through 32,000 records
  - Equivalence class 3: More than 32,000 records
- Reduces the total number of testing cases to only one or few tests per equivalence class.

Special Input Testing:
Extreme Input Testing & Invalid Input Testing

- **Extreme Input Testing** is conducted by running / exercising the simulation model by using only minimum values, only maximum values, or arbitrary mixture of minimum and maximum values for the model input variables.
- **Invalid Input Testing** is performed by running / exercising the simulation model under incorrect input data and cases to determine whether the model behaves as expected.
  - Unexplained behavior may reveal model representation errors.

Special Input Testing: Real-Time Input Testing

- Real-Time Input Testing is particularly important for assessing the accuracy of simulation models built to represent embedded real-time systems.
- For example, different design strategies of a real-time software system to be developed to control the operations of the components of a manufacturing system can be studied by simulation modeling.
- The simulation model representing the software design can be tested by way of running it under real-time input data that can be collected from the existing manufacturing system.
- Using real-time input data collected from a real system is particularly important to represent the timing relationships and correlations between input data points.

Special Input Testing: Self-Driven Input Testing

- **Self-Driven Input Testing** is conducted by running / exercising the simulation model under input data randomly sampled from probabilistic models representing random phenomena in a real or futuristic system.
- A probability distribution (e.g., exponential, gamma, weibull) can be fit to collected data or uniform, triangular or beta probability distribution can be used in the absence of data to model random input conditions.
- Then, using random variate generation techniques, random values can be sampled from the probabilistic models to test the model validity under a set of observed or speculated random input conditions.

Special Input Testing: Stress Testing

- Used for testing the model under high workloads to probe its viability under resource overload.
- It is useful for time-dependent real-time systems.
- Example parameters to change to perform Stress Testing:
  - Arrival rate of transactions (messages)
  - Amounts of resources used by transactions (messages) (e.g., CPU time, memory, disk storage, etc.)
- Forces race conditions.
- Totally distorts the normal order of processing, especially processing that occurs at different priority levels.
- Forces the exercise of all system limits, thresholds, or other controls designed to deal with overload situations.
- Greatly increases the number of simultaneous actions.
- Depletes resource pools in extraordinary and un-thought of sequences.

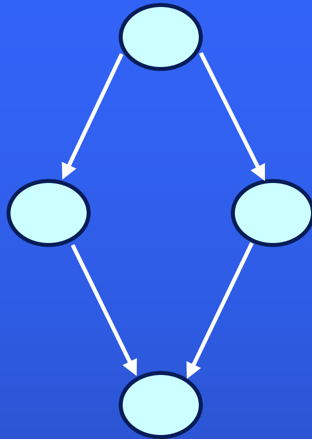
Special Input Testing: Trace-Driven Input Testing

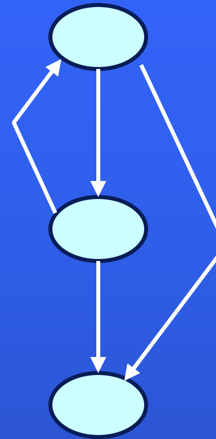
- Trace-Driven Input Testing is conducted by running / exercising the simulation model under input trace data collected from a real system.
- For example, a computer system can be instrumented by using software and hardware monitors to collect data by tracing all system events.
- The raw trace data is then refined to produce the real input data for use in testing the simulation model of the computer system.

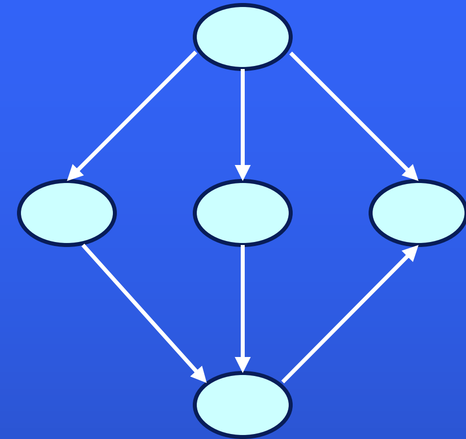
Statistical Techniques Proposed for Model Validation

- Analysis of Variance
- Confidence Intervals/Regions
- Factor Analysis
- Hotelling's T^2 Tests
- Multivariate Analysis of Variance
  - Standard MANOVA
  - Permutation Methods
  - Nonparametric Ranking Methods
- Nonparametric Goodness-of-fit Tests
  - Kolmogorov-Smirnov Test
  - Cramer-Von Mises Test
  - Chi-square Test
- Nonparametric Tests of Means
  - Mann-Whitney-Wilcoxon Test
  - Analysis of Paired Observations
- Regression Analysis
- Theil's Inequality Coefficient
- Time Series Analysis
  - Spectral Analysis
  - Correlation Analysis
  - Error Analysis
- t-Test

Structural (White-Box) Testing: Branch Testing







- **Used to run / exercise the model under test data so as to execute as many branch alternatives as possible, as many times as possible & to substantiate their accurate operations.**
- **The more branches are successfully tested, the more confidence we gain in model's accurate execution with respect to its logical branches.**
- **Aims to ensure that each possible branch from each decision point is executed at least once, thus ensuring that all reachable code is executed.**

Structural (White-Box) Testing: Condition Testing

- Used to execute the model under test data so as to execute as many (compound) logical conditions as possible, as many times as possible and to substantiate their accurate operations.
- Types of Conditions:
  - Simple Conditions:
    - Boolean Variables: A boolean variable, or application of a boolean function.
    - Relational Expressions: comparisons of two values of the same type (e1 op e2).
  - Compound Conditions:
    - Conditions obtained by combining simple conditions by using logical operators such as AND, OR, NOT.

Structural (White-Box) Testing: Condition Testing

Example:

(j >= 1) && (A[j] > A[j+1])

Both of the relational expressions involve an ordered type, so that there are $3^2 = 9$ combinations that should be considered in testing:

(j < 1) && (A[j] < A[j+1])
(j < 1) && (A[j] == A[j+1])
(j < 1) && (A[j] > A[j+1])
(j == 1) && (A[j] < A[j+1])
(j == 1) && (A[j] == A[j+1])
(j == 1) && (A[j] > A[j+1])
(j > 1) && (A[j] < A[j+1])
(j > 1) && (A[j] == A[j+1])
(j > 1) && (A[j] > A[j+1])

Generating Test Cases:

j = 0, A[j] = 1, A[j+1] = 0
j = 0, A[j] = 1, A[j+1] = 1
j = 0, A[j] = 1, A[j+1] = 2
j < 0, A[j] = 1, A[j+1] = 0
j < 0, A[j] = 1, A[j+1] = 1
j < 0, A[j] = 1, A[j+1] = 2
j > 0, A[j] = 1, A[j+1] = 0
j > 0, A[j] = 1, A[j+1] = 1
j > 0, A[j] = 1, A[j+1] = 2

Structural (White-Box) Testing: Data Flow Testing

- This technique uses the control flow graph to explore sequences of events related to the status of data structures and to examine data-flow anomalies.
- For example, sufficient paths can be forced to execute under test data to assure that every data element and structure is initialized prior to use or every declared data structure is used at least once in an executed path.
- A USE is a reference to a variable’s value; a DEF is an assignment of a new value to that variable. A USE-DEF pair is a path from the point the variable is defined to the point the variable is referenced.
- Data flow testing requires that all DEF-USE pairs be executed.

Structural (White-Box) Testing: Data Flow Testing

Example:

```
if (some_exp)              (1)
    some_var = 1;          (2)
else                       (3)
    some_var = 2;          (4)
if (some_case)             (5)
    P1(some_var);          (6)
else                       (7)
    P2(some_var);          (8)
```

DEF: at 2 and 4;

USE: at 6 and 8;

So there are 4 DEF-USE pairs.

No.	DEF	USE	Path
1	2	6	<2-5-6>
2	2	8	<2-5-8>
3	4	6	<4-5-6>
4	4	8	<4-5-8>

Test cases:

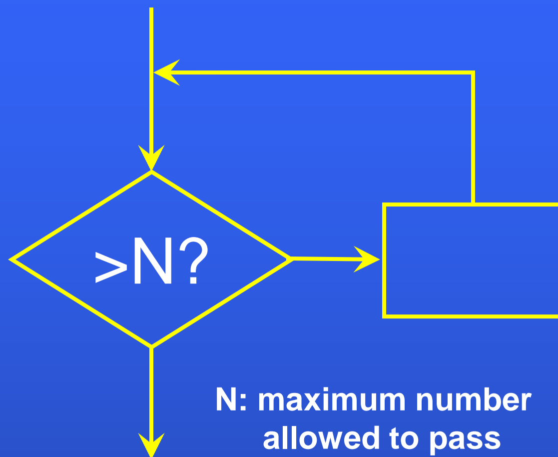
case 1 (some_exp = true, some_case = true) traverses path 1
case 2 (some_exp = true, some_case = false) traverses path 2
case 3 (some_exp = false, some_case = true) traverses path 3
case 4 (some_exp = false, some_case = false) traverses path 4

Structural (White-Box) Testing: Loop Testing

- This technique is conducted to execute the model under test data so as to execute as many loop structures as possible, as many times as possible and to substantiate their accurate operations.
- Loop classes include:
  - Simple, nested, concatenated, and unstructured
- Sets of test should uncover errors at:
  - basic path, initialization, indexing, and bounding

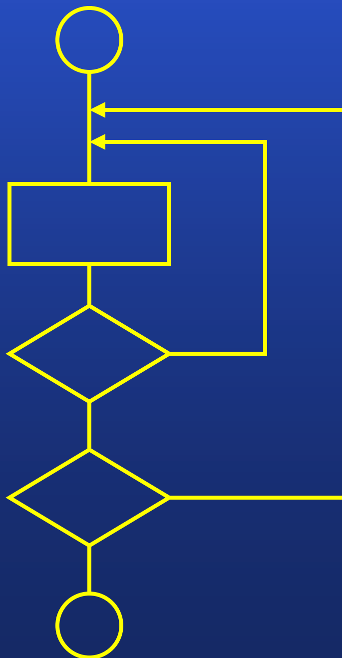
Structural (White-Box) Testing: Loop Testing

Simple Loop:



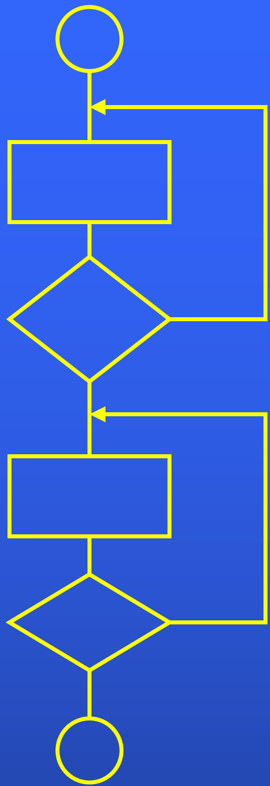
- Skip loop
- one pass through the loop
- two passes through the loop
- m passes where m < n
- n-1, n, n+1 passes through the loop
- excluded values

Nested Loop:



- start from innermost loop
  - simple loop test
  - outer loop holding minimum value
- work outward
  - all outer loops keep minimum value
  - other loops keep typical value

Structural (White-Box) Testing: Loop Testing

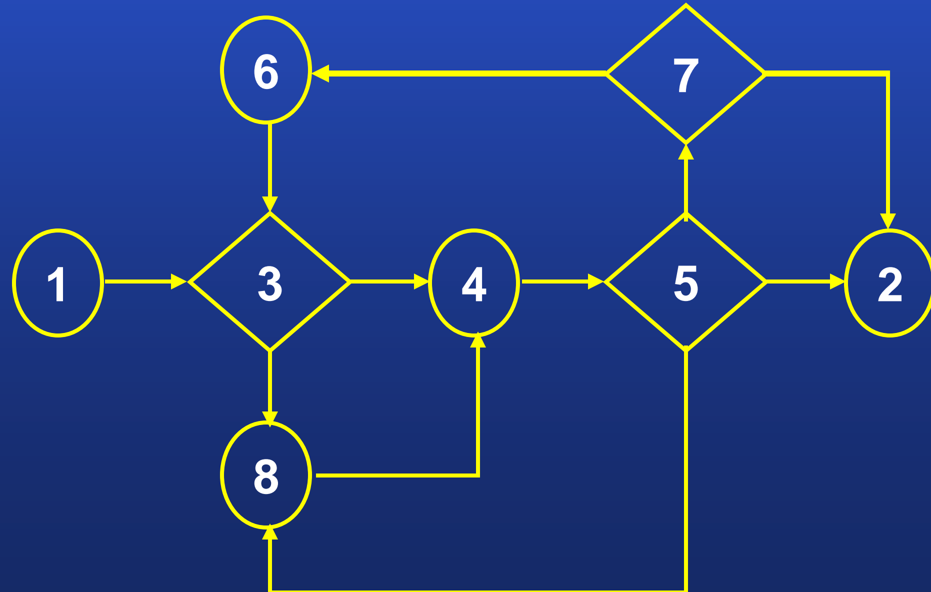


Concatenated Loop:

- One after another but still on a path from entrance to exit
- independent
  - counter in one loop is unrelated to another
  - treat as simple loop
- related
  - counter for one loop as initial value for the other
  - treat as nested loop

Unstructured Loop:

- Should avoid in the first place
- hard to select counter
  - break effect
  - cross-connected loop
  - hidden loop
- things need to be considered
  - end points
  - looping values



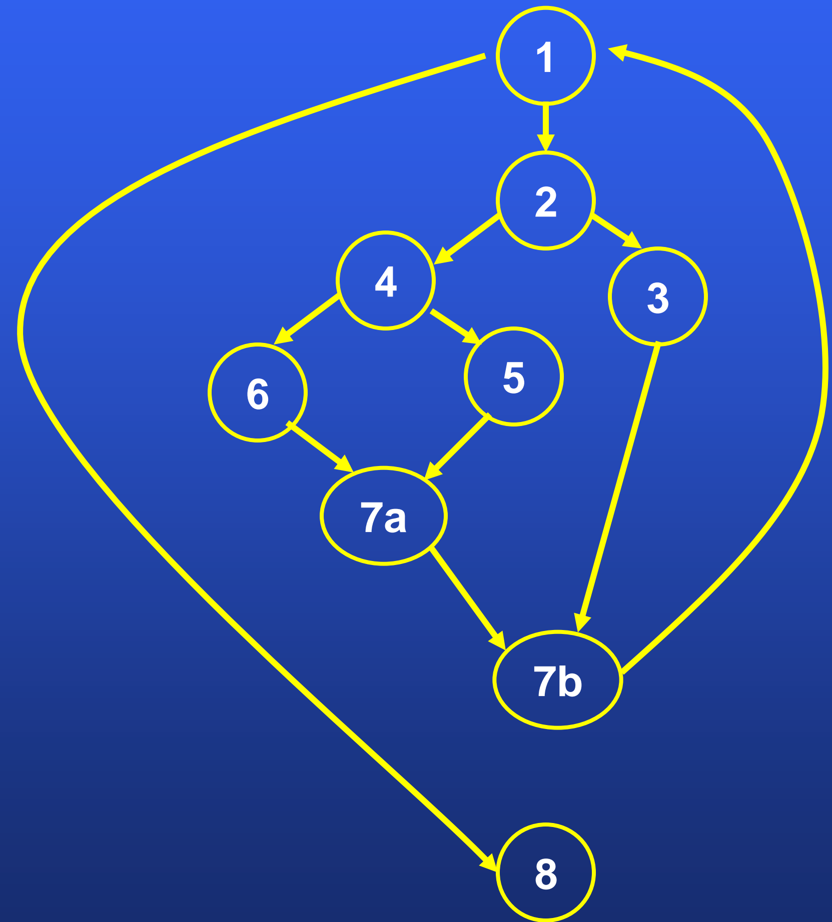
Structural (White-Box) Testing: Path Testing

- What is it?
  - Test based on selecting sets of test paths through the program
- Generate tests from control flow
- Criteria for selecting paths
- Generate test case
- Determine path-forcing input values
- Path Selection Criteria:
  - Complete testing
    - Every path from entry to exit
    - Every statement or instruction
    - Every branch/case statement, in each direction
  - Adaptive Strategy
    - Path prefix strategy
    - Essential branch strategy

Structural (White-Box) Testing: Path Testing

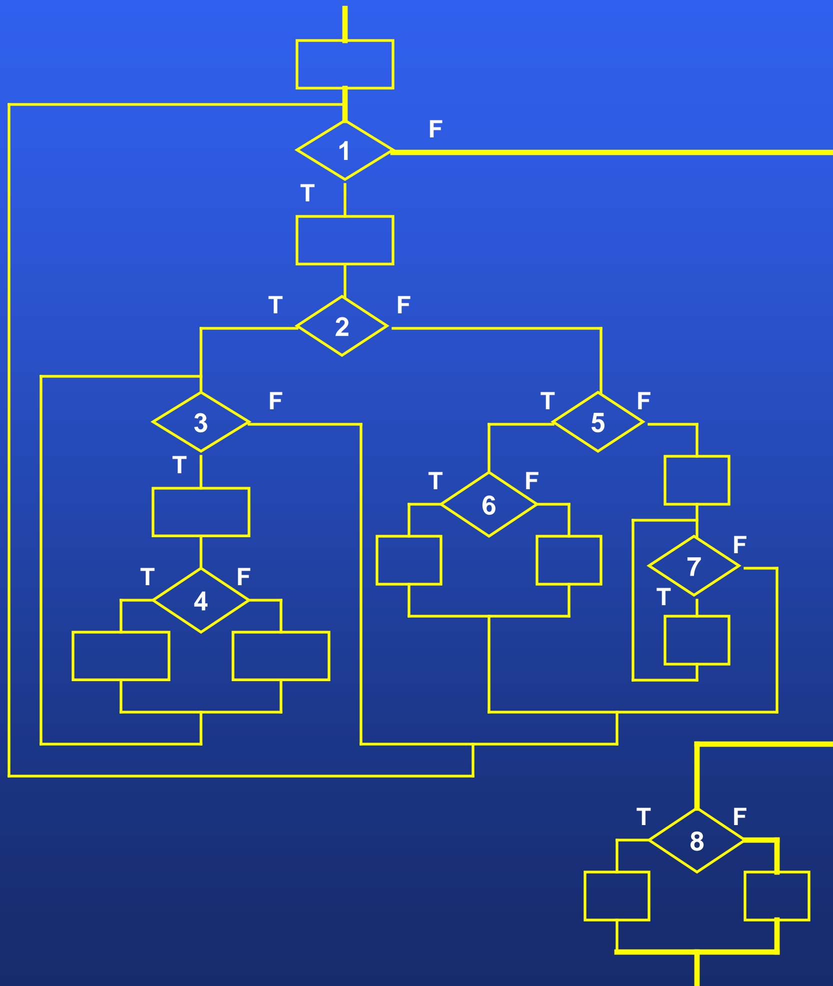
Control Flowgraph

```
1: do while records remain
       read record;
2: if record field 1=0
3: then process record;
       store in buffer;
       increment counter;
4: elseif record field 2=0;
5: then reset counter;
6: else process record;
       store in file;
7a: endif
    endif
7b: enddo
8:end
```



Structural (White-Box) Testing: Path Testing

Path Selection Criteria

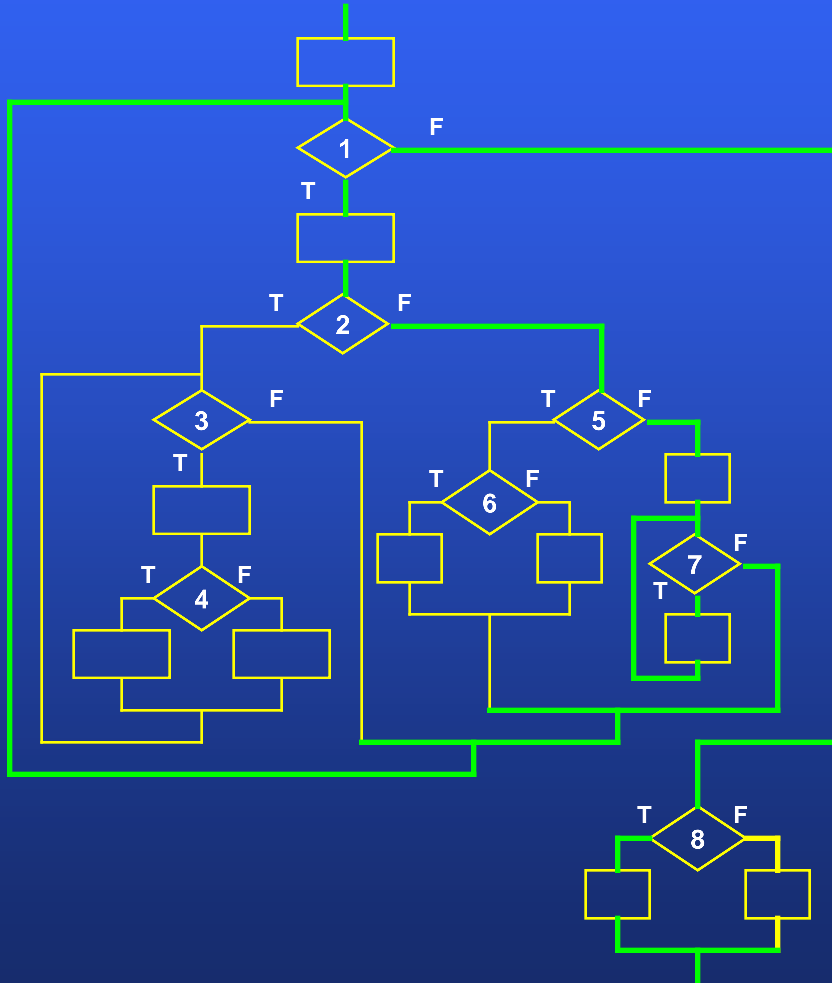


A) 1F8F

	T	F
1		X
2		
3		
4		
5		
6		
7		
8		X

Structural (White-Box) Testing: Path Testing

Path Selection Criteria



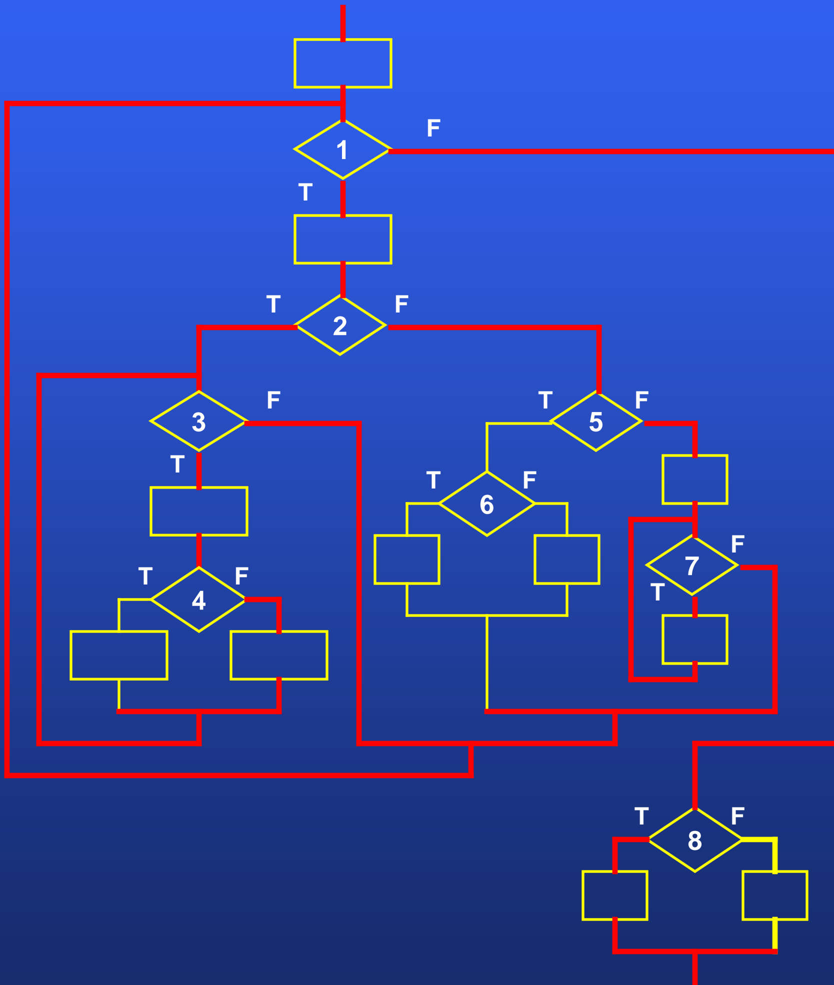
B) 1F → 1T

1T2F5F7T7F1F8T

	T	F
1	X	X
2		X
3		
4		
5		X
6		
7	X	X
8	X	X

Structural (White-Box) Testing: Path Testing

Path Selection Criteria



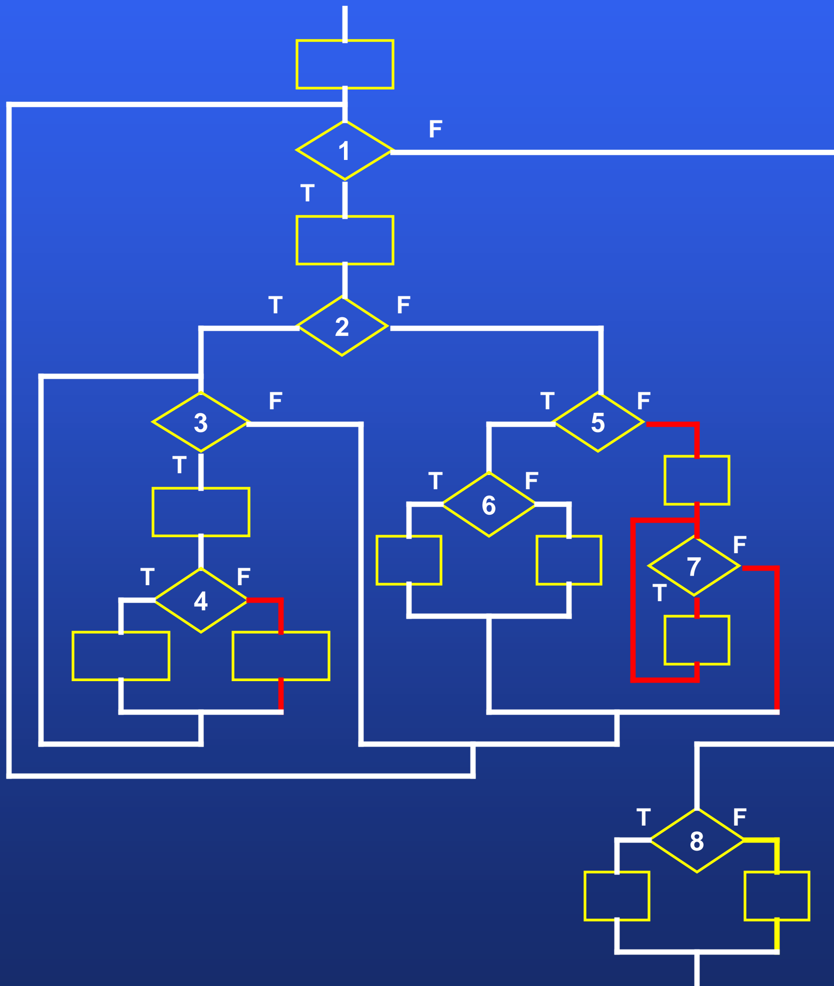
C) 1T2F → 1T2T

1T2T3T4F3F1T2F5F7T7F1F8T

	T	F
1	X	X
2	X	X
3	X	X
4		X
5		X
6		
7	X	X
8	X	X

Structural (White-Box) Testing: Path Testing

Path Selection Criteria



d) two choices

1T2F5F **1T2F3T4F**

shortest one: 1T2F5F → 1T2F5T

1T2F5T6T1T2T3T4T3F1T2F5T6F1F8T

	T	F
1	X	X
2	X	X
3	X	X
4	X	X
5	X	X
6	X	X
7	X	X
8	X	X

Structural (White-Box) Testing: Statement Testing

- Minimum testing requirement of IEEE unit test standard
- Why?
  - Untested code in proportion to probability of bugs
- How?
  - Execute all statements at least once under test

Common Errors: Example

```
x = 5;
y = 6;
z = x++ + y++;

x      y     z
6      7     11
```

```
x = 5;
y = 6;
z = ++x + ++y;

x      y     z
6      7     13
```

NOTE:

- **++n increments n *before* its value is used.**
- **n++ increments n *after* its value is used.**

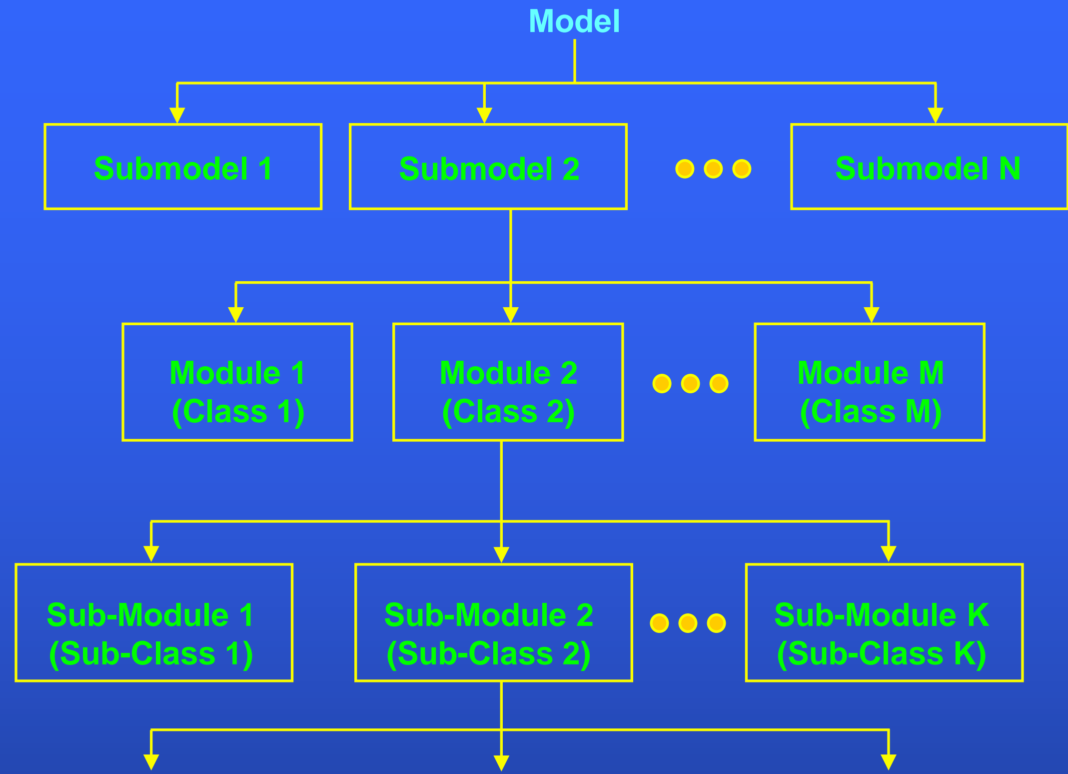
Submodel / Module Testing

- Submodel / Module Testing requires a top-down model decomposition in terms of submodels / modules.
- The executable model is instrumented to collect data on all input and output variables of a submodel.
- The system is similarly instrumented (if possible) to collect similar data.
- Then, each submodel behavior is compared with corresponding subsystem behavior to judge submodel validity.
- If a subsystem can be modeled analytically (e.g., as an M/M/1 model), its exact solution can be compared against the simulation solution to assess validity quantitatively.

Symbolic Debugging

- Using a debugging tool, model execution is manipulated, while viewing the source code, for debugging purposes.
- By setting “breakpoints”, the tester can interact with the entire model one step at a time, at predetermined locations, or under specified conditions.
- While using a symbolic debugger, the tester may alter data values or cause a portion of the model to be “replayed”, i.e., executed again under the same conditions (if possible).
- The tester can set “watch” variables to monitor data flow.
- Typically, the tester utilizes the information from execution history generation techniques, such as tracing, monitoring and profiling, to isolate a problem or its proximity. Then the debugger is employed to understand how and why the error occurs.

Top-Down Testing



- Testing is conducted in the same pattern as top-down development.
- Begin testing the global level and then proceed to testing the lower levels.
- When testing a given level, calls to sublevels are simulated by using "stubs" – a dummy component which has no other function but to let its caller complete the call.

Top-Down Testing

- **Advantages:**
  - Model integration testing is minimized.
  - Early existence of a working model results.
  - Higher level interfaces are tested first.
  - A natural environment for testing lower levels is provided.
  - Errors are localized to new modules and interfaces.
- **Disadvantages:**
  - Thorough module testing is discouraged (the entire model must be executed to perform testing).
  - Testing can be expensive (since the whole model must be executed for each test).
  - Adequate input data is difficult to obtain (because of the complexity of the data paths and control predicates).
  - Integration testing is hampered (again, because of the size and complexity induced by testing the whole model).

Visualization / Animation

- Visualization/Animation of a simulation model greatly assists in model VV&T.
- Displaying graphical images of internal (e.g., how customers are served by a cashier) and external (e.g., utilization of the cashier) dynamic behavior of a model during execution enables us to discover errors by seeing.
- For example, in a traffic intersection visual simulation, we can observe the arrivals of vehicles in different lanes and their movements through the intersection as the traffic light changes.
- Seeing the animation of the model as it executes and comparing it with the behavior of the system being modeled can help us identify discrepancies between the model and the system.
- However, note that, "Seeing is not believing in model validity!"

Formal VV&T Techniques

- Induction
- Inductive Assertions
- Inference
- Lambda Calculus
- Logical Deduction
- Predicate Calculus
- Predicate Transformation
- Proof of Correctness

Induction

- Induction, inference and logical deduction are simply acts of justifying conclusions on the basis of premises given.
- An argument is valid if the steps used to progress from the premises to the conclusion conform to established rules of inference.
- Inductive reasoning is based on invariant properties of a set of observations (assertions are invariants since their value is defined to be true).
- Given that the initial model assertion is correct, it stands to reason that if each path progressing from that assertion can be shown to be correct and subsequently each path progressing from the previous assertion is correct, etc., then the model must be correct if it terminates.
- Formal induction proof techniques exist for the intuitive explanation just given.

Inductive Assertions

- It is conducted in three steps.
  - In Step 1, input-to-output relations for all model variables are identified.
  - In Step 2, these relations are converted into assertion statements and are placed along the model execution paths in such a way as to divide the model into a finite number of “assertion-bound” paths, i.e., an assertion statement lies at the beginning and end of each model execution path.
  - In Step 3, verification is achieved by proving that for each path:
    - if the assertion at the beginning of the path is true and all statements along the path are executed, then the assertion at the end of the path is true.
  - If all paths plus model termination can be proved, by induction, the model is proved to be correct.

Inference

- Induction, inference and logical deduction are simply acts of justifying conclusions on the basis of premises given.
- An argument is valid if the steps used to progress from the premises to the conclusion conform to established *rules of inference*.
- Inductive reasoning is based on invariant properties of a set of observations (assertions are invariants since their value is defined to be true).
- Given that the initial model assertion is correct, it stands to reason that if each path progressing from that assertion can be shown to be correct and subsequently each path progressing from the previous assertion is correct, etc., then the model must be correct if it terminates.
- Formal induction proof techniques exist for the intuitive explanation just given.

Lambda Calculus

- The λ-calculus is a system for transforming the model into formal expressions.
- It is a string-rewriting system and the model itself can be considered as a large string.
- The λ-calculus specifies rules for rewriting strings, i.e., transforming the model into λ-calculus expressions.
- Using the λ-calculus, the model engineer can formally express the model so that mathematical proof of correctness techniques can be applied.

Logical Deduction

- Induction, inference and logical deduction are simply acts of justifying conclusions on the basis of premises given.
- An argument is valid if the steps used to progress from the premises to the conclusion conform to established *rules of inference*.
- Inductive reasoning is based on invariant properties of a set of observations (assertions are invariants since their value is defined to be true).
- Given that the initial model assertion is correct, it stands to reason that if each path progressing from that assertion can be shown to be correct and subsequently each path progressing from the previous assertion is correct, etc., then the model must be correct if it terminates.
- Formal induction proof techniques exist for the intuitive explanation just given.

Predicate Calculus

- The predicate calculus provides rules for manipulating predicates.
- A *predicate* is a combination of simple relations, such as `completed_jobs > steady_state_length`.
- A predicate will either be true or false.
- The model can be defined in terms of predicates and manipulated using the rules of the predicate calculus.
- The predicate calculus forms the basis of all formal specification languages.

Predicate Transformation

- The Predicate transformation provides a basis for verifying model correctness by
  - formally defining the semantics of the model with a mapping which transforms model output states to all possible model input states.
- This representation provides the basis for proving model correctness.

Proof of Correctness

- Formal proof of correctness corresponds to
  - expressing the model in a precise notation and then
  - mathematically proving that the executed model terminates and
  - it satisfies the requirements specification with sufficient accuracy.
- Attaining proof of correctness in a realistic sense is not possible under the current state of the art.
- However, the advantage of realizing proof of correctness is so great that when the capability is realized, it will revolutionize the model VV&T.