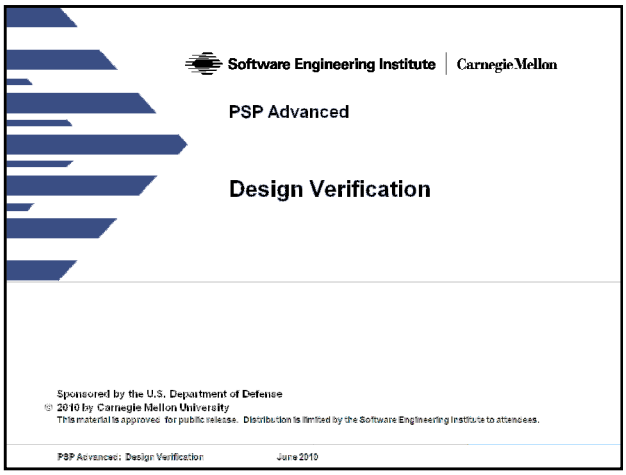


Lecture Topics

This lecture provides a survey of several design verification topics.

- Reasons for design verification
- Design verification methods
  - Design standards
  - Execution-table verification
  - Trace-table verification
  - Correctness verification
  - UML verification

Need for Design Verification -1

It is hard to eliminate many design defects just by making process changes.

Checklists and a structured review process can improve code-review yield.

Design reviews with checklists are also helpful. but not sufficient.

A checklist item, *"Is module logic correct?"* cannot be confirmed by scanning the specification templates.

Need for Design Verification -2

To improve design-review yield, you must use disciplined verification methods.

An orderly approach to design verification is essential because

- many common design defects are caused by overlooked conditions
- situations that seem unlikely become more likely with complex systems and high-powered computers
- conditions that were initially impossible may be likely after a program is modified

Benefits of Design Verification

By following a structured design verification procedure, you are more likely to

- see overlooked conditions
- identify rarely-exposed risks
- recognize possible future exposures

By recording information during each design review, you can improve the effectiveness of later design inspections.

Using Design Verification

Design verification methods should be used during

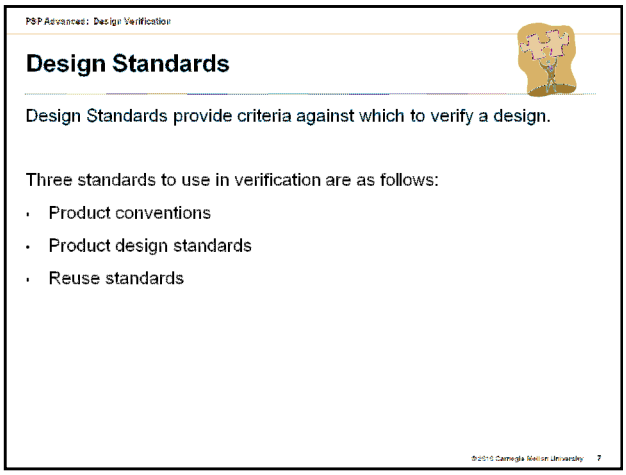
- design
- design reviews
- design inspections

Verifying designs with source-code is time-consuming and error-prone.

Use design verification methods to focus on the defect types that cause you the most trouble in test.

Select verification methods that are compatible with design techniques used.

Use your data to decide which verification methods are most effective for you.



A standard is a norm or basis for comparison that is established by some authority or by common consent. Because the purpose of verification is to determine whether the design is correct, you must first define what constitutes correctness.

Familiarize yourself with the appropriate standards before starting the design work and again when you do the verification.

Product conventions are essential if multiple developers are to produce coherent and usable systems. These conventions should at least address the user interfaces, external naming, system error handling, installation procedures, and help facilities.

Product design standards range from simple conventions to complete system architectures. These standards typically cover calling and naming conventions, header standards, test standards, documentation formats, etc. Although many product design standards are arbitrary, a standard is needed in order to produce consistent designs that are reviewable by both yourself and your team members.

In order for reusable soft to be widely used, standard software parts must be precisely specified, of the highest quality, and adequately supported. This is accomplished through the establishment of reuse standards.

Execution Tables -1

An execution table is an orderly way to trace program execution.

- It is a manual check of program flow.
- It starts with initial conditions.
- A set of variable values is selected.
- Each execution step is examined.
- Every change in variable values is entered.
- Program behavior is checked against the specification.

Execution Tables -2

The advantages of execution tables are that they

- are simple
- give reliable results

The disadvantages of execution tables are that they

- check only one case at a time
- are time-consuming
- are subject to human error

Execution Table Procedure

To use an execution table

1. identify the key program variables and enter them at the top of the table
2. enter the principal program steps
3. determine and enter the initial conditions
4. trace the variable values through each program step
5. use additional execution-table copies for each cyclic loop

Trace Tables

The **trace table** is a generalized form of the execution table.

- With an execution table, you manually work through every test scenario.
- With a trace table, you first consider all the possible logical cases and select the test scenario for each one.

Trace tables examine general program behavior, rather than verifying individual cases.

Trace tables are much more efficient, and provides a more general proof, than execution tables.

Case Checking

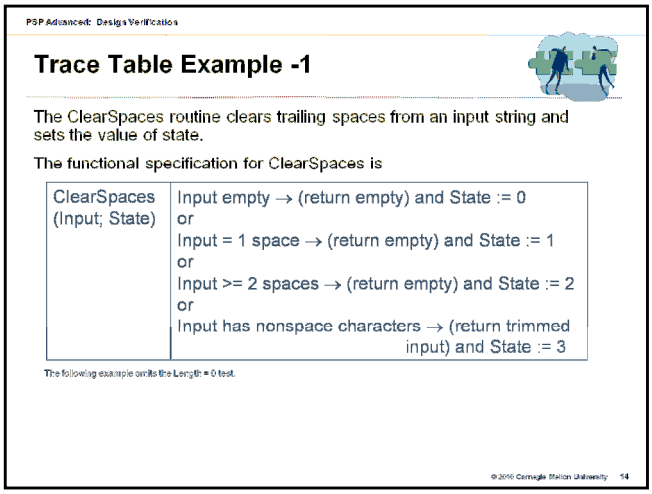
Case checking involves the following five steps.

1. Examine the program to determine its behavior categories.
2. For these categories, identify the cases to check.
3. Check the cases to ensure that they cover all possible situations.
4. Examine program behavior for each case.
5. Once every case has been verified, program behavior has been verified.

Trace Table Procedure

The trace table procedure is the same as that for the execution table, plus three steps.

1. Identify the key program variables and enter them at the top of the table.
2. Enter the principal program steps.
3. ***Identify symbolic execution possibilities.***
4. ***Define all of the possible cases and select those to check.***
5. Determine and enter the initial conditions.
6. Trace the variable values through each program step.
7. Use additional trace-table copies for each cyclic loop.
8. ***For multi-cycle loops, group intermediate steps if their results are obvious.***

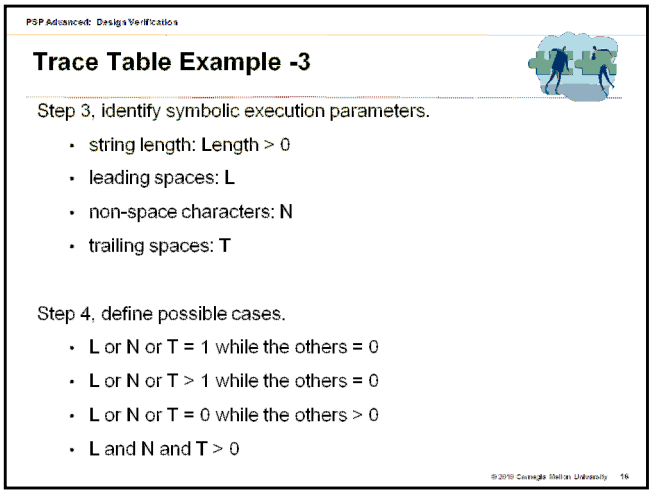


Just an example of what one looks like. No need to walk through in detail. The student will get practice during the design verification exercise, as well as with each programming assignment.

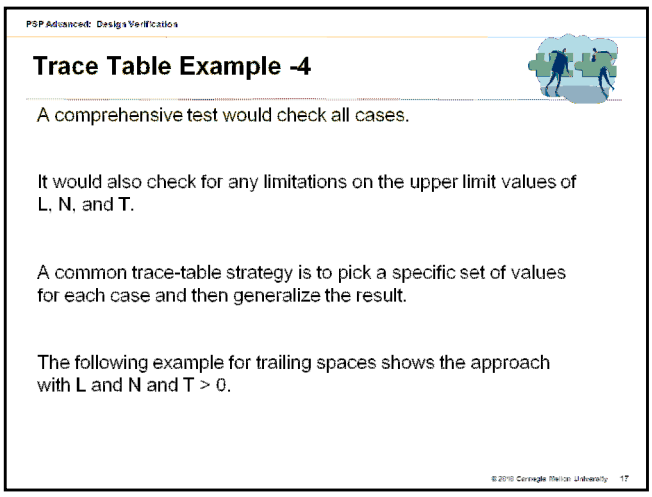
Trace Table Example -2

Steps 1 & 2: identify key variables and enter principal program steps.

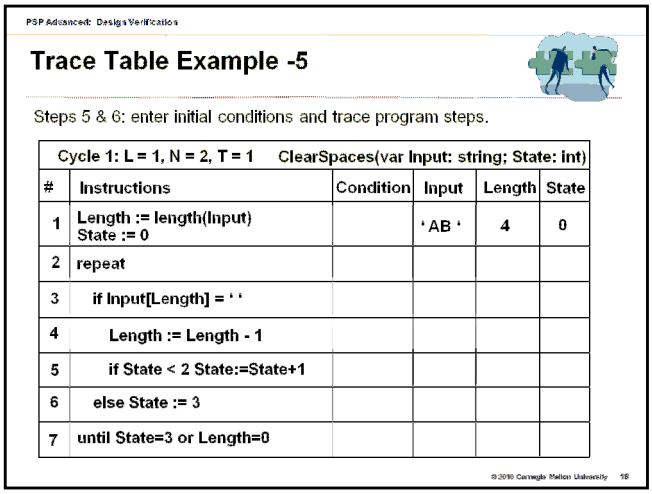
	ClearSpaces(var Input: string; State: int)				
#	Instructions	Condition	Input	Length	State
1	Length := length(Input) State := 0				
2	repeat				
3	if Input[Length] = ' '				
4	Length := Length - 1				
5	if State < 2 State:=State+1				
6	else State := 3				
7	until State=3 or Length=0				



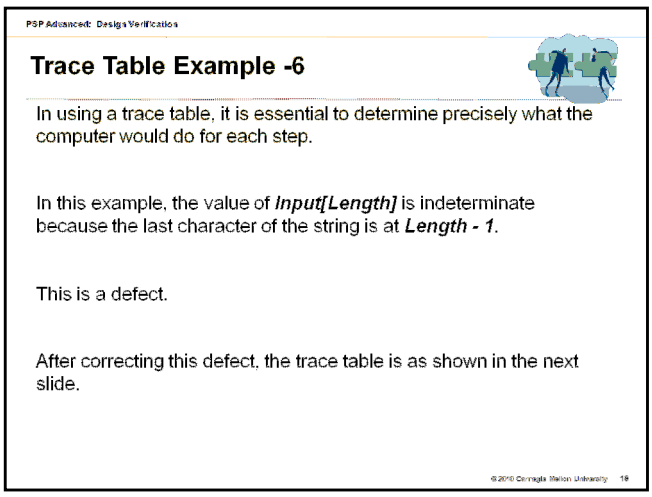
Just an example of what one looks like. No need to walk through in detail. The student will get practice during the design verification exercise, as well as with each programming assignment.



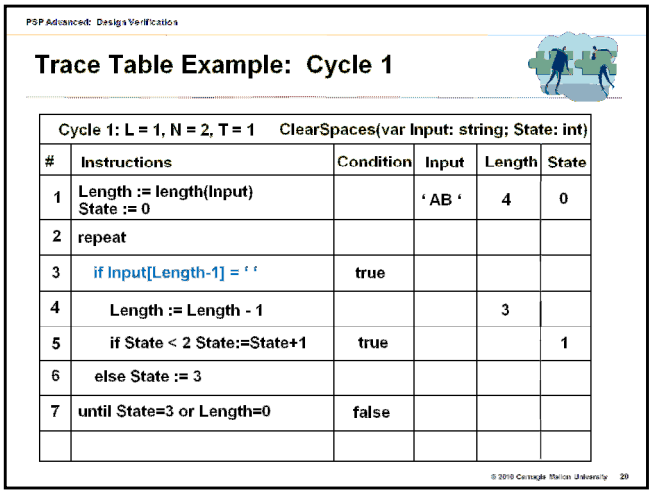
Just an example of what one looks like. No need to walk through in detail. The student will get practice during the design verification exercise, as well as with each programming assignment.



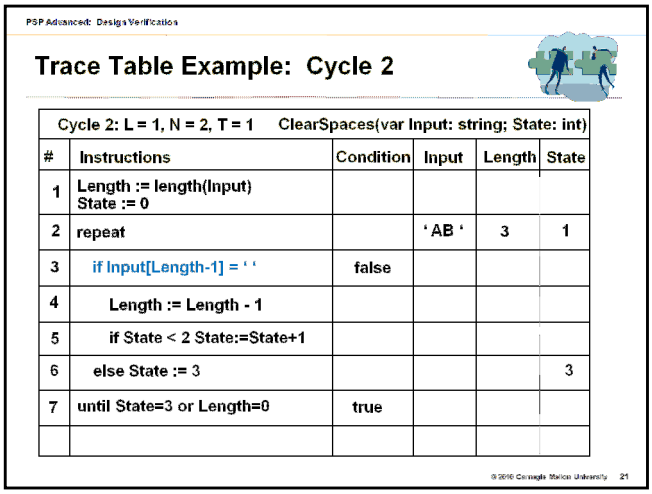
Just an example of what one looks like. No need to walk through in detail. The student will get practice during the design verification exercise, as well as with each programming assignment.



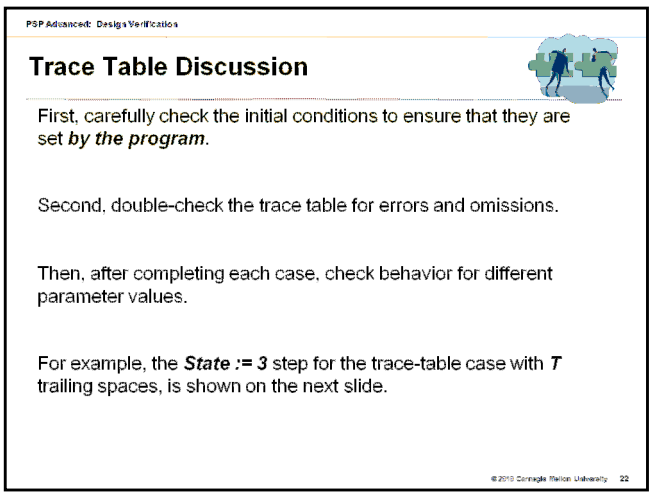
Just an example of what one looks like. No need to walk through in detail. The student will get practice during the design verification exercise, as well as with each programming assignment.



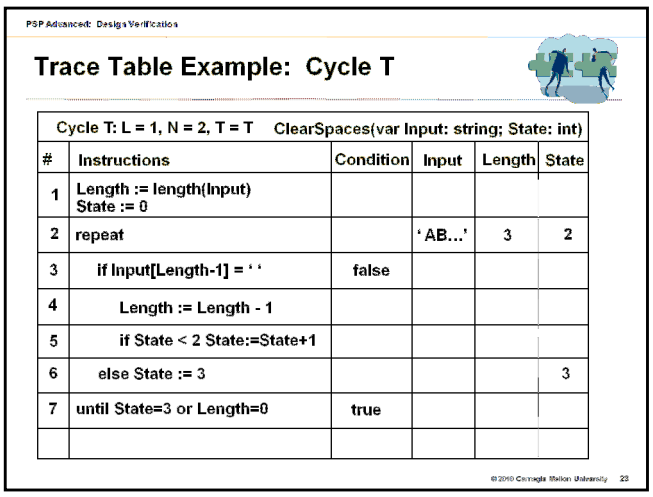
Just an example of what one looks like. No need to walk through in detail. The student will get practice during the design verification exercise, as well as with each programming assignment.



Just an example of what one looks like. No need to walk through in detail. The student will get practice during the design verification exercise, as well as with each programming assignment.



The use of trace tables may seem like a laborious way to verify a design, particularly when a computer is available to do the calculations. When using a debugger to perform a trace of your code, you will find that for a thorough test, you must do the equivalent of all the trace table analyses without the aid of the trace table form and methods. Working through the trace tables during design review and then using the results of the trace table to verify you coded the design during unit test is an effective use of the debugger, as you will have already worked out the solution in which to compare the debugger's results.



Just an example of what one looks like. No need to walk through in detail. The student will get practice during the design verification exercise, as well as with each programming assignment.

Verifying Program Correctness

Correctness verification methods treat programs as mathematical theorems.

These methods are particularly useful during design and design review.

Because program verification often involves sophisticated reasoning, it is important to check your work carefully.

Correctness Verification

To verify a program

- identify all of the program cases
- consider each verification question while reviewing each non-trivial construct
- where the answer is not obvious, use a trace table to evaluate the conditions that the program must satisfy

While this verification approach works with most design constructs, only "***while loops***" are described in this lecture.

Proofs for repeat-until and for-loops are described in the textbook (pages 271 - 277).

WhileLoop Verification -1

The ***WhileLoop*** is assumed to have the following form.

```
while WhileTest
   begin
     LoopPart
   end
```


WhileLoop Verification -2

A whileloop is correct if all of the following questions can be answered with "yes."

- Question 1: Is loop termination guaranteed for any argument of ***WhileTest***?
- Question 2: When ***WhileTest*** is true, does ***WhileLoop*** = ***LoopPart*** followed by ***WhileLoop***?
- Question 3: When ***WhileTest*** is false, does ***WhileLoop*** = identity?

WhileLoop Example -1

Assuming n > 0, calculate n!

```
f = 1
i = 1
while i != n     {i not equal to n}
begin
     i = i + 1
     f = f * i
end
```


WhileLoop Example -2

Question 1: Is loop termination guaranteed for any argument of *WhileTest* ?

Yes, the loop terminates when $i = n$. Since i steps inside the loop, it will terminate as long as $n > 0$.

Question 2: When *WhileTest* is true, is *WhileLoop* = *LoopPart* followed by *WhileLoop* ?

Yes, the *LoopPart* increments i and sets f equal to $f * i$, so f is still equal to $i!$

Note: The initial conditions ensure that $f = i!$ initially.

WhileLoop Example -3

Question 3: When *WhileTest* is false, does *WhileLoop* = identity?

Yes, when *WhileTest* is false, the loop terminates without further changes to any variables, so $i = n$ and $f = i!$

Hence, we can deduce that $f = n!$ as required.

Comments: Program Verification

If done correctly, these verification methods can guarantee program correctness.

Mastery of these verification methods requires skill and practice.

The loop termination test is critical because endless loops are hard to identify with other methods.

While you can often answer the verification questions by inspection, when in doubt, check the results with a trace table.

UML and Verification -1

Graphical designs can be difficult to verify and often require special care.

UML diagrams can and should be checked for consistency.

The names of all classes, operations, and attributes should also be defined and used consistently.

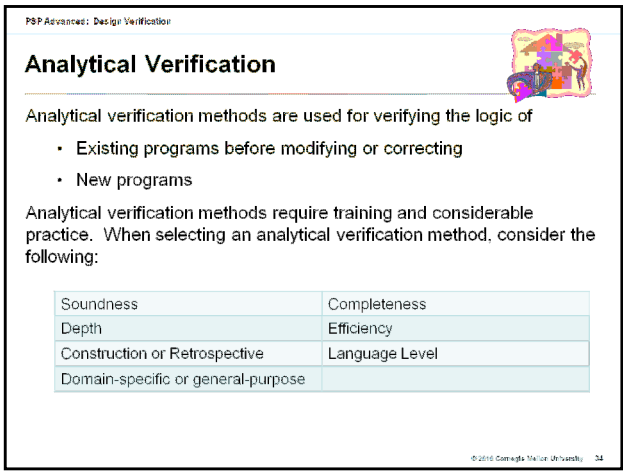
UML state charts can be verified as shown before, but the state and transition conditions must be explicit.

UML and Verification -2

Execution in UML sequence diagrams can be traced if the diagrams are sufficiently detailed.

Similarly, every message sent between objects must be supported by a traversable link defined in a class diagram.

Some UML tools can simulate execution of UML models, but these tools should be used only after other review techniques.



Soundness: Does the verification provide misleading results? That is, can it indicate that the program is free of a defect category when it is not?

Completeness: Does the verification method find 100% of the defects of a given category or only as many as it can? An incomplete method can be "honest" and indicate areas of doubt or incompleteness or it can be silent. The "false-alarm" rate is also critical, as excessive false alarms can waste development effort.

Depth: A deep analysis covers such complex topics as buffer overflow, while a shallow analysis might address coding standards and formats. Deep analyses are rarely complete.

Efficiency: Efficiency concerns the time required for the analysis and the degree of human assistance required.

Constructive or retrospective: A constructive analysis can be run on partially completed products, whereas a retrospective analysis can be used only with finished products.

Language level: The language level concerns the types of products that can be verified. Examples would be the specification level, the design level, or the source-code level.

Domain-specific or general-purpose: Does the method work only for selected application domains or is it of more general use?

Messages to Remember

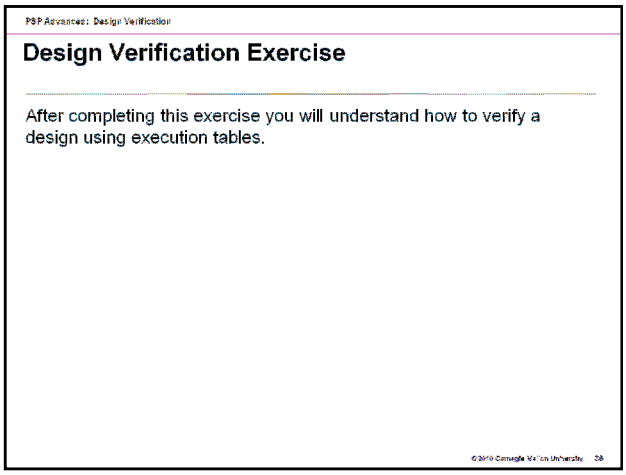
There are many verification methods, and we have only covered a few of them. So try any method that looks promising and use your PSP data to evaluate the results.

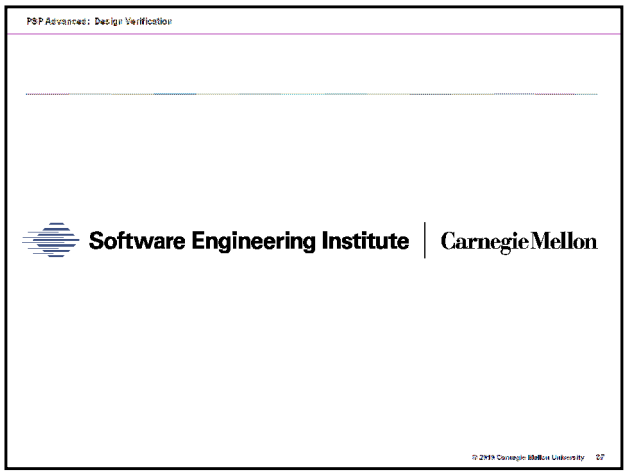
In order to become fluent with any verification method, you must use it several times on a variety of programs.

Once reasonable proficient, select the most effective ones for finding the defects that you most commonly find in testing.

You will significantly improve your design-review yield by using disciplined design review methods.

With complex programs, the time spent verifying designs will be more than repaid by the testing time saved.


PSP Advanced: Design Verification
Design Verification Exercise
After completing this exercise you will understand how to verify a design using execution tables.
©2010 Carnegie Mellon University 38


PSP Advanced: Design Verification
Software Engineering Institute
Carnegie Mellon
© 2010 Carnegie Mellon University 37