Chapter 4
Software Testing Roundup

J. C. Huang
Department of Computer Science
University of Houston

Topics

A few more points to ponder before wrapping up the discussion on testing:

- Ideal test sets
- Operational testing
- Integration testing
- Testing of object-oriented programs
- Regression testing
- Criteria for stopping the test
- Choosing a test-case selection criterion

State of the art

- At present, there does not exist a test method that allows us to conclude, from a successful test, that a program does not contain any error.

- If such a method exists, what properties it should have?

Notations

Let

S be a program,

D be the input domain of S,

S(d) denote the result of executing S with input d$\in$ D,

Notations (continued)

T	be a subset of D, called a test set,
OK(d)	be a predicate which becomes true if and only if program S terminates cleanly for an execution with input d and S(d) is an acceptable result.

Ideal set of test cases

Elements of T are called *test cases*.

T constitutes an *ideal set of test cases* if

$$(\forall t)_T(OK(t)) \supset (\forall d)_D(OK(d)).$$

If we can find an ideal set of test cases, we can conclude from a success test that the program contains no error.

Successful test

A test using T is said to be *successful* if the program executes correctly with every element of T. Formally,

$$\text{SUCCESSFUL}(T) \equiv (\forall t)_T(\text{OK}(t)).$$

How a set of test cases is selected?

Typically, a set of test cases is a subset of D selected to satisfy some test-case selection criterion, C, which is usually a predicate over 2^D, the power set (i.e., the set of all subsets) of D.

Reliable selection criteria

A test-case selection criterion, C, is said to be *reliable* if and only if the program succeeds or fails consistently when executing a set of test cases satisfying C. Formally,

$$\text{RELIABLE}(C) \equiv (\forall T_1)_{2^D}(\forall T_2)_{2^D}((C(T_1) \wedge C(T_2)) \supset (\text{SUCCESSFUL}(T_1) \equiv \text{SUCCESSFUL}(T_2)))$$

Valid selection criteria

A test-case selection criterion, C, is said to be *valid* for a particular program if and only if there exists a set of test cases satisfying C which will cause the program to fail the test if the program is incorrect. To be more precise,

VALID(C)

$\equiv (\exists d)_D(\neg OK(d)) \supset (\exists T)_{2^D}(C(T) \wedge \neg SUCCESSFUL(T))$

On validity

Note that validity does not imply that every set of test cases selected with C will cause the program to fail the test if the program is incorrect.

Example

Suppose the program is intended to double the input, but instead, it squares the input. To express it in the formalism introduced above,

$F(d) = d \times d$ for all $d \in D$ and

$OK(d) \equiv (F(d)=d+d)$.

The program is intended to compute the remainder of d÷5, but instead it computes the remainder of d ÷3, where "÷" denotes the operation of integer division. To express it in the formalism introduced above,

$S(d) = (d \ mod \ 3)$ for all $d \in D$ and

$OK(d) \equiv S(d)=(d \ mod \ 5)$.

Example (continued)

Observe that the program works correctly for all integers 15k+0, 15k+1, and 15k+2, that is, OK(15k+0), OK(15k+1), and OK(15k+2), for all non-negative integer k, but it works incorrectly otherwise.

Example (continued)

$C_1(T) \equiv (T=\{1\}) \vee (T=\{2\})$

(reliable but not valid)

Example (continued)

$C_2(T) \equiv (T=\{t\}) \wedge (t \in \{0, 1, 2, 3, 4\})$

(not reliable but valid)

Example (continued)

$C_3(T) \equiv (T=\{t\}) \wedge (t \in \{3, 4, 5, 6\})$

(reliable and valid)

Example (continued)

$C_4(T) \equiv (T=\{t, t+1, t+2, t+3\}) \wedge (t \in D)$

(reliable and valid)

Goodenough and Gerhart's result:

A test as a proof of correctness

Theorem 4.1: A successful test constitutes a direct proof of program correctness, if it is done with a set of test cases selected by a test criterion which is both reliable and valid. Formally,

$$(\exists C)(VALID(C) \wedge RELIABLE(C) \wedge (\exists T)_{2^D}(C(T) \wedge SUCCESSFUL(T))) \supset SUCCESSFUL(D)$$

Comment

- In general, it is difficult to prove the validity and reliability of a test-case selection criterion. In some cases, however, it may become trivial.

- For example, the proof of validity becomes trivial if C, the test-case selection criterion, does not excludes any member of D, the input domain, from being selected as a test case.

Observation

- If C does not allow any member of D to be selected, then C can be valid only if the program is correct.
- Thus, in that case, it is required to prove the program's correctness in order to prove the validity of C.

Observation

- The proof of reliability becomes trivial if C requires selection of all elements in D because in that case there will only be one set of test cases.
- The proof of reliability also becomes trivial if C does not allow any element from D to be selected.

Second question

- Does there exist an effective procedure to find an ideal set of test cases?
- An ideal set of test case for a program is the one with which a successful test constitutes a direct proof of the correctness.

Howden's result

Theorem 4.2: There exists no computable procedure H which, given an arbitrary program S with domain D, can be used to generate a nonempty finite set $D \supset T$ such that.

$$(\forall t)_T(OK(t)) \supset (\forall d)_D(OK(d)).$$

Operational testing

- The test cases are selected based on an operational profile.
- USE(d), a predicate, is true if d is used in a production run of the program.
- p(USE(d)) is the probability that d will be used in a production run.
- A distribution of p(USE(d)) over the elements of the input domain is called an *operational profile*.

Operational testing (continued)

In other words, the test set to be used in an operational testing is

$$T_o = \{ t \mid t \in D \wedge p(USE(t)) \geq \mu \}.$$

where μ is some constant, and D is the input domain.

Operational testing (continued)

The value of μ should be chosen in such a way that the sum of p(USE(t)) over all elements in T_o should be greater than or equal to some predetermined constant γ.

If a test with T_o is successful, we can conclude that the program is $\gamma \times 100\%$ reliable.

Operational vs. debug testing

Elements of T_o are those inputs that have high probabilities of being used in a production run. Therefore, if a fault is revealed in an operational test, removal of that fault will contribute significantly to the enhancement of the program reliability.

Operational vs. debug testing (continued)

This is in contrast to a debug test in which a test case t might have a very high p(¬OK(t)) but very low p(USE(t)). If it reveals a fault, and if that fault is removed, it would have a relatively small impact on the program reliability.

Operational vs. debug testing (continued)

On the other hand, the reliability achieved through debug testing remains valid unless the program is modified in some way.

The same cannot be said about the reliability assessed through an operational test. Operational profiles change in time and with application environments. Consequently, it is entirely possible that, in another time or another place, the same software system may in fact not as reliable as it is claimed to be.

Integration testing

Integration testing is a process in which the components of a software system are integrated into a functioning whole. After we have all components completed and thoroughly tested, we have to put them together to form an executable image of the entire system.

The futility of "big-bang" approach

Unless the system is very small, consisting of just a few components, we cannot integrate the system simply by compiling each component and then linking them together. One cannot expect the program to work flawlessly the first time around. With all components integrated at the same time, it would be very difficult to isolate the problems and locate the sources of the problems.

Incremental approach

A better way is to integrate the components incrementally. We start by putting two components together to form a subsystem and test it thoroughly. We then proceed to integrate one component at a time into the subsystem so formed until the entire system is completed.

Incremental approach (continued)

Integrate the system incrementally makes it easier to locate and remove the faults. Furthermore, there is no need to recompile and relink all components every time when a change is made to a component.

Bottom-up integration

This approach starts with unit testing, followed by subsystem testing, followed by testing of the entire system.

Bottom-up approach (continued)

Modules are tested in isolation from one another in an artificial environment known as a *test harness*, which consists of the driver programs and data necessary to exercise the modules. Modules are combined to form subsystems, and subsystems are then integrated to form the system as a whole.

Bottom-up approach (continued)

This approach is particularly appropriate for programs like operating systems in which the lowest level modules are device drivers and service routines. It would be difficult to tell if the program is working correctly or not without those lower-level modules in place.

Bottom-up approach: advantages

1. Unit testing is eased by a system structure that is composed of small, loosely coupled modules.
2. Since most input-output operations are done by the lower level modules, how to feed test cases to the program is less of a problem.

Bottom-up approach: disadvantages

1. The necessity to write and debug test harness. Test harness preparation can amount to 50% or more.
2. The necessity to deal with the complexity resulting from combining modules and subsystems into larger and larger units.

Top-down approach

It starts with the main routine and one or two immediately subordinate routines in the system structure.

After this top-level skeleton has been thoroughly tested, it becomes the test harness for its immediately subordinate routines.

Top-down approach (continued)

Top-down integration requires the use of program stubs to simulate the effect of lower-level routines that are called by those being tested.

Top-down approach: advantages

1. System integration is distributed throughout the implementation phase; modules are integrated as they are developed.
2. Top-level interfaces are tested first and most often.
3. The top-level routines provide a natural test harness for lower-level routines.
4. Faults are localized to the new modules and interfaces that are being added.

Top-down approach: disadvantages

1. Sometimes it may be difficult to find top-level input data that will exercise a lower-level module in a particular manner.
2. The evolving system may be very expensive to run as a test harness for new routines on a lower level.
3. It may be costly to re-link and re-execute a system each time a new routine is added.
4. It may not be possible to use program stubs to simulate modules below the current level.

Sandwich integration

This approach is predominantly top-down, but bottom-up techniques are used on some modules and subsystems.

This mix alleviates many of the problems encountered in pure top-down testing and retains the advantages of the top-down integration at the subsystem and system level.

Sandwich integration (continued)

Start by finding a way to form a small subsystem consisting of modules on all levels to minimize the need of test drivers and stubs, and then proceed to add one module at a time to complete the integration.

Sandwich integration (continued)

This approach also makes it easier to determine what should be the test cases, and what the expected corresponding outputs are.

This approach will lead to the production of a running subsystem sooner.

Phased integration

Units at the next level are integrated all at once, and the system is tested before integrating all modules at the following level.

Controversies about unit testing

Unit testing is a point of contention often raised in debates over the strategies to be used in integration testing.

Those who support say unit tests "let the engineers locate the problem areas and causes within minutes."

Those who oppose it advocate that units are not tested in isolation, but are tested only after integration into a system.

An experimental result

Experiments conducted by Solheim and Rowland [SORO93] indicated that the top-down strategies generally produce the most reliable systems and are the most effective in terms of fault detection.

The higher fault detection rate appeared to be caused by the fact that the top-down strategy exercises more modules per test case than do the other strategies.

Testing object-oriented programs

All test methods discussed in the preceding chapters were developed for programs written in a procedural language. Those test methods should, in principle at least, remain applicable to object-oriented programs because an object oriented language is a procedural language as well.

Testing OO programs (continued)

Nevertheless, there are some complicating factors idiosyncratic to the use of an object-oriented language/paradigm that require certain changes in the ways those methods are to be applied.

Common practice

The common practice in debug-testing a large software system is that we do unit testing first, followed by integration testing.

Common practice (continued)

The chunk of source code to be unit-tested should be small so that, if the test failed, the source of the failure can be readily located and removed. On the other hand, it must be large enough to constitute a complete syntactic unit that can be separately compiled.

For a traditional software system written in C, it is commonly taken as a function or a set of related functions.

What is a "unit" in OO programs?

A method in an object-oriented program has the form of, and works like, a function in C. Nevertheless it cannot be treated as a unit in unit-testing because it is not separately compilable. It is encapsulated in a larger program unit called a class.

Therefore, a class, or a set of related classes, appears to be an appropriate unit for unit testing.

What we should know in making this choice

A class usually contains a storage unit that is implemented as some data structure in the private part that is not directly accessible to the test harness. Thus if we test-execute the unit with an input that causes changes to the content of the storage, the tester will have no way to determine if the correct changes have been made.

What we should know (continued)

A possible solution to this problem is to require the designer to make the class testable by including additional methods in the class that can be invoked to inspect the internal state of the class.

What we should know (continued)

A class is a program unit with memory. Its response to an input is influenced by previous inputs (or ordering of the inputs) as well.

What we should know (continued)

For instance, the sequence of inputs “PUSH”, “POP” will produce a test result different from that of “POP”, “PUSH”.

This phenomenon is not unique, but is far more common, to object-oriented programs.

Messages

Whatever that we wish to do with an object it has to be accomplished by invoking its methods in sequence, and thus can be specified by a sequence of identities of methods. Such a sequence is called a *message* in the language of the object-oriented technology

Valid messages

When a program unit in an object-oriented program needs to do something by making use of methods, it passes messages to invoke those methods.

A message is said to be *valid* if it is a sequence of invocations of methods in the class that will cause the methods to be executed with inputs from their intended input domains.

Message graphs

The validity of a message can be defined by a directed graph called *message graph*.

There shall be as many nodes as the number of methods in the class, each of which is uniquely associated with a method. There shall be an edge emanates from node A and terminates at node B if it is permissible to send message A followed by message B.

An example stack class

Figure 4.1 shows an example message graph of a stack class. With this class, one can create a new stack, push an element on the stack, pop an element from the top, or examine the top element by using the methods named NEW, PUSH, POP, and TOP, respectively.

The message graph

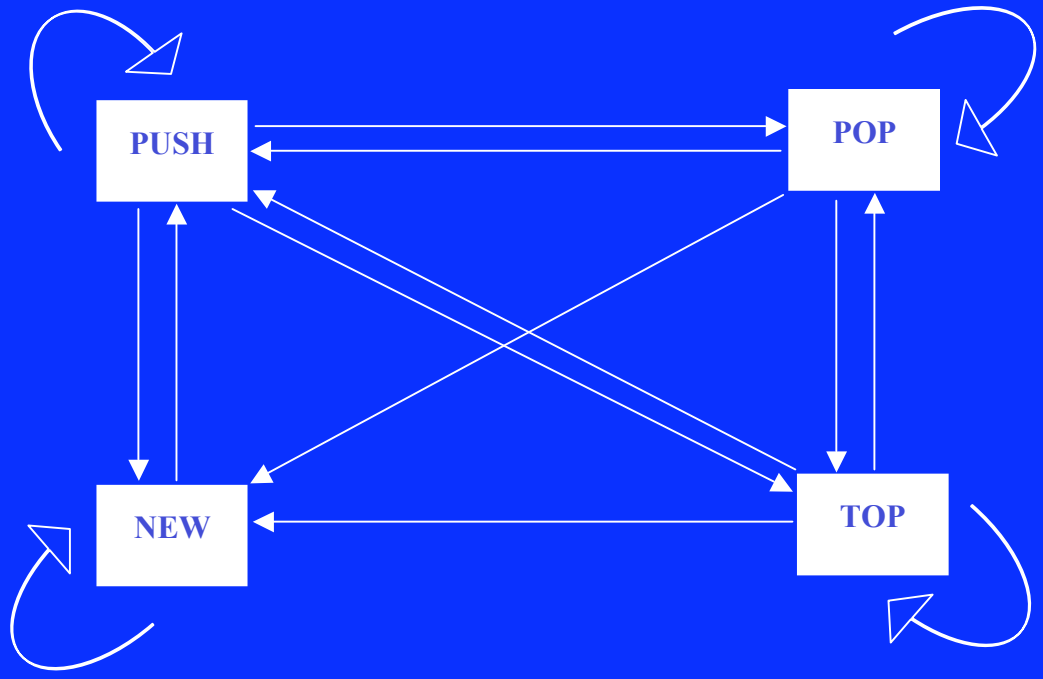


Figure 4.1. Message graph for a simple stack module.

What the graph shows

This graph shows that message NEW cannot be followed by message POP or TOP because the stack is empty immediately after the message NEW.

A message is valid if and only if it forms a path in the message graph, e.g., NEW PUSH POP NEW is valid, but NEW PUSH POP NEW POP PUSH is not.

Test-case selection

Choose a valid message as a test case if the goal is to cause a certain program component to be exercised during the test.

Choose an invalid messages as a test case if the purpose is to determine if appropriate exceptions would be raised.

Test-case selection (continued)

The test-case selection methods discussed in the preceding chapters can be applied to unit-test a class as usual. A possible complicating factor is that the test cases are now messages instead of mere input data. If a message is chosen to cause a particular component in a method to be exercised, and if that message turns out to be invalid, it has to be replaced by a valid message that will serve the same purpose.

The question of inheritance

Because some classes in an object-oriented program may be created by making use of inheritance, the question may arise as to the extent to which a class has to be unit-tested if it is similar in some way to another that has been completely unit-tested.

The question of inheritance (continued)

Weyuker et al. (1988) has proposed 11 axioms for checking the completeness of a test [FRWE88], some of which repeated below may be useful in answering that type of questions.

The question of inheritance (continued)

If we replace an inherited method with a locally defined method that performs the same function, will the test set for the inherited method be adequate for the locally defined one?

No. Weyuker's fifth axiom says that an adequate test set for one algorithm is not necessarily an adequate test set for another, even though they compute the same function.

The question of inheritance (continued)

Is the same test set adequate for two programs of the same shape, i.e., if one can be transformed into another through simple replacement of one or more relational/arithmetic operators, or one or more constant?

No, according to the Weyuker's sixth axiom or the second principle of test-case selection.

The question of inheritance (continued)

Does a program need to be retested if the context changes?

Yes, according to Weyuker's 7th axiom.

The question of inheritance (continued)

If we change the underlying implementation of an object and keep the interface intact, is it insufficient to test the modified object in isolation?

No, according to Weyuker's 8th axiom. We will have to retest all dependent units as well.

Integration testing of OO programs

The guidelines for integration testing discussed in the preceding section remains mostly applicable to an object-oriented software system. Classes will be unit-tested in isolation, and then be integrated into the system incrementally for integration testing.

Integration testing of OO programs
(continued)

For traditional programs, integration testing can be done either in top-down or bottom-up manner. Unfortunately, there is no clear top or bottom in the organization of an object-oriented program (i.e., no invocation hierarchy). Therefore, it is difficult to develop a general rule.

Integration testing of OO programs
(continued)

In general, one may start with a small set of classes that can be used to produce an executable image, and then expand it by adding one class at a time. The order in which the classes are integrated into the system will affect (a) the amount of test harness required, (b) the number of new and useful test cases we can find. Therefore, it should be chosen accordingly to optimize the test efficiency.

Regression testing

Regression testing should be performed to a program after it is modified to remove a fault or to add a function.

Regression testing (continued)

The process is to rerun all or some of the previous tests, and introduce new test cases if necessary, to assure that no errors have been introduced through the changes or to verify that the software still performs the same functions in the same manner as its older version.

Regression testing (continued)

A library of test cases is needed to make this test possible. It should be built and maintained in such a way that past test cases and corresponding symbolic traces can be efficiently stored and retrieved.

Criteria for stopping the test

If we select test cases based on the operational profile, or if the test cases are chosen randomly from the input domain, new elements can be added to the test set being constructed endlessly. Under those circumstances the question of when to stop testing becomes germane.

Criteria for stopping the test (continued)

An obvious answer to this question is to dilute the test requirement by reducing the number of test cases to be used. For instance, it is not unusual to see that a software development contract includes a stipulation allowing only some (say, 60 percent), instead of all, statements in the source code be exercised at least once during the test.

Criteria for stopping the test (continued)

There is, however, a more elegant way to determine when to stop the test. S. A. Sherer (1991) has shown that it is possible to estimate the cost and benefit of doing program testing methodically, and therefore it is possible to find the optimal time to stop.

Cost and benefit estimation

S. A. Sherer (1991) has shown that it is possible to estimate the cost and benefit of doing program testing methodically, and therefore it is possible to find the optimal time to stop testing.

Cost and benefit estimation (continued)

She started by identifying potential faults in the program, assessing the possible economical impacts of failures that may be caused by those faults, and estimating the probabilities of such failures during a predetermined operation period to compute R, the risk.

Cost and benefit estimation (continued)

The theory provides a formula to compute ΔR, the amount of reduction in R resulting from detection followed by removal of faults in the program through debug testing. It is assumed that no new fault is introduced in the debugging process.

Cost and benefit estimation (continued)

The theory also provides a way to estimate C, the cost of testing, which is essentially the sum of the cost of machine time and the labor required to do testing and debugging.

Cost and benefit estimation (continued)

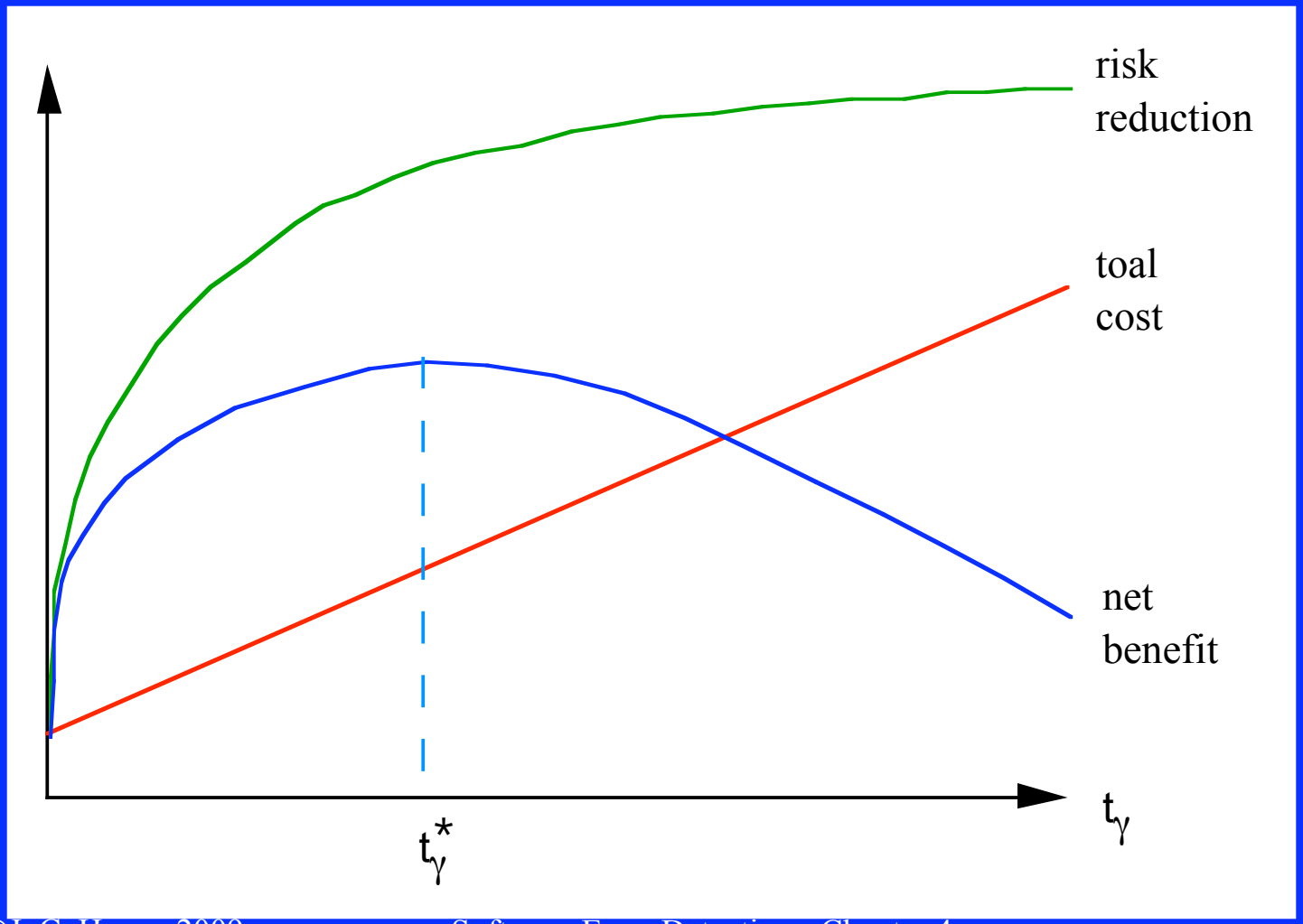
The net benefit (NB) of doing the debug testing, therefore, is equal to the reduction in risk minus the cost of testing, i.e., NB = ΔR - C.

The cost of testing increases almost linearly with the time for doing the test. Initially, the latent faults in the program are more numerous and thus easier to find. Thus ΔR, the reduction in risk, tends to increase exponentially with time. Gradually the latent faults become fewer and harder to find, and the magnitude of ΔR starts to level off.

Cost and benefit estimation (continued)

The net benefit, NB, therefore increases exponentially at the beginning, then starts leveling off and reaches its maximum at t_γ*. If further testing is done beyond that point, the increase in cost starts to outpace the increase in benefit, and the net benefit decreases, and eventually becomes zero and then negative. The relationship among risk, cost, and benefit is depicted the curves shown in Fig. 4.2.

The net-benefit curve



Optimal time to stop testing

The obvious time to stop the testing is t_γ^*, when the net benefit becomes maximal, or, if the ultra-reliability is required, the point at which the net benefit becomes zero.

Choosing a test-case selection criterion

If a good operational profile is available, and if the required reliability is not ultra-high, choose to do an operational test.

Otherwise do debug testing.

Choosing a criterion (continue)

The best way to do debug testing is to apply the first principle of test-case selection repeatedly to add new test cases to the test set being constructed, and use the second principle to determine when to stop the process.

Choosing a criterion (continue)

The first principle says that, in choosing an additional element for the test set, preference should be given to those that are computationally as loosely coupled to the existing elements as possible.

The second principle says that there should be enough elements in the test set to exercise every component at least once during the test.

Choosing a criterion (continue)

For unit testing in a software development effort, the most prevalent choice of component appears to be program statement. It is commonly known as statement testing that requires every statement in the program to be exercised at least once during the test.

Choosing a criterion (continue)

The choice of component to be exercised can be "upgraded" from program statement to branch in the control flow if the statement test is deemed inadequate.

Choosing a criterion (continue)

If the fault-discovery capability needs to be strengthened further, use the coverage tree depicted in Fig. 2.5 as a guide to select another criterion that has a higher coverage. Generally speaking, the higher the coverage of a criterion, the greater the fault-discovery capability of the test set chosen by using that criterion.

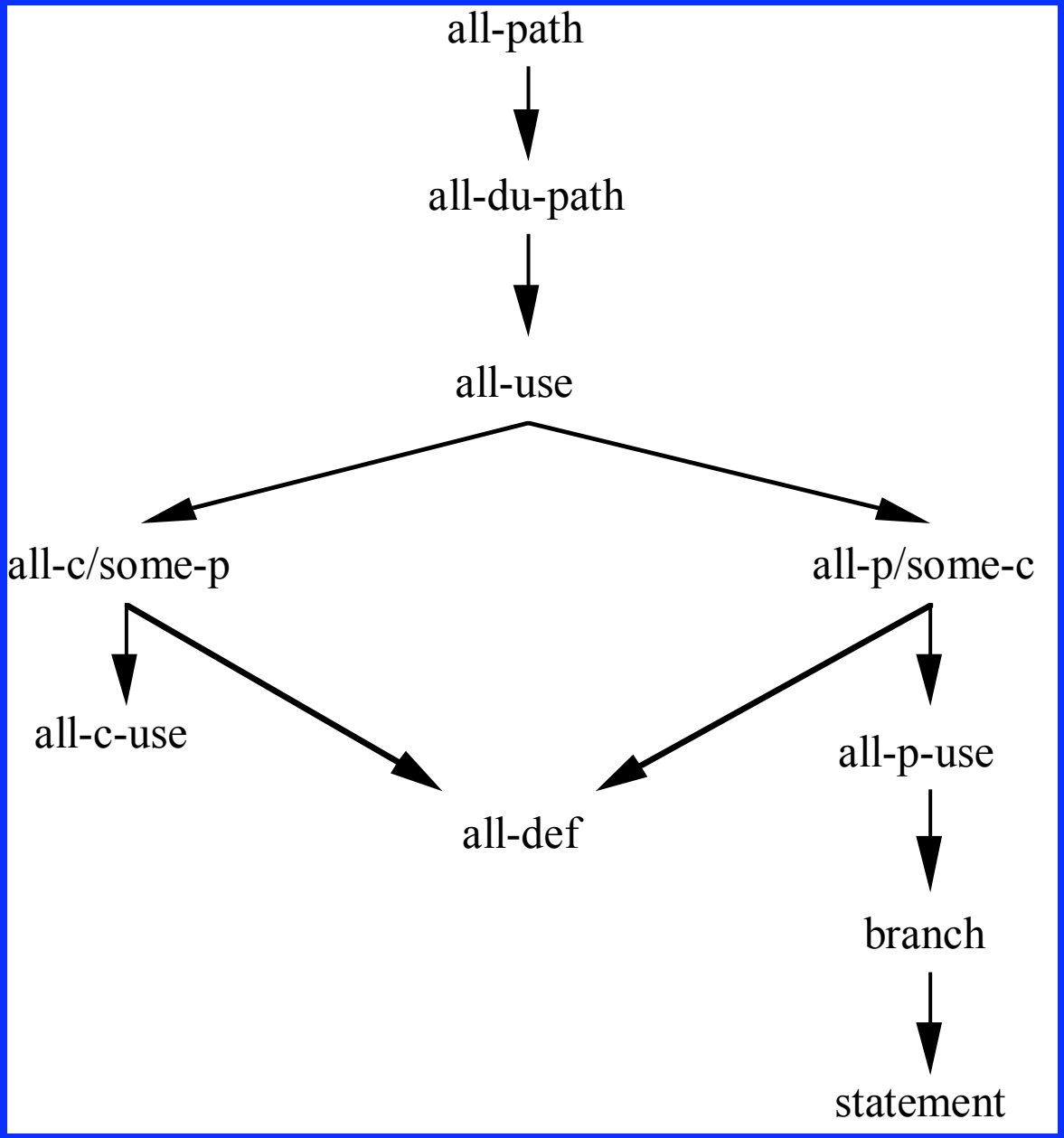


Figure 2.5
The coverage relation

Choosing a criterion (continue)

The fault-discovery capability can also be strengthened by choosing additional type of component to be exercised from the detailed design or program specification as described in Chapter 3.

Choosing a criterion (continue)

If a lesser method is desired due to time or resource constraints, add predicate testing instead, which is effective in discovering faults related to the decomposition of the function specified, or add boundary-value analysis instead, which is effective in discovering faults related to the program's ability to handle the input and output values on, and at the vicinities of, the upper and lower bounds correctly.

Choosing a criterion (continue)

Use error-guessing to choose additional test cases whenever possible because it is relatively easy to do, and many programmers can do it reasonably well.

Choosing a criterion (continue)

In theory, the mutation method appears to be a good testing method if we know for sure that the programmer is competent, and the type of mistakes he or she tends to make can be expressed in terms of a set of mutation rules. In practice, however, its practical value is limited by the large number of test cases it requires.