12

Design Verification

There are many ways to verify a program's design. Some of these methods, like testing, can be automated, while others cannot. For large and complex programs, even the most time-consuming of the labor-intensive methods is much faster and more effective than testing. This chapter explains why this is true and describes some of the more effective methods for verifying the correctness of your programs.

Although it may seem pointless to verify the design of each small PSP exercise, you should use these methods as often as you can. This will give you the experience needed to use the methods on larger and more complex programs, and it will help you to build the skills required to properly use the methods when you need them.

The following five verification methods described in this chapter are relatively straightforward and can be mastered fairly quickly:

1. **Design standards:** Standards provide the basis for verifying that a design properly performs its intended functions.
2. **Execution-table verification:** The execution table provides a simple and orderly but labor-intensive way to check a program's logic.
3. **Trace-table verification:** A trace table is a general way to verify a program's logic.
4. **State-machine verification:** State-machine verification provides analytical methods for verifying that a given state machine is proper and thus correct.

31422 12 pp. 253-286 r3ah.ps 2/9/05 5:21 PM Page 254

5. **Analytical verification:** Analytical verification provides a mathematical way to demonstrate the validity of program logic.

This chapter works through several examples that show how to use some of these verification methods with the PSP exercises. In order to become fluent with the methods, you must use them several times on a variety of programs. Then you should be reasonably proficient and able to use them to verify larger-scale designs.

There are many verification methods, and this book covers only a few of the most effective ones. Some of the most promising verification methods that are not covered are based on mathematical proofs. Although many of these methods are very powerful, useful descriptions would take more space than is practical for this book. In addition to the many verification methods already available, new ones will almost certainly be developed, so try any methods that look promising and use your PSP data to evaluate the results. Then make a practice of using those that you find most effective.

12.1 Why Verify Programs?

The principal challenge with design verification is convincing yourself to do it. We all learned to program by writing small programs. After a few tests and fixing a few defects, we could usually get our programs to run. Although the finished programs occasionally still had defects, these were usually easy to fix. These early experiences often give the impression that a few defects are acceptable as long as the program can be made to work. Unfortunately, this belief is dangerous. One or two defects in every 100-LOC program means that there are 10 to 20 defects in a 1,000-LOC program, and 10,000 to 20,000 defects in a 1,000,000-LOC program. This is no longer just a couple of bugs. Some of these defects could cause serious problems.

For example, the Osprey aircraft was designed to transport U.S. marines under battlefield conditions. It flies like a high-speed aircraft and lands like a helicopter. Although the Osprey is a marvelous engineering achievement, it is also very complex and contains a lot of software. And, like most software, the Osprey programs have latent defects. On December 11, 2000, as one of these aircraft was carrying four marines, the hydraulic system failed just as the pilot was landing. Even though the Osprey had a built-in backup system to handle such emergencies, a software defect in the flight-control computer caused “significant pitch and thrust changes in both prop rotors.” This led to a stall and a crash (Gage and McCormick 2004). All aboard were killed. Although development management told me that the Osprey software was very high quality, the quality was obviously not high enough.

To most people in the software business, improving quality means running more tests, but the Osprey software had been extensively tested. Although its quality, by today’s standards, was almost certainly high, this experience demonstrates that today’s generally accepted software quality standards are not good enough for life-critical systems. The problem is not that testing is wrong but that it is not enough. To find the thousands of latent defects in large software systems, we must thoroughly verify their designs, carefully review and inspect the code, and then run exhaustive tests.

Various authors have described the problems of using a test-based process to produce highly dependable programs. Littlewood cites a case in which it took over 74,000 hours of testing to determine that a software system had a mean time to failure (MTTF) of over 1,000 hours (Littlewood and Strigini 1993). Similarly, Butler shows that more than 10^5 years of testing would be needed to ensure that a software system could achieve the reliability levels now commonly specified by transportation and other life-critical systems (Butler and Finelli 1993).

To see why extensive testing alone cannot ensure software quality, consider what a comprehensive test would have to cover. Because untested software systems typically have thousands of defects, one obvious testing strategy would be to test every possible path through the software. Covering all possible paths through a program would be like threading all of the possible paths through a maze. Suppose, for example, that the program were like the maze shown in Figure 12.1, where every intersection was a branch instruction. As shown in Table 12.1, testing every possible path from point A to point B can require a lot of tests. For this simple 4-by-4 maze, there are 70 possible paths. As you can also see, the number of possible paths through larger mazes increases very rapidly. For a maze with

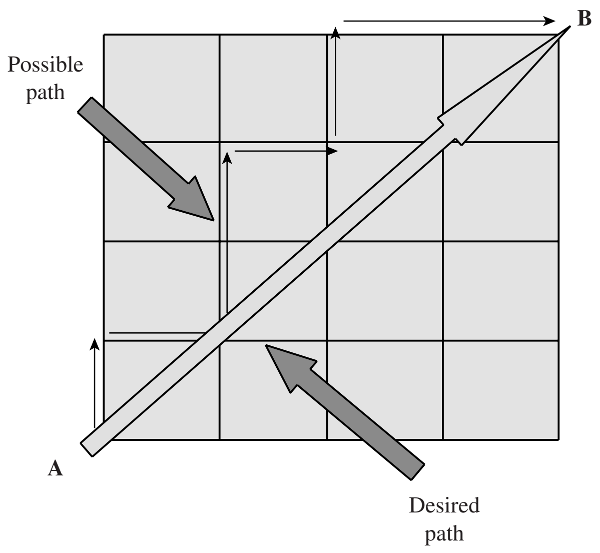

FIGURE 12.1 THE TESTING MAZE

only 400 branches, the number of possible paths is astronomical, and these are just for the numbers of paths. They do not consider loops, case statements, or multiple data values. As these numbers show, it is practically impossible to test all possible paths in any but the simplest programs. Furthermore, when you start with thousands of defects, just about any untested path could uncover a latent defect.

Since our job is to get complex systems to work, what can we do? No method will magically guarantee program quality but some verification methods are more effective than others. For example, testing is an ineffective and expensive way to find the hundreds to thousands of defects in the programs developers now typically deliver to test. However, testing is also the only practical way to find the few really complex problems that involve overall system behavior. Conversely, personal reviews and team inspections are very fast and effective for finding coding and simple design problems, but not as useful for complex performance or usability problems. When teams properly use all of these quality-management methods, quality improvements of 100 times are common.

For high-quality software work, no single method is sufficient by itself. We must use sound design methods and verify every program's design during the design review. We should write code only after we are confident that the design is sound and defect-free. We should then review, compile, inspect, and test the program to verify that it was correctly implemented. The verification methods described in this book can be cost-effective and efficient, but they are most effective when used in combination with team inspections and thorough testing. This chapter describes several ways to do design verification. It also discusses some issues to consider in deciding how to verify a program's correctness.

TABLE 12.1 NUMBER OF POSSIBLE PATHS THROUGH THE TESTING MAZE

Size of Maze	Number of Branches	Number of Possible Paths
1	1	2
2	4	6
3	9	20
4	16	70
5	25	252
6	36	924
7	49	3,432
8	64	12,870
9	81	48,620
10	100	184,756
20	400	$1.38 * 10^{11}$

12.2 Design Standards

Although design standards do not fit the common definition of a verification method, they do provide criteria against which to verify a design. A **standard** is a norm or a 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. Standards are an important part of this basis. Three standards to use in verification are as follows:

1. Product conventions
2. Product design standards
3. Reuse standards

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

Product Conventions

The value of product conventions is obvious when you use a system that did not have them. In developing IBM's early BPS/360 system, several programmers were assigned to work on the file-handling routines. Unfortunately, nothing had been standardized except the interfaces between their programs. One programmer specified that the file naming and addressing be in lowercase and another programmer called for uppercase. Because the two programs worked in tandem, the result was total user confusion. This problem forced us to withdraw and replace the entire BPS/360 operating system.

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. A quality system must have a clear and consistent user interface, regardless of who designed each part.

Product Design Standards

Product design standards range from simple conventions to complete system architectures. These standards typically cover calling and naming conventions, header standards, test standards, and documentation formats. Although many product design standards are arbitrary, it is a good idea to establish your own personal set. It will help you to produce consistent designs and will make your designs easier to review. This will also help in your personal work and will greatly facilitate your work on a software team.

Reuse Standards

Reuse is a powerful technique but it must be properly managed. When it is, it can save time and money and improve product quality. Reuse can be at the system-design level, the component-design level, or the modular-part level. To get maximum benefit, think about reuse during design and specifically address reuse in the design reviews.

The effective reuse of component parts requires the absolute integrity of those parts. The group in IBM's German laboratory that developed standard parts for use in IBM's operating systems distributed and supported its standard parts much like a commercial parts supplier. Their catalogue listed the parts and their specifications. The parts were essentially sealed and could not be modified, but they did have adjustable parameters. This group had product representatives in several major IBM laboratories who helped in identifying parts applications, assisted in their use, and identified new parts needs. When I last visited this development group, some of their parts had been included in IBM's products for up to 10 years. The engineers were proud to report that none of their users had ever reported a defect in any of their parts.

To be widely usable, standard software parts must be precisely specified, of the highest quality, and adequately supported. Their acceptance will then depend on their quality, their price, the degree to which they meet the users' needs, the effectiveness of their marketing, and the adequacy of their technical support. Modern development environments generally come with large libraries of standard functions; and by learning to use these libraries, you can greatly improve your personal productivity, as well as the quality of your products. You should also develop a library of reusable parts tailored to your particular application domain. Familiarize yourself with the standard parts that are available, use them whenever you can, and make a regular practice of contributing new parts to your team's or organization's application-specific reuse library.

Unless you are confident that the library functions and other programs you reuse are defect-free, you must verify their correctness when you verify the correctness of the new functions you develop. Reuse is important for the PSP because if the reused parts are of high quality, their use will substantially improve your productivity. One objective of the PSP is to develop the personal discipline required to produce reuse-quality software.

12.3 Execution-Table Verification

There are many ways to verify a program's logic before it is implemented. One of the most general methods is the **execution table**. Execution tables provide an orderly way to check a program's logical flow. Although the method can be time-consuming, particularly for large and complex programs, it is reliable and simple and far faster than testing as a way to identify complex design problems. An execution table can produce a fairly general verification, but it usually requires a fairly large number of cases. To minimize the time required while maximizing effectiveness, follow an orderly verification procedure and carefully check every step. Use the following procedure:

1. Review the program's pseudocode and identify all of the loops and complex routines to verify with an execution table. If the program has already been implemented, review the source code instead.
2. Decide on the order for doing the analysis. If you believe that the higher-level design is solid, start with the lowest-level routines and work up to the higher-level ones. Otherwise, do an initial check of the highest-level structure, assuming that the lower-level routines are correct.
3. To analyze the first routine, construct a blank execution table like the one shown in Table 12.2 for the LogIn routine (from Table 11.9, p. 240). Make a column in the table for every variable that changes during program execution, and a column for true (T) and false (F) for all of the true/false test results.
4. Perform an execution-table analysis for each of the selected scenarios.

The following example walks through an execution-table analysis for the LogIn routine given in Table 12.2. The steps in the execution-table analysis are as follows:

1. In Table 12.2, the only logical structure to check is the repeat-until loop from lines 3 to 12. You can do this with multiple copies of one execution table.
2. Produce the execution table and make a sufficient number of blank copies to handle all of the verification scenarios.
3. Select a scenario to verify. For this example, we check the following three-step sequence: !Valid ID, Valid ID & !Valid PW, and Valid ID & Valid PW. To check additional cases, you would use additional copies of the execution table.
4. Label a copy of the execution table with the scenario step being tested and "Cycle 1," check the program design carefully to determine the initial conditions, and enter them at the top of the cycle 1 table. In doing so, make sure that these initial conditions are actually set by the program. Incorrectly assumed initial conditions are a common source of error.
5. Work through the program one step at a time, entering all variable changes exactly as the program would make them. As you do, consider the logical choices and satisfy yourself that the program behaves correctly in every case. When you encounter a repetitive loop structure, complete the first cycle

through the loop on the cycle 1 table and use a second copy of the table for cycle 2, and so on.

6. At the end, verify that the result is correct.

As simple as this program is, constructing and checking the execution table revealed three problems with the logic in Table 11.9 in Chapter 11 (see p. 240). First, step 6 was to both get the password and check it. This step had to be split into the current step 6 and a new step 8, so that the password would be checked only if there was a valid ID. Second, in step 9, the *n* counter should be stepped if either !Valid ID or !Valid PW. Finally, in step 12, if the loop is repeated, ID should first be set to !Valid; otherwise, the third cycle would start with Valid ID. Although these are all obvious problems that you would likely correct during coding, they are the kinds of problems that lead to logical defects that are hard to find in testing. They are precisely the kinds of problems that design verification is intended to find and fix. The three cycles of this corrected execution table are shown in Tables 12.2 through 12.4.

TABLE 12.2 LOGIN EXECUTION TABLE

Cycle 1: Invalid ID: Cycle 1		Function: LogIn				
#	Instruction	Test	ID	PW	Fail	n
1	Initialize: n := 0; ID: = !Valid; PW := !Valid; Fail := false.	F	!Valid	!Valid	F	0
2	Get user ID.					
3	Repeat the main loop until a Valid ID and password or Fail.					
4	Check ID for validity. {CheckID state}		!Valid			
5	If no ID response in MaxTime, set Fail := true.	F				
6	Get password					
7	If no password response in MaxTime, set Fail := true.	F				
8	If ID Valid, check PW for validity. {CheckPW state}	F				
9	If PW !Valid or ID !Valid, step the n counter.	T				1
10	If n exceeds nMax, set Fail := true.	F				
11	Until ID and PW valid or Fail = true.	F				
12	Otherwise, repeat the main loop; ID := !Valid	T	!Valid			
13	If Fail = true, cut off user, else, log in the user. {End state}					

TABLE 12.3 LOGIN EXECUTION TABLE

Cycle 2: Valid ID and invalid password		Function: LogIn				
#	Instruction	Test	ID	PW	Fail	n
1	Initialize: n := 0; ID: = !Valid; PW := !Valid; Fail := false.					
2	Get user ID.					
3	Repeat the main loop until a valid ID and password or Fail.		!Valid	!Valid	F	1
4	Check ID for validity. {CheckID state}		Valid			
5	If no ID response in MaxTime, set Fail := true.	F				
6	Get password					
7	If no password response in MaxTime, set Fail := true.	F				
8	If ID Valid, check PW for validity. {CheckPW state}	T		!Valid		
9	If PW !Valid or PW !Valid, step the n counter.	T				2
10	If n exceeds nMax, set Fail := true.	F				
11	Until ID and PW valid or Fail = true.	F				
12	Otherwise, repeat the main loop; ID := !Valid	T	!Valid			
13	If Fail = true, cut off user, else, log in the user. {End state}					

TABLE 12.4 LOGIN EXECUTION TABLE

Cycle 3: Valid ID and valid password		Function: LogIn				
#	Instruction	Test	ID	PW	Fail	n
1	Initialize: n := 0; ID: = !Valid; PW := !Valid; Fail := false.					
2	Get user ID.					
3	Repeat the main loop until a valid ID and password or Fail.		!Valid	!Valid	F	2
4	Check ID for validity. {CheckID state}		Valid			
5	If no ID response in MaxTime, set Fail := true.	F				
6	Get password					
7	If no password response in MaxTime, set Fail := true.	F				
8	If ID Valid, check PW for validity. {CheckPW state}	T		Valid		
9	If PW !Valid or ID !Valid, step the n counter.	F				
10	If n exceeds nMax, set Fail := true.	F				
11	Until ID and PW valid or Fail = true.	T				
12	Otherwise, repeat the main loop; ID := !Valid	F				
13	If Fail = true, cut off user, else, log in the user. {End state}	F				

12.4 Trace-Table Verification

The **trace table** is a generalized form of execution table. With an execution table, you are essentially acting like a computer and manually working through every step of a test scenario. With a trace table, you first consider all of the possible logical cases and select a test scenario for each one. For example, if you know that a function that works properly at $n = 0$ and at $n = 1{,}000$ would also work properly for all intermediate values, you would only need to check the $n = 0$ and $n = 1{,}000$ cases. You should also check one intermediate value and any possible cases with $n < 0$ and $n > 1{,}000$.

Where practical, do a trace-table analysis for each possible logical case. The proper selection of cases can significantly reduce the number of required scenarios. Although trace tables look just like execution tables, the method is much more efficient and provides a more general proof. The trace-table procedure is as follows:

1. Review the program's pseudocode and identify all of the loops and complex routines to verify with a trace table. If the program has already been implemented, review the source code instead.
2. Decide on the order for doing the trace-table analysis. If you believe that the higher-level design is solid, start with the lowest-level routines and work up to the higher-level ones. Otherwise, do an initial check of the highest-level structure, assuming that the lower-level routines are correct.
3. Identify all of the logical cases to check with a trace table.
4. To analyze the first routine, construct a blank trace table like the one shown in Table 12.2 for the LogIn routine. Make a column in the table for every variable that changes during program execution, and a column for true (T) and false (F) for all of the true/false test results.
5. Perform a trace-table analysis for all of the nontrivial cases.

A Trace-Table Example

As one example of the trace-table method, consider the following program to clear the leading spaces from a string. This example uses a for loop to remove the leading spaces from a string of length Length:

```
for N = 1 to Length do
    if (Input[N-1] <> ‘ ’ or First)
            Output := Output + Input[N - 1]
            First := true
end
```

Starting with the program preconditions, examine all of the cases that are established by the program inputs and prior logic. For this short program fragment, the preconditions are as follows:

1. First is false.
2. Length may be any positive integer greater than 0.
3. The input string includes q nonspace characters, where q is an integer from 0 to Length.
4. The input string starts with p spaces, where p may be any positive integer from 0 to Length.
5. There may be spaces embedded in the input string after the first nonspace character.
6. The output string is initially empty.

In determining the possible cases, consider the number of leading spaces p and the number of message characters q. The numbers p and q could be 0, 1, or some larger number m or n, but p + q = Length. Therefore, the possible cases are as follows:

- ☐ A single message character with no leading spaces: p = 0, q = 1, Length = 1
- ☐ Several message characters and no leading spaces: p = 0, q = n, Length = n
- ☐ No message characters and one leading space: p = 1, q = 0, Length = 1
- ☐ A single message character with one leading space: p = 1, q = 1, Length = 2
- ☐ Several message characters with one leading space: p = 1, q = n, Length = n + 1
- ☐ No message characters and several leading spaces: p = m, q = 0, Length = m
- ☐ A single message character with several leading spaces: p = m, q = 1, Length = m + 1
- ☐ Several message characters with several leading spaces: p = m, q = n, Length = m + n

These eight conditions include all of the cases except those with zero message and space characters. If you can verify these eight cases and show that they are true for all possible values of p and q, you have verified all of the nonzero character cases. Note that for a complete test, you would also have to include the zero character case and cases for various numbers of embedded spaces.

These cases can be checked with a trace table, as shown in Table 12.5. To check all eight cases, you would need eight trace tables. Note, however, that you can abbreviate cycles where they merely repeat the same actions. In Table 12.5, for example, the . . . (ellipsis) on line 7 indicates multiple additional identical cycles that merely increase the value of N by 1 and add one additional character to the output. By checking all eight cases in this way, you can demonstrate that the

TABLE 12.5 CLEARLEADINGSPACES TRACE TABLE (CASE p = 1, q = n)

Cycles 1 through N: p = 1, n characters		Function: Clear Leading Spaces					
#	Instruction	T/F	Input	Output	Length	First	N
1	for N = 1 to Length do		‘ a..n’	‘’	n+1	F	1
2	if (input[N-1] <> ‘ ‘ or First	F					
3	Output := Output + Input[N-1] First := true						
4	for N = 1 to Length do			‘’			2
5	if (input[N-1] <> ‘ ‘ or First	T					
6	Output := Output + Input[N-1] First: := true			‘a’		T	
7	…						
8	for N = 1 to Length do			‘a..n-1’			n+1
9	if (input[N-1] <> ‘ ‘ or First	T					
10	Output := Output + Input[N-1] First := true			‘a...n’			
11	end			‘a…n’			

design is correct. Although this may not seem very important for a simple loop like ClearLeadingSpaces, it is a quick way to comprehensively check simple cases and a very effective way to check complex ones.

Trace-Table Practices

In using the trace-table method, the fastest and most effective technique is to make a blank trace-table form like the one shown in Table 12.5. Then make enough copies for all of the cases and cycles that you plan to test. In completing these tables, work through the program one instruction at a time. For each variable change, think through precisely what the program would do. When you make these checks on every branch and every path and ensure that every part of every condition is correct, you will perform very high-yield design reviews and will usually produce programs that have no design defects.

During a trace-table analysis, consider program robustness. For example, examine all of the possible limits and confirm that the design will handle them. For input ID or password strings in particular, consider what would happen with 0, 10, 100, or even 100 million or more characters. Check for any limits on the routines that calculate string length and the system's handling of buffer space. Alternatively, you could modify the program to enforce limits on string length. Also consider any possible special characters and what would happen if they were included in an input string. Then take appropriate precautionary steps. Finally, clearly state any input limitations in the program's specifications and comments.

Using paper forms may seem like a laborious way to verify a design, particularly when a powerful computer is available to do the calculations. However, no computer can think through all of the possible cases as quickly as you can. To prove this, work out all of the possible cases for a simple program and then try using a debugger to test them all. If you measure the time and the work required, 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 forms and methods. In general, however, such intuitive analyses are both time-consuming and error-prone. With a blank trace table, it takes much less time to work through the program steps and fill in the blanks. When you are done, you will have a verified program and a detailed script for checking the verification work with a debugger. Then, for any but the simplest logic, check the more complex trace-table cases with the debugger to verify your trace-table analysis.

Complex programs often result in complex trace tables, but this is when a trace-table analysis is most important. It is almost impossible to verify a complex logical structure with test cases. Often, however, these complex structures can be greatly simplified by concentrating only on the decision points and treating all of the other instruction sequences as single pseudocode lines. If the logic is still too complex to analyze with a trace table, it is also too complex to adequately test and should be redesigned.

12.5 Verifying State Machines

When a program includes a state machine, it is important to prove that it is properly designed and used. To determine whether a program is a state machine, check its external behavior. If it can behave differently with identical inputs, it is a state machine. An example would be the behavior of the character reader for a LOC counter. While in the regular portion of the program, the character reader would pass all characters to the counter. When it detected the start of a comment, however, it would ignore all subsequent characters until it came to the end of the comment. To do this, the character reader would need two states: CommentText and ProgramText. With the identical character as input, in one case it would produce no output and in the other it would pass the input character to the output.

Completeness and Orthogonality

Although there is no general way to prove that a state machine is properly designed, there are some common design problems. Examples are overlooking possible states or having conflicting or overlapping state transition conditions. Such problems are extremely hard to find in testing but can be quickly found in a design review. You can also greatly reduce the likelihood of having state-machine design problems if you consider these issues during program design. A state machine that conforms to all generally accepted design criteria is called a **proper state machine**. To be proper, it must be possible for it to reach a return state, if there is one, from every other state; and all of the transitions from each state must be complete and orthogonal. If there is no return state, there must be a way to transition from every state to every other state, with the possible exception of the initial state.

Complete Functions

A complete function is one that includes all possible logical conditions. For example, if you were designing a poll to determine how people feel about an issue, you might consider the following four population characteristics: male or female, political party member or not, married or not, and registered voter or not. If you then produced three reports—poll A to tally the opinions of the male party members, poll B for the opinions of the registered voters, and poll C for the opinions of married people—would you have a complete set of opinions? If you didn't, who would you have left out?

This is not a complete function or set of poll reports. It leaves out female unmarried nonvoters and unmarried men who are not party members or voters. To see why, examine Table 12.6. This table is called a **Karnaugh map** and it shows all of the possible combinations of all the variables. The cells covered by each report are labeled A, B, and C, respectively, and the empty cells indicate combinations not covered by any report. A complete set of poll reports, like a complete function, would cover all of the possible combinations of the variables, with no empty cells.

TABLE 12.6 MAP OF POLL REPORTS

	Male				Female			
	Member		Nonmember		Member		Nonmember	
	Married	Single	Married	Single	Married	Single	Married	Single
Voter	ABC	AB	BC	B	BC	B	BC	B
Nonvoter	AC	A	C		BC		BC	

Orthogonal Functions

A set of functions is orthogonal if none of its members have any common conditions. In the polling example, this would mean that no groups were double-counted. From the Karnaugh map shown in Table 12.6, it is clear that the three poll reports are not orthogonal. The double- and triple-counted groups are married voters and male party members who are either married or voters. The only triple-counted group is male party members who are both married and voters.

Complete and Orthogonal Functions

An example of orthogonal reports would be D: male party members, and E: female voters, as shown in Table 12.7a. However, this is not a complete set of reports. A complete set would be polls of F: all women, G: male voters, and H: men who are not voters or are not party members, as shown in 12.7b. A complete and

TABLE 12.7 MAP OF POLL REPORTS

Table 12.7a D: male party members, E: female voters

	Male				Female			
	Member		Nonmember		Member		Nonmember	
	Married	Single	Married	Single	Married	Single	Married	Single
Voter	D	D			E	E	E	E
Nonvoter	D	D						

Table 12.7b F: women, G: male voters, H: male nonparty members or nonvoters

	Male				Female			
	Member		Nonmember		Member		Nonmember	
	Married	Single	Married	Single	Married	Single	Married	Single
Voter	G	G	GH	GH	F	F	F	F
Nonvoter	H	H	H	H	F	F	F	F

Table 12.7c I: male voters or singles, J: nonvoters who are women or married, K: women voters

	Male				Female			
	Member		Nonmember		Member		Nonmember	
	Married	Single	Married	Single	Married	Single	Married	Single
Voter	I	I	I	I	K	K	K	K
Nonvoter	J	I	J	I	J	J	J	J

31422 12 pp. 253-286 r3ah.ps 2/9/05 5:21 PM Page 268

orthogonal set of reports would cover all of the cells in the Karnaugh map with no duplication. For example, as shown in Table 12.7c, the following three reports would be complete and orthogonal. Report I is for men who are either voters or unmarried (or both), report J for the nonvoters who are either women or married (or both), and report K for the women who are voters.

When the state transitions of a state machine are complete and orthogonal, all of the possible next states have been specified and each is unique. This is important for software design because program loops involve state transitions. If such a state machine does not have a complete set of transitions, some possible conditions have not been defined. Similarly, a state machine with state transitions that are not orthogonal means that there is some set of conditions for which more than one next state is possible.

When either of these conditions is encountered, a program is likely to enter into an endless loop, hang up, or do something unexpected.

Verifying That a State Machine Is Proper

By taking the following steps, you can verify that a state machine is proper:

1. Draw the state diagram and check its structure to ensure that it has no hidden traps or loops.
2. Examine the design to ensure that the set of all transitions from each state is complete. That is, there is a defined transition from every state to some next state for every possible combination of conditions.
3. Verify that the set of all transitions from each state is orthogonal and that there is only one next state for every possible combination of state-machine input values and attributes.

Step 1 means that the state machine cannot get stuck in an endless loop and never reach a return state. This could happen if the transitions among a set of states can become circular. For example, if the transitions among a set of states only increased n and the exit from this set of states required that $n = 100$, an endless loop would occur whenever the state machine entered any of these states with $n > 100$.

State-Machine Verification Examples

The procedure for verifying the correctness of a state machine is illustrated with two examples. The first is for the LogIn function designed in Chapter 11 (see pp. 231–240) The second is for the Search state machine, also designed in Chapter 11 (see pp. 242–246), to find the value of x for which $F(x) = N$.

Verifying the LogIn State Machine

The State Specification Template for LogIn is shown in Table 11.7 see p. 238). The state-machine verification procedure is as follows:

1. From checking the state diagram in Figure 11.4 (see p. 237), it is clear that there are no hidden traps or loops. Because the only possible loop is between CheckID and CheckPW and because n := n+1 for every cycle, the n >= nMax exit condition means that there will eventually be an exit.
2. The transition conditions from every state can be examined for completeness and orthogonality by examining the state template in Table 11.7 (see p. 238). From Start, the only transition is to CheckID. From CheckID, the three possible transitions are CheckPW {ValidID}, CheckPW {!ValidID}, or End {Timeout}. Although the actions where !ValidID ∧ Timeout do not appear to be defined, they are logically included in the Timeout case. These transition cases are thus complete and orthogonal.
3. Finally, from CheckPW, the transitions are to CheckID {(!ValidID ∨ !ValidPW) ∧ n < nMax ∧ !Timeout)} or to End {(ValidID ∧ ValidPW) ∨ (n >= nMax ∨ Timeout)}. Because these cases are too complex to check by inspection, the decision table in Table 12.8 is used to examine all the possibilities.

TABLE 12.8 DECISION TABLE FOR CHECKPW TRANSITION CONDITIONS

n	< nMax				= nMax				> nMax			
Password	Valid		!Valid		Valid		!Valid		Valid		!Valid	
ID	Valid	!Valid	Valid	!Valid	Valid	!Valid	Valid	!Valid	Valid	!Valid	Valid	!Valid
Next State for Each Defined Transition Condition												
A		ID	ID	ID								
B	End				End				End			
C					End	End	End	End	End	End	End	End
Resulting Values of Fail for Each Defined Transition Condition												
A												
B	F				F				F			
C					T	T	T	T	T	T	T	T

In a decision table, the variable combinations are shown in a treelike structure with all of the possible final values shown at the "leaf" end at the bottom. The three transition conditions from CheckPW are as follows:

A: $(!\text{ValidID} \vee !\text{ValidPW}) \wedge n < \text{nMax} \wedge !\text{Timeout} \rightarrow$ CheckID / $n := n + 1$

B: $\text{ValidID} \wedge \text{ValidPW} \rightarrow$ End / Fail := false and LogIn user

C: $n >= \text{nMax} \vee \text{Timeout} \rightarrow$ End / Fail := true and cut off user

The Timeout cases are not shown because Timeout always transitions to End with Fail = 1. The cases discussed therefore assume !Timeout. For the next states, all cases are defined with no conflicts. For the values of Fail, the undefined cases are left at 0, the initialized value of Fail. In two cases, however, there is a possible conflict. This indicates that the LogIn state machine has a defect because there are two different ways to get to End from CheckPW and they each result in different values of Fail. The case of a valid password with ValidID and n > nMax is impossible, so it can be ignored, but the case of a valid password with ValidID and n = nMax is possible. It could occur from the following sequence: a transition from CheckPW to CheckID where n = nMax -1 followed by a valid ID and a transition back to CheckPW with n = nMax. Then, with a valid password, this conflict condition would arise.

This problem can be fixed by changing transition condition C as follows:

C: $(n >= \text{nMax} \vee \text{Timeout}) \wedge (!\text{ValidPW} \vee !\text{ValidID}) \rightarrow$ End / Fail := 1

Note that this problem did not show up in the execution-table check because neither the case of nMax = 2 nor the Timeout cases were checked. In addition, the pseudocode logic stepped the *n* counter to n + 1 before the test for n > nMax. This implementation choice resolved the conflict.

Verifying the Search State Machine

Next, we verify the correctness of a slightly more complex state machine, the Search function designed in Chapter 11. This state machine finds a value of *x* for which F(*x*) = M, where F(*x*) is a monotonically increasing or decreasing function of *x*. Here, the value of M is given by the user, and you are assumed to have a program that generates F(*x*), given *x*. The State Specification Template for Search is shown in Table 11.10 (see p. 245) and the state diagram is in Figure 11.6 (see p. 246). The state-machine verification procedure is as follows:

1. From checking the state diagram in Figure 11.6, it is clear that there are no hidden traps or loops. Although this may not be immediately obvious, the loop where Down transitions to Down with the high condition progressively

reduces the value of F(x). Ultimately, F(x) must drop below M. The same argument holds for the loop where Up transitions to Up. For the loop between Up and Down, the value of F(x) is oscillating from above to below M. Because Delta is reduced for every cycle, however, F(x) must eventually fall within the error range around M.

2. The transition conditions are also complete: Start always goes to DeltaSign, and the transition conditions from DeltaSign, Up, and Down are the same: whether F(x) < Target, F(x) > Target, or F(x) within the Target. Because these are all the possibilities, the transition conditions are complete.
3. Similarly, these transition conditions are orthogonal because they do not overlap.

This test verifies that the Search state machine is proper. If you are not sure of any orthogonality or completeness checks, use a decision table to check all of the possibilities.

12.6 Loop Verification

The PSP Design Review Checklist calls for verifying that all loops are properly initiated, incremented, and terminated (see Table 9.7 on p. 184). Because it is rarely possible to verify loop correctness by merely reading over the design, it is important to follow a defined verification method. The methods described here can be very helpful in verifying the logic of an existing program before modifying or correcting it or when verifying loop logic in a design review or inspection. Mills and Dyer described these methods many years ago, but they still provide useful guidance on determining loop correctness (Dyer 1992; Mills et al. 1987). The methods that are most generally useful are for the for loop, the while loop, and the repeat-until loop.

Even if you do not follow a rigorous loop-verification process every time you use these methods, they can still be helpful. For example, in a quick review, consider the proof questions when you review a loop's design. Just thinking about these questions will help you to identify problems that you might otherwise miss.

For-Loop Verification

Before doing the for-loop verification, identify the for-loop preconditions and verify that these preconditions are always met. The for-loop verification is as follows:

Does ForLoop = FirstPart + SecondPart + . . . + LastPart?

Here, ForLoop is assumed to have the following form:

```
ForLoop
    for n = First to Last
        begin
        nPart
    end
```

where nPart takes the values FirstPart, SecondPart, ThirdPart, and so on through LastPart. If you laid out the execution of this program through all the loops, it would be as follows:

```
    ForLoop
    begin
        FirstPart
        SecondPart
        ThirdPart
. . .    . . . . .
        LastPart
    end
```

Although this test may seem obvious, the actual verification is not trivial. To verify the equality, either construct a trace table or analyze the logic for each side of the equation and show that they always produce identical results. In this case, you would determine the results produced by ForLoop and then determine the results produced by FirstPart + SecondPart + … + LastPart. If the two results are identical, the for-loop logic is verified.

While-Loop Verification

Before doing the while-loop verification, identify the while-loop preconditions and verify that they are always met. This while loop is in the following form:

```
WhileLoop
    begin
        while WhileTest
            LoopPart
    end
```

In this loop, the WhileTest is performed first; if it fails, the LoopPart is not performed. The while-loop verification involves answering the following questions:

- ☐ Is loop termination guaranteed for any argument of WhileTest?
- ☐ When WhileTest is true, does WhileLoop = LoopPart followed by WhileLoop?
- ☐ When WhileTest is false, does WhileLoop = identity?

The first question is crucial: Is loop termination guaranteed for any argument of WhileTest? If loop termination is not guaranteed, this logic could possibly cause an endless loop, which is obviously to be avoided. Proving that this condition is either true or false can sometimes be difficult, but it can be demonstrated by identifying every possible case and either using a trace table or carefully examining the logic to prove that it is true. Although you will often be able to verify this condition by inspection, it is essential to consider every possible case.

The second question is somewhat trickier. To determine whether the WhileTest is true, you must find out if the function WhileLoop is identical to a function that consists of LoopPart followed by WhileLoop. If this is not true, then WhileLoop has a defect. To understand why this is true, consult the Mills or Dyer references (Mills et al. 1987; Dyer 1992). As before, you can answer the second question by constructing a trace table or studying the logic for the WhileLoop and the logic for LoopPart followed by WhileLoop and showing that they produce identical results for all cases.

The final question asks whether, when WhileTest is false, the logic continues through WhileLoop with no changes. If it does not, there is a defect. Again, this case can often be verified by inspection.

Example of While-Loop Verification

This next example illustrates the general loop-verification approach with the following pseudocode logic for a while loop:

```
NormalizeReal(Mant, Exp, NegEx)
1       while Mant > 10 do
2               Mant = Mant/10
3               if NegEx then Exp = Exp – 1
4               else Exp = Exp + 1
5       while Mant < 1 do
6               Mant = Mant*10
7               if NegEx then Exp = Exp + 1
8               else Exp = Exp – 1
```

This logic reduces a nonzero real number in the form *Mantissa**$10^{Exponent}$ to a standard form in which the Mantissa has a single nonzero digit before the decimal point. In this problem, Mant = abs(Mantissa) and Exp = abs(Exponent). Thus, Exp is a nonnegative integer and Mant is a positive real number. Two Boolean variables, NegMant and NegEx, indicate whether the Mantissa and/or the Exponent are negative. It is assumed that the values of these numbers are within the range of the number systems for the compiler and language being used.

To verify this logic, verify both while loops. Both loops are examined for question 1, but only the first loop is used to illustrate questions 2 and 3. First examine the while tests to determine whether they terminate. The cases to examine are for Mant >10, Mant < 1, and 1 <= Mant <= 10. The first case is Mant > 10. Here, when Mant > 10, the first loop is executed and successive divisions by 10 ultimately reduce Mant to less than 10. The while test is therefore not satisfied and the first while loop terminates. Now, Mant is less than 10 and greater than 1, so the second while loop will not be invoked and the loop will terminate.

The second case is Mant < 1. The first while loop is skipped and the second is invoked. Now Mant is progressively multiplied by 10 until its absolute value exceeds 1 and the second while test fails. This also terminates the loop.

Even though these two while loops terminate in these cases, this examination raises the question of what happens when Mant = 10 or Mant = 1. In the first case, both loops fail with the incorrect result of Mant = 10. The program thus has a defect, and line 1 should be changed as follows:

```
NormalizeReal(Mant, Exp, NegEx)
1       while Mant >= 10 do
2               Mant = Mant/10
3               if NegEx then Exp = Exp - 1
4               else Exp = Exp + 1
5       while Mant < 1 do
6               Mant = Mant*10
7               if NegEx then Exp = Exp + 1
8               else Exp = Exp - 1
```

When Mant = 1, both loops are skipped and the number is unchanged. Because it is already in the correct form, this is the correct result. Both while loops in this procedure now satisfy question 1 of the while-loop verification proof. The second question is, when WhileTest is true, does WhileLoop = LoopPart followed by WhileLoop?

To answer this question, you must examine all cases of WhileTest and WhileLoop. To invoke the first while loop, the cases must all have Mant >= 10.

Because Mant must be a nonnegative number, this then involves only two conditions as follows:

1. Mant >= 10, NegExp
2. Mant >= 10, ! NegExp

For the first case, the test consists of substituting the Mant and NegExp values into the program to determine whether WhileLoop equals LoopPart followed by WhileLoop. For the first case, the WhileLoop side of the equality is as follows:

```
1    while Mant >= 10 do
2         Mant = Mant/10
3         if NegEx then Exp = Exp - 1
4         else Exp = Exp + 1
```

For the test of whether LoopPart followed by WhileLoop produces the same result, we have the following pseudocode:

```
2         Mant = Mant/10
3         if NegEx then Exp = Exp - 1
4         else Exp = Exp + 1
5    while Mant >= 10 do
6         Mant = Mant/10
7         if NegEx then Exp = Exp - 1
8         else Exp = Exp + 1
```

This equality can be easily demonstrated either with a trace table or by carefully working through the loop logic. The first part of this equality in the top table cycles through the while test until the value of Mant is less than 10. If it started with Mant >= 10 and less than 100, for example, there would be one cycle of the while loop and the final result would be Mant/10 and Exp - 1. Mant is now equal to or greater than 1 and less than 10.

Under these same conditions, with Mant between 10 and 100, the second part of the equality would take one pass through LoopPart in steps 2 through 4, leaving the result of Mant/10 and Exp - 1. In this case, the while test fails and steps 5 through 8 are not executed. In the general case, the while loop in the second part of the equality would be performed one less time than the first while loop, leaving the identical result. The equality is thus true.

The second verification case is identical, except with !NegExp. Here, the results are also identical, except that the resulting Exp values are increased to Exp + 1 when Mant starts between 10 and 100.

The cases for the second while loop, with Mant < 1, are handled the same way, so their verification is not repeated.

The third question for the while test is, when WhileTest is false, does WhileLoop = identity? This condition can be demonstrated by inspection. The first while loop is as follows:

```
1    while Mant >= 10 do
2         Mant = Mant/10
3         if NegEx then Exp = Exp - 1
4         else Exp = Exp + 1
```

Here, when Mant < 10, the entire loop is skipped and step 5 is next executed. This means that nothing is changed, and this logic is equivalent to identity. This same reasoning can be used with the second while loop. The program with the change made to correct the defect when Mant = 10 thus passes these tests and is correct.

Repeat-Until Verification

The repeat-until verification is similar to that for the while test. Before doing the repeat-until verification, identify the repeat-until preconditions and verify that they are always met. These tests assume repeat-until has the following form:

```
RepeatUntil
     begin
          LoopPart
          UntilTest
     end
```

In the repeat-until loop, the until test is performed after the LoopPart, so the LoopPart is always executed at least once, regardless of the initial conditions. The repeat-until verification questions are as follows:

1. Is loop termination guaranteed for any argument of UntilTest?
2. When UntilTest after LoopPart is false, does RepeatUntil = LoopPart followed by RepeatUntil?
3. When UntilTest after LoopPart is true, does RepeatUntil = LoopPart?

As before, answer the second question by constructing a trace table or studying the logic for RepeatUntil and the logic for LoopPart followed by RepeatUntil. The third question can be similarly answered with the aid of a trace table if needed.

12.7 Other Analytical Verification Methods

In addition to compilation and testing, both manual and automated verification methods can be helpful. Some of these methods provide more general proofs than are possible with test cases or other single-case verification practices. Analytical verification methods are particularly useful for verifying the logic of an existing program before modifying or correcting it or when verifying the logic of new programs.

Because analytical verification methods generally require some training and considerable practice, this section provides only an overview of the key topics to consider in using analytical verification. It also describes a few of the currently available automated verification methods. As previously noted, however, new methods will almost certainly be developed, so you should watch the technical literature and use your PSP data to determine which methods are most effective for you.

Analytical Verification Issues

The key issues to consider in selecting an analytical verification method concern the usefulness and effectiveness of the method. In addition, because various vendors offer tools for automated verification, it is important to consider how effective these tools are and the degree to which they can help you to economically and quickly produce quality products. In discussing verification methods, the following terms are commonly used:

- **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?

Automated Verification

Although automated verification methods can be very helpful, they can also be dangerous. The problem is not with the tools but with how they are used. For example, many programmers assume that their compiler will find all of their simple coding mistakes. Although good compilers will identify all of the cases in which the programmer produced incorrect syntax, they will miss those cases where the programmer made a mistake that accidentally produced valid syntax but incorrect logic.

A principal reason why many verification tools find such a low percentage of the logic defects is that the programs they analyze are written in imprecise languages such as C, C++, and Ada. Moreover, these tools often have no way to determine what the programmer intended to do. When programs use a precisely defined language and when the programmers precisely describe their intentions in program annotations, tools can often find 95% or more of many defect types (Barnes 2003). Programming with these methods is generally called **design-by-contract**.

PSP data show that with traditional languages, compilers generally miss about 10% of the typographical or simple coding mistakes we make. The problem is that when a tool provides around 90% or higher completeness, it is easy for programmers to fall into the habit of believing that it has found 100% of their mistakes, and not bother trying to find the rest.

Suppose you had a tool that would reliably find only 10% of all of your logical design mistakes. How would you use it? One possible strategy would be to use the tool first in hopes of fixing as many problems as you could before doing personal reviews or inspections. However, this would not reduce your workload by much and you would get limited benefit from the tool. A counterstrategy would be to do your personal reviews and possibly even your inspections before running the tool. Then, when you ran the tool, you would have a way to measure the quality of your reviews and inspections. If, for example, the tool didn't find any defects, you probably found most of them yourself. Conversely, if the tool identified 11 defects, you could assume that you probably missed over 100 defects and that you had better make a more concerted effort to clean up the program before putting it into testing.

Although this should not be an emotional issue, it often is. Many programmers who have not completed PSP training become upset when I suggest that they review their programs before compiling them. They would presumably have the same reaction to reviewing their designs before using an automated tool to find buffer overflows, for example. However, the available evidence indicates that with commonly used languages and methods, automated tools for finding complex defects such as buffer overflows are rarely more than about 25% complete.

The danger is not with tools that are 25% complete, as most developers quickly learn that they can't trust them. The problem is with tools that are 90% or 99% or even 99.9% complete. Now the key question is, how important is it to produce a defect-free product? If it is a life-or-death matter, you must do the review first and use the tool to provide the information needed to evaluate the quality of your work. Then you can logically decide on the proper quality-management steps.

To rationally make such decisions, you need the following information:

- How sound, complete, and efficient is the tool?
- How does the completeness and efficiency of the tool change when run before or after your personal reviews and inspections?
- How do the completeness (yield) and cost of your reviews and inspections change when they are run before or after the tool?
- What is the variability of these data? That is, does your personal yield or the tool's completeness vary widely or is it reasonably consistent?
- What is the expected cost of missed defects?

After they examine their PSP data, most developers conclude that doing careful reviews before compiling or testing their programs pays off in saved time and improved program quality. Although I suspect that would also be true for any verification tool that was less than 100% complete, the only way to know is to run experiments, gather data, and make a rational fact-based decision.

Verification Methods

Considerable work has been done on automatic verification methods. This section describes some of these methods. The following is only a limited listing of the available materials.

ASTREE

The ASTREE static analysis tool was developed in 2001 and has been used on several high-dependability programs with considerable success. It uses abstract interpretation to identify possible runtime errors (Blanchet et al. 2003).

Microsoft
Microsoft has produced a number of tools both for internal use and for general availability (Larus et al. 2004). Early examples are the PREfix and PREfast tools that perform various types of static analysis. PREfix has been used on the Windows products with considerable success. PREfast performs local analysis on C and C++ programs, with a principal focus on programming mistakes. More recent Microsoft verification tools include SLAM, ESP, and Vault.

SPARK
The SPARK language and Examiner tool statically analyze programs written in the SPARK language (Barnes 2003). This approach restricts the programmer to a clearly defined and unambiguous Ada subset (the SPARK language) and then utilizes programmer-written annotations to analyze and prove program correctness.

Splint
The Splint tool performs static checking of C programs for security vulnerabilities and coding mistakes. It was produced by the Computer Science Department at the University of Virginia (Yang and Evans 2004).

Because the effectiveness of automated design verification and other formal methods depends on the clarity and precision of the design description, developers using these techniques often use a formal notation based on mathematical methods to document their designs. One widely used formal notation is Z (pronounced "zed"), for which many support tools are available (Jacky 1996). As the safety and security problems with software developed with current methods increase, more formal methods will have to be adopted. If you are serious about producing truly high-quality programs, you should become familiar with such methods.

12.8 Verification Considerations

When designing complex logic, just about everybody makes mistakes. Therefore, it is important to thoroughly verify every design; and the more complex the design, the more important it is to use proven and practiced verification methods. This, in turn, suggests that you identify and consistently use a family of verification techniques. To build verification skill, use these methods even when you write small programs. If you don't use these methods with small programs, you will not have the skill, experience, and conviction to use them when you are verifying larger programs.

Verification Strategy

For small programs like the examples in this chapter, the pseudocode logic will look like code, but more complex programs will generally have several classes and methods. You can merely name these classes and methods in the trace table, but a complete design must specify the detailed logic for every one of the main program's loop constructs. In producing the trace table, enter every program pseudocode instruction, but just name the classes and methods. If you know the behavior of these higher-level constructs, you can treat them as single trace-table steps. In the trace-table analysis, just enter the result from each class or method without tracing through its execution. This suggests a general verification strategy: design down and verify up. That is, when designing each routine, specify the functions to be provided by each class and method. Then start the design review process at the bottom and verify each class or method before verifying the high-level routines that use them. That way, you can be confident that each class or method works properly when you encounter it in a higher-level verification.

The best time to use the design verification and analysis methods described in this chapter is while you are producing the design. While doing the analysis work, make a permanent record of the results and keep this record with the program's design documentation. Then, when you do the design review, follow the Design Review Script shown in Table 12.9 and the Design Review Checklist shown in Table 12.10 to check what you did during the design work. Ensure that the design analysis covered all of the design, was updated for any subsequent design changes, is correct, and is clear and complete. If you did not complete any of the analyses, complete and document them during the design review. Note that these design review script and checklist tables are different from the PSP2 versions shown in Chapter 9 because these include use of the verification methods described in this chapter. Use this script and checklist hereafter in verifying your designs.

Verification Practice

Verification methods take time, and as programs get larger and more complex, the verification methods get more complex and take even more time. Thus, as the need for these methods increases, they become progressively harder to use. Although there is no simple solution to this problem, the key is to learn which verification methods work best for you and then to use these methods on every program you develop. If you do this consistently, your large programs will be composed of multiple small programs that you have already verified. The verification problems will then be reduced to ensuring that the small programs were properly combined. Although this might not be a trivial check, it is far simpler than checking the details of every small program.

TABLE 12.9 DESIGN REVIEW SCRIPT

Purpose		To guide you in reviewing detailed designs
Entry Criteria		• Completed program design ***documented with the PSP Design templates*** • Design Review checklist • Design standard • Defect Type standard • Time and Defect Recording logs
General		Where the design was previously verified, check that the analyses • covered all of the design and were updated for all design changes • are correct, clear, and complete Where the design is not available or was not reviewed, do a complete design review on the code and fix all defects before doing the code review.
Step	**Activities**	**Description**
1	Preparation	• Examine the program and checklist and decide on a review strategy. • ***Examine the program to identify its state machines, internal loops, and variable and system limits.*** • ***Use a trace table or other analytical method to verify the correctness of the design.***
2	Review	• Follow the Design Review checklist. • Review the entire program for each checklist category; do not try to review for more than one category at a time! • Check off each item as you complete it. • Complete a separate checklist for each product or product segment reviewed.
3	Fix Check	• Check each defect fix for correctness. • Re-review all changes. • Record any fix defects as new defects and, where you know the defective defect number, enter it in the fix defect space.
Exit Criteria		• A fully reviewed detailed design • One or more Design Review checklists for every design reviewed • ***Documented design analysis results*** • All identified defects fixed and all fixes checked • Completed Time and Defect Recording logs

In practicing verification methods, track your time and the defects you find and miss and use these data to improve your verification practices. Try whatever methods seem potentially most useful, but be guided by your data. After you have tried several methods and used them enough to build reasonable competence, your data should indicate the methods that work best for you.

TABLE 12.10 DESIGN REVIEW CHECKLIST

Student	__________	Date	__________
Program	__________	Program #	__________
Instructor	__________	Language	__________

Purpose	To guide you in conducting an effective design review
General	• Review the entire program for each checklist category; do not attempt to review for more than one category at a time! • As you complete each review step, check off that item in the box at the right. • Complete the checklist for one program or program unit before reviewing the next.

Complete	Verify that the design covers all of the applicable requirements. • All specified outputs are produced. • All needed inputs are furnished. • All required includes are stated.				
External Limits	Where the design assumes or relies upon external limits, determine if behavior is correct at nominal values, at limits, and beyond limits				
Logic	***Use a trace table, mathematical proof, or similar method to verify the logic.*** • Verify that program sequencing is proper. • Stacks, lists, and so on are in the proper order. • Recursion unwinds properly. • Verify that all loops are properly initiated, incremented, and terminated. • Examine each conditional statement and verify all cases.				
State Analysis	***For each state machine, verify that the state transitions are all complete and orthogonal.***				
Internal Limits	Where the design assumes or relies upon internal limits, determine if behavior is correct at nominal values, at limits, and beyond limits.				
Special Cases	• Check all special cases. • Ensure proper operation with empty, full, minimum, maximum, negative, zero values for all variables. • Protect against out-of-limits, overflow, underflow conditions. • Ensure that “impossible” conditions are absolutely impossible. • Handle all possible incorrect or error conditions.				
Functional Use	• Verify that all functions, procedures, or methods are fully understood and properly used. • Verify that all externally referenced abstractions are precisely defined.				
System Considerations	• Verify that the program does not cause system limits to be exceeded. • Verify that all security-sensitive data are from trusted sources. • Verify that all safety conditions conform to the safety specifications.				
Names	Verify that • all special names are clear, defined, and authenticated • the scopes of all variables and parameters are self-evident or defined • all named items are used within their declared scopes				
Standards	Ensure that the design conforms to all applicable design standards.				

12.9 Summary

The objective of design verification is to determine whether a design meets its requirements and is correct. This means that the program's specifications and standards are an important part of the verification process. The standards of concern are product conventions, product design standards, and reuse standards.

There are many ways to determine the correctness of a design before you implement it. The methods described in this chapter are trace-table checking, state-machine analysis, and analytical verification. You can use various combinations of these methods on most programs and they are both relatively fast and effective. Using these methods saves time because when your designs are complete, clear, and correct, the implementation will be faster and the finished program will have few (if any) design defects.

Although trace-table verification can be time-consuming, it is much faster and more effective than verifying all possible cases with testing. Because most of the work for a trace-table analysis must be done to properly use a debugger, it is faster and more efficient to use that analysis effort to verify the program's correctness with a trace table. With a completed trace table, it is then relatively easy to run any needed tests.

When a product includes a state machine, verify that the state machine is properly designed and consistently used. The conditions for a proper state machine are twofold: all state transitions should be complete and orthogonal and the machine must be able to reach a program return state from every other state.

Verifying design correctness is simple in concept, but it takes time to build sufficient skill to do it quickly and efficiently. To build this skill, use these verification methods on every program you develop and measure and track your work to determine the verification methods that are most effective for you. Then use these data to regularly analyze and improve your design-verification skills.

12.10 Exercises

The assignment for this chapter is to use PSP2.1 to write one or more programs. For these programs, use the PSP design templates and the state-machine and trace-table verification methods. The PSP2.1 process and the program specifications are given in the assignment kits, which you can get from your instructor or at **www.sei.cmu.edu/tsp/psp**. The kits contain the assignment specifications, example calculations, and the PSP2.1 process scripts. In completing the assignment, faithfully follow the PSP2.1 process, record all required data, and produce the program report according to the specifications given in the assignment kits.

References

Barnes, J. *High Integrity Software: The SPARK Approach to Safety and Security.* Reading, MA: Addison-Wesley, 2003.

Blanchet, B., P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. "A Static Analyzer for Large Safety-Critical Software." In *PLDI 2003—ACM SIGPLAN SITSOFT Conference on Programming Language Design and Implementation,* 2003 Federated Computing Research Conference, San Diego, Calif., June 7–14, 2003, 196–207.

Butler, R. W., and G. B. Finelli. "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software." *IEEE Transactions on Software Engineering* 19, no. 1, January 1993.

Dyer, M. *The Cleanroom Approach to Quality Software Development.* New York: John Wiley & Sons, 1992.

Gage, D., and J. McCormick, "Can Software Kill?" *eWeek Enterprise News and Reviews,* March 8, 2004. www.eweek.com.

Jacky, J. *The Way of Z: Practical Programming with Formal Methods.* Cambridge, UK: Cambridge University Press, 1996.

Larus, J. R., T. Ball, M. Das, R. DeLine, M. Pahndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy. "Righting Software." *IEEE Software* (May/June 2004): 92–100.

Littlewood, B., and L. Strigini. "Validation of Ultrahigh Dependability for Software-Based Systems." *Communications of the ACM* 36, no. 11, November 1993.

Mills, H. D., V. R. Basili, J. D. Gannon, and R. G. Hamlet. *Principles of Computer Programming: A Mathematical Approach.* Newton, MA: Allyn and Bacon, 1987.

Yang, J., and D. Evans. "Dynamically Inferring Temporal Properties." *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2004),* June 7–8, 2004.