Solutions to Exercises

Chapter 1

1.1. One can imagine some advantages for "just-in-time" test design. For example, it may be easier to design test cases for which oracles and other test scaffolding are easy to construct. It will place a great deal of test design at the end of a development cycle, where it slows product delivery and places tremendous pressure on test designers to rush and short-cut their work. Overall, it is likely to be a disaster for project schedule, or quality, or both.

1.2. The feasibility study estimates the complexity of the project and its impact on resources and scheduling. Quality requirements have a conceivable impact on the project complexity, the required effort and the overall schedule. Ignoring or badly estimating quality requirements may increase the estimation of errors, and cause losing the contract, in case of large overestimation of resources needed for quality (which is rarely the case), or, more often, economical losses, in case of underestimation of resources for quality. The experience of the quality manager can reduce the estimation error of quality resources and thus the risks of contract or economical losses.

1.3. A test case is designed (and inspected) once, but it is likely to be executed many times. Scaffolding, likewise, is used each time the tests are re-executed. Since re-inspection is much more expensive than re-executing test cases, it is likely that tests will be re-executed much more often than inspection is repeated.

1.4. Prototyping in the spiral model is a risk assessment activity, and measurement and evaluation are vital parts of risk assessment. Designing components or subsystems for incremental measurement and evaluation, with a build plan that permits assessment of each major risk in turn, will (often, but not always) also result in a build plan and major interfaces that support testing each major subsystem at its interfaces. This complementarity can break down, and even become a conflict, when the risks to be resolved are diffused across many parts of the system. For example, a turn of the spiral devoted to assessing performance issues may not help at all in establishing interfaces and build order suitable for testing functional correctness.

1.5. Consider an off-peak period during which about 10 customers are engaged in a download; they must be arriving at a rate around 1 per six minutes. A single 3-minute outage will cause about 10 customers to have to restart a download, and there is about a 50% chance that another arriving customer will be unable to connect. In a given week, approximately 21 customers experience a failure.

If we double availability without changing mean time between failures, then we can expect in the same period a single 1.5 minute outage — forcing the same 10 customers to restart the download but reducing the likelihood of inconveniencing an arriving customer. If we double MTBF without changing availability, then we should expect a single 6-minute outage about once a week. On average we will interrupt those 10 downloads only once a week instead of twice, while on average one arriving customer will be unable to connect. In a given week, approximately 11 customers experience a failure. Clearly, under these assumptions it is better to improve MTBF and sacrifice availability.

Does the situation change if we assume 150 customers are connected when each failure occurs? Now about 2.5 customers arrive each minute, so two 3-minute down-time per week causes approximately 315 customers to experience failure. One 6-minute outage causes 180 customers to experience failure, while two 1.5 minute outages cause 307 customers to experience failure. The heavier the load, the more the advantage of improving MTBF at the expense of availability.

One can easily imagine scenarios in which availability is more important. The key here is that the transactions — 60 minute downloads — are very long. If the transactions are short relative to the down-time, availability will become more important and MTBF less.

1.6. In-store use of the software may differ markedly from using a Web interface at home or in the office. For example, in an in-store environment, one may be less likely to begin a session, take a lunch break, and then finish the session an hour later. It is even less likely that use by store personnel is an accurate model of use by actual customers. Reliability estimates based on this beta testing experience could therefore be misleading.

For a more accurate usage profile, there is simply no replacement for observation of the software in its target environment. This might be obtained, for example, by a second beta release for home use by a small number of users, prior to public release. But since this small sample is likely to be at least partly self-selected, even it may not perfectly reflect use by a wider customer base. Considering that even the "real" customer base can shift or change behavior over time, Chipmunk may never be able to obtain a stable and accurate usage profile.

If Chipmunk developers anticipate some particular differences between observed use and likely use by the target audience, and if sufficient information about observed use and failures can be gathered during beta test, Chipmunk may be able to adjust reliability estimates by weighting behavior under different circumstances. For example, if purchasing on corporate accounts is expected to be more common in the public release than in the in-store beta test period, then use involving corporate accounts may be weighted more heavily in calculating reliability during beta test. One may likewise

factors in uncertainty, experimenting with several different weighting schemes to determine whether reliability would be acceptable under each of several plausible usage patterns.

1.7. The quality manager can offer several arguments in favor of storing complete test data and producing good test documentation.

The main motivation stems from regression testing: test cases are used not only during development, but also after delivery to test new releases. Test cases must be maintained to adapt to changes in the program and in its specifications. Storing only the essential information for automatically re-executing test cases during development may be enough to reduce the test re-execution effort during development, if the size of the project is small enough to allow single test designers to manage test cases without complete documentation, but is insufficient during regression testing, since test cases must be adapted and modified in a long time frame and will be soon impossible to recall motivations and structure of the test suites.

Complete test documentation may also be useful for reusing test cases across different projects. Test reuse may be cost effective especially for large system test suites deigned for general requirements. For example, many financial applications share several general requirements that may call for test reuse.

Finally, complete test documentation may turn useful in case of legal problems with the use of the software, to demonstrate the quality of the test executed before releasing the product. Thorough test documentation is the first requirements of many certification agencies, but may be of general use for any legal dispute.

Chapter 2

2.1. The requirement is not verifiable because it is expressed referring to a subjective sensation (”annoy”), which cannot be measured without referring to actual users.

A simple way to turn the requirement into a verifiable one would be to bind the overall start-up time to a value small enough to avoid users to get annoyed (e.g., 5 seconds). Unfortunately, an upper bound too small may be impossible to satisfy without violating technology and budget constraints, while an upper bound too high may violate the initial requirement.

A better way of formulating the requirement in a verifiable (and satisfiable) way would be to require (1) the identification of a minimum set of functionalities that can be initialized within a short upper bound (e.g., 5 seconds), and that can allow users to start working in quick-mode while to operating is still completing the start-up, and (2) a set of functions that inform the users about the status of the start-up and the time still needed for the system to be operating in quick- and full-mode.

2.2. An *SL* function F is connected to the diagram with a set I_F of input and a set O_F of output flows. A diagrams D that substitute a function F is also connected to the original diagram with a set I_D of input and a set O_D of output flows. Since input and output flows represent data flow between program transformations, we expect some

consistency between the flows entering and exiting corresponding elements, function F and diagram D in this case.

A simple self-consistency check could verify that all flows entering (exiting) F enter (exit) also D and vice versa, i.e., for each flow f_I in I_F thee exists a flow d_I in D_I with the same label of f_I and vice versa, and similarly for for O_f and O_I.

A more sophisticated self check could admit some refinement between F and D, i.e., it would require that all flows in F_I find a flow in D_I with the same label, and that all flows in F_O find a flow in D_O with the same label, but not vice-versa, i.e., the diagram D corresponds to F, but can consume/produce additional data.

2.3. Target times for reminders (e.g., 15 minutes before a meeting, 2 days before a project due-date) might be settable by the user, and/or default target values might be set by the human factors team or a requirements specialist. Additionally, some level of acceptable precision should be set — for example, if the target time is 15 minutes before an event, is 14 minutes acceptable? 22 minutes? When the window of acceptable times is set, we have a testable dependability property in place of the usefulness property.

2.4. Coarse grain locking is farthest out on the "simplified properties" axis. It has little optimistic or pessimistic inaccuracy because it is very simple to check (although it is common for programmers to omit locks because they simply did not anticipate multi-threading problems).

The static check for serializability, without imposing a concurrency control protocol, can be viewed in two ways. If we consider that it is really a check for serializability, then it is at the extreme (no simplification) of the "simplified properties" axis, but it has a great deal of pessimistic inaccuracy. On the other hand, in most cases such a static check will accept programs that obey certain simple syntactic rules, and if those rules are made explicit for the programmer to follow,then we again minimize pessimistic inaccuracy by moving further out on the "simplified properties" axis — but not as far as simple coarse-grain locking. Imposing a particular concurrency control protocol, such as two-phase locking, is identical to this latter possibility: No optimistic inaccuracy, little or no pessimistic inaccuracy, and considerable loss of generality (but less than requiring coarse-grain locking, which can be thought of as the simplest and worst concurrency control protocol).

None of the approaches considered above have any optimistic inaccuracy: If they verify serializable access, we can count on it. Constructing a serializability graph at run-time, on the other hand, is a "perfect" verification only for that particular execution. We might observe thousands or millions of serializable executions during testing, and yet under different execution conditions (e.g., in the user's environment) non-serializable executions might be frequent. Serializability graph testing does not involve property simplification nor pessimistic inaccuracy (it never rejects a serializable execution), but it has the usual optimistic inaccuracy of any method that depends on a finite number of test executions to predict the outcome of future executions.

2.5. The property is a safety property, in fact it is expressed in a negative form ("not to exclude any test that may reveal a possible fault."), and indicates what cannot happen.

Chapter 3

3.1.

1. *Use of an externally readable format also for internal files, when possible.* It is an application of the *visibility* and the *sensitivity* principles to program execution: it makes intermediate results easily observable (visibility), and allows to distinguish executions that differ only for the internal state represented in the internal files (sensitivity)

2. *collect and analyze data about faults revealed and removed from the code.* It is an application of the *feedback* principle, since the collected information can help tune the development and quality process.

3. *separate test and debugging activities.* It is an application of the *partition* principle.

4. *distinguish test case design from execution.* It is an application of the *partition* principle.

5. *produce complete fault reports.* It is an applications of the *feedback* and the *visibility* principles: reports can be analyzed for tuning the development and quality process (feedback), and allow to inspect the results of test execution (process visibility).

6. *use information from test case design to improve requirements and design specifications.* It is an application of the *feedback* principle.

7. *provide interfaces for fully inspecting the internal state of a class.* It is an application of the visibility principle to the program, since it increases the observability of the code.

3.2. Redundancy for fault tolerance is not an application of the redundancy principle. The principle suggests redundant checks to increase the opportunities of reveal possible faults; redundancy for fault tolerance is a mechanism for increasing fault tolerance and as such is an integral part of the solution, and not a redundant aspect added to increase the probability of detecting fault during test and analysis.

3.3. Safety properties exclude the occurrence of critical failures. Such failures are not part of the software requirements and are thus excluded from correct implementations. Safety properties are thus redundant with respect to requirements specifications.

3.4. Visibility can be applied to process in many intuitive ways by making progresses visible: analysis and test reports, complexity measures, progress reports are simple implementations of the visibility principle that, when applied, can simplify the management of the analysis and test process.

Chapter 4

4.1. To be correct, software must behave correctly in all circumstances allowed by its specification. A single incorrect behavior makes the program incorrect, even if that incorrect behavior may never be observed in actual use. For example, suppose a transaction-logging subsystem of an automatic bank teller machine (ATM) works correctly only when each individual withdrawal is less than $5000 USD. It might be 100% reliable in its current environment, because withdrawals are normally limited to much less than $5000 USD, but it could still be incorrect if this limitation is not part of its specification. An implication is that reliability, being relative to use, is also subject to change when the way it is used changes. If a round of currency devaluation makes it necessary to permit individual withdrawals greater than $5000 USD, the previously reliable logging module could become unreliable.

4.2. Reliability measured as the fraction of all packets correctly routed could differ from reliability measured as the fraction of total service time if the behavior of the router is sensitive to traffic, e.g., if it is more likely to drop or misroute packets during periods of heavy load.

4.3. If I am downloading a very large file over a slow modem, I am likely to care more about mean time between failures (MTBF) than availability, particularly if the MTBF falls below the time required to complete the download. For example, availability is the same whether the internet service provider is down for one hour of the day (MTBF $\approx$ 11.5 hours), 2.5 minutes of every hour (MTBF $\approx$ 28 minutes), or 2.5 seconds of every minute (MTBF $\approx$ 28 seconds).

4.4. Since correctness is relative to a specification, a system can be correct but unsafe due to oversights in the specification. Requirements specification, including hazard analysis, plays an even more critical role in safety-critical software than in other kinds of software.

4.5. It is often the case that a system can be made more safe by making it less reliable. In general, this will be the case whenever the system has a "safe state" that is not functional. For example, if an automobile will not start, it is in a safe state. Suppose the automobile performs a self-check of brakes and steering, and will start only if the self-check is satisfactory. Moreover, the self-check itself is not perfect; sometimes a satisfactory result is not obtainable, even though the brakes and steering are fully functional. The car will then be safer, but it will be somewhat less reliable, because on

some occasions it will refuse to start when a car without the self-check functionality would have functioned satisfactorily.

4.6. Responses will vary among students, depending on the application domains with which they are familiar. Some possible answers would include:

- Medical devices: Software safety is paramount, and reliability requirements are very high. Cost and schedule are secondary.
- Productivity software on personal computers: All three factors are roughly in balance, or dependability may be marginally less critical than schedule.

4.7. We can easily refine this specification by giving a concrete time limit, e.g., we might decide that 0.1 seconds is fast enough to consider "instantaneous." However, some users might be quite satisfied with longer latencies, and the simple time bound may make the specification impossible to satisfy in some environments (e.g., over a heavily loaded trans-Atlantic network link). At the same time, while truly instantaneous transmission would guarantee display of messages in the same order they are sent, an implementation that satisfies the latency specification of 0.1 seconds could display nearly-simultaneous messages out of order, and might even jumble portions of near-simultaneous messages on the screen. Thus we may both want to relax the specification (for example by making the latency requirement depend on underlying network conditions) and further constrain it (for example by specifying that messages be displayed in some consistent order to all users).

4.8. Correctness properties refer to expected behavior, e.g., the expected results of formatting commands, such as boldface text as a result of a command that turns text to boldface.

Robustness properties refers to behaviors in unexpected conditions, e.g., specifications of the behavior of the wordprocessor when the file system is full, the memory is exhausted or the user recover from a system error that caused the wordprocessor to crash. We can for instance require the wordprocessor to revert to the last saved version of the open documents.

Safety properties are usually given as negative properties that forbid hazardous states. We can for example forbid modifications to files not opened in the session.

Chapter 5

5.1. Application of model compactness, predictiveness, semantic meaningfulness, and sufficient generality to approaches for modularizing models of programs could include the following:

Compactness: Compactness applies to any representation that must be constructed to use an individual "module" of the design. It is desirable that its size be proportional to the size of the module, and not proportional to the size of the system as

a whole. For example, a context insensitive call graph is compact in this sense, but a context sensitive call graph is not.

Predictiveness: The requirement of predictiveness constrains the approaches we can use to attain compactness. For example, while a context insensitive call graph is more compact, it may be less predictive than a context sensitive call graph.

Semantic meaningfulness: Like predictiveness, semantic meaningfulness is in tension with compactness. If an explanation involves details in several different modules, we will be able to produce it only by considering multiple modules in detail. Thus, to achieve semantic meaningfulness and compactness together, it is necessary to summarize in some way just the "interface behavior" of each module, and to form explanations from these summaries. Designing a truly modular form of analysis or testing inevitably involves adapting or inventing some form of module interface specifications or descriptions. For example, Java analysis of exceptions (enforcing the rule that each method declare the exceptions that it may raise or propagate) can proceed modularly because the only "external" information needed to analyze any method is the list of exceptions that may be thrown by methods at the next level in the call hierarchy.

Sufficiently general: We may need to impose some constraints on systems to achieve compactness together with predictiveness and semantic meaningfulness, but the constraints should not rule out too many of the systems we wish to analyze. For example, constructing a call graph is made considerably easier if we impose some constraints on the way function pointers are used in C and C++. On the other hand, if we ruled out dynamic method invocation, our approach would not be sufficiently general for C++ and Java programs.

5.2. The importance of a carefully defined model is primarily in the specification and design of an analysis technique, and only secondarily in its implementation. For example, it would be difficult to state clearly and unambiguously what the method overloading analysis should do without describing the call graph and class hierarchy. These models may actually exist as data structures, to decouple the first part of the analysis from the second part. They would be no less important, though, if they were a purely mental tool and were never realized as data structures.

5.3. If we consider only analysis of models directly to find program faults, it is hard to find a justification for distinguishing sequential control flow from other control flow in LCSAJs. However, LCSAJs are actually used to guide selection of tests. It is quite plausible that sequential control flow and other control flow differ with respect to difficulty of understanding and the likelihood that some program path will fail. In this sense, the distinction could be justifiable with respect to the predictiveness criterion.

5.4. A simple bound on the number of basic blocks is that it cannot exceed the number of characters in the program (assuming a typical procedural or object-oriented programming language such as Java, C, C++, Perl, Python, etc.). It is difficult to obtain a tighter bound without introducing a much more complicated notion of program size.

5.5.

1. If we consider a directed graph as a representation of a relation, we can never have two distinct edges from one node to another, because a set cannot have duplicate elements.

2. The nodes in a directed graph represent the domain and codomain of a relation. While we informally say that a relation is a set of ordered pairs, some careful definitions say that a relation is a domain set A, a codomain set B, and a set of ordered pairs (a,b) such that $a \in A$ and $b \in B$. Some elements in the domain and codomain sets may not appear in any of the pairs. In this case, the "extra" nodes would not be connected by any edges. Thus we can say that the set of nodes is superfluous only if we limit ourselves to connected graphs, or equivalently that the set of nodes is not superfluous when the graph is disconnected.

5.6. We can still have an abstraction function from concrete states to model states, even if some of the model states are infeasible. There are two possible ways to define it. First, note that a function must map every element of the domain to an element of the codomain, but it does not require that every element of the codomain be accounted for. Therefore we could simply say that the domain of the abstraction function is the set of concrete program states that can really occur, and the infeasible states in the state machine model are not associated with any domain element. A second approach, which is usually more convenient, is to define the abstraction function from a larger domain set of "well-formed" program states to model states. The well-formed states include everything that fit the description of a concrete program state, regardless of whether it can actually occur in any program execution.

5.7. The number of basic blocks can exceed the number of program statements when individual program statements contain internal control flow. There are several cases in which this occurs in Java, C, and C++:

- Short-circuit evaluation of conditions. For example, the single expression

```
( a < 10 || b < 10 || c < 10 || d < 10 )
```

  is broken into four different basic blocks, with the last test (`d < 10`) executed only if all three of the prior tests evaluate to *False*.

- Conditional expressions like `a < 10 ? 20 : 30`, which might be nested to any depth, e.g., `a < 10 ? (b < 10 ? 30 : 20) : 10)`.

- Complex statements like the head of a for loop, which contains a loop initialization expression, a loop continuation test, and a loop increment expression, all of which appear in different basic blocks.

Chapter 6

6.1.

1. Dominance is a forward, all-paths flow problem. It is a forward analysis because we propagate values (dominators) forward to a node from its predecessors. It is an all-paths problem because m dominates n only if *all* paths from the root to n pass through m.

   Since the dominators of a node are other nodes, the tokens that we will propagate through the graph are nodes.

   All paths through node n pass through node n, i.e., each node n dominates itself. Using this fact, we will let the gen set of n be $\{n\}$. The kill sets will be empty.

2. For any non-root node n,

$$Dom(n) = (\bigcap_{m \in pred(n)} Dom(m)) \cup \{n\}$$

3. Since it is an all-paths problem, and the flow equation therefore intersects sets of tokens propagated from predecessors, we must initialize $Dom(n)$ to the set of all nodes. If we started with the empty set, as we would for an any-path problem, the algorithm might converge prematurely and omit some of the dominators.

4. A sample implementation is available from the authors.

5. This may be the trickiest part of the problem, although the solution is actually not difficult. The authors know of no reasonably simple way to define the immediate dominators relation directly using flow equations. However, it is simple to define the set of dominators that are themselves dominated by other dominators:

$$Indirect(n) = \bigcup_{m \in Dom(n)} Dom(m)$$

   The direct dominators can then be obtained simply by removing the indirect dominators from the set of all dominators:

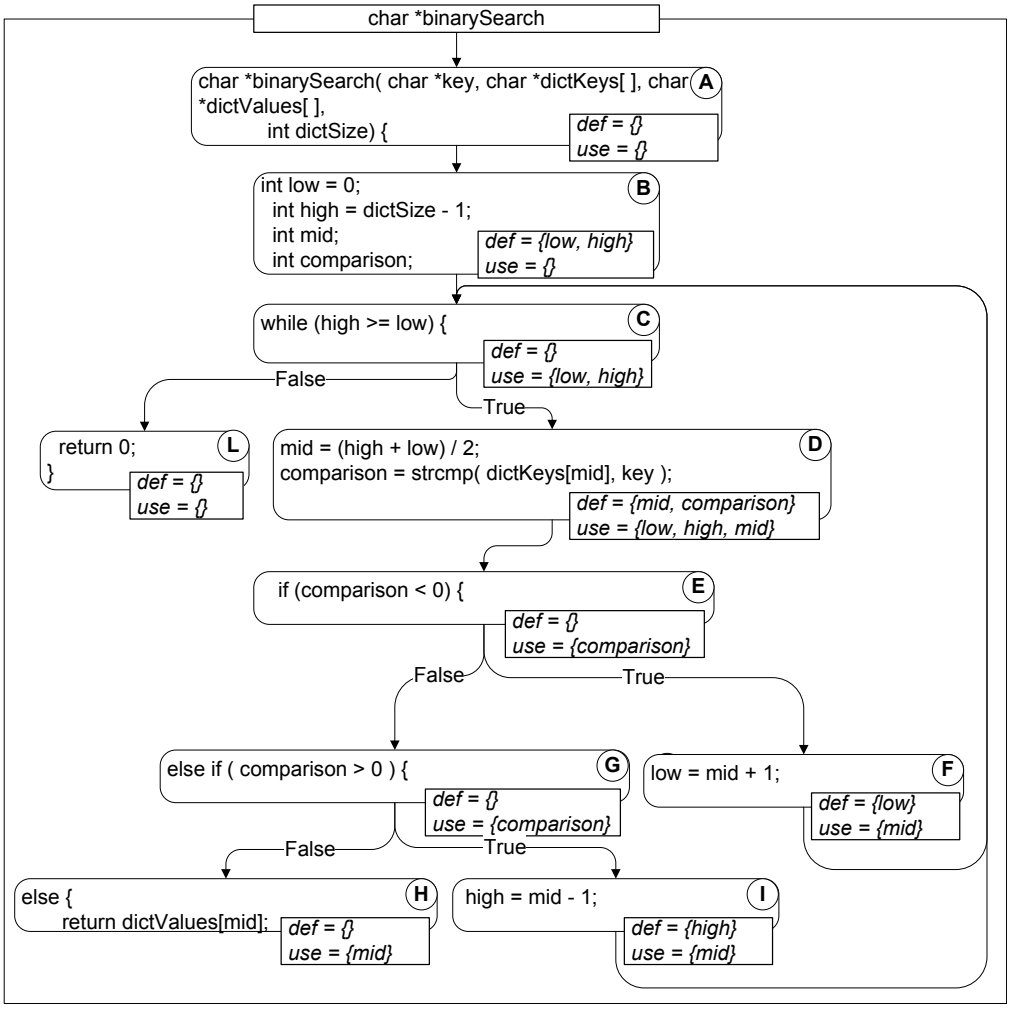
$$DirectDom(n) = Dom(n) \setminus Indirect(n)$$

6.2. Since inevitability is a *backward* analysis, we propagate values to a node from its successors. Since it is an all-paths analysis, we intersect sets of tokens. Thus:

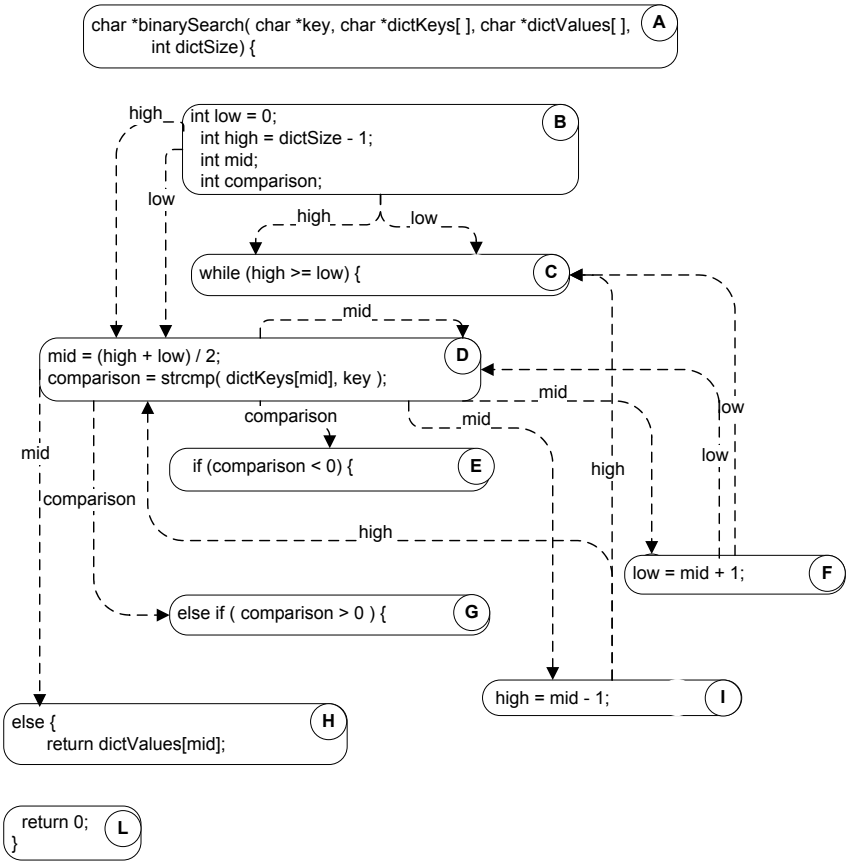
$$\begin{aligned} Inev(n) &= \bigcap_{m \in \text{succ}(n)} InevOut(m) \\ InevOut(n) &= (Inev(n) \setminus kill(n)) \cup Gen(n) \end{aligned}$$

6.3. A local data flow analysis is sufficient to check compliance with the initialization rule for local variables in a method. Determining whether an object field is initialized before it is read, however, would require an inter-procedural analysis to determine the possible order of accesses, whether those accesses are through method calls or direct access to a public field.

6.4. We start by building the program cfg annotated with definitions and uses. The presence of the arrays dictKeys[] and dictValues[] requires some attention. In this program the two arrays, as well as the variable key are used, but never defined within the scope of the program, and can thus be considered constants when analyzing this program:



The data dependence graph can be built straightforwardly by walking through the annotated cfg:



The self-loop on node D describes the data dependence of the second statement from the former one. The two isolated nodes (A and L) do not contain definitions nor uses relevant in the scope of the analysis.

The control dependence is straightforward for all nodes, except node C, which depends from node E.

Chapter 7

7.1. If a variable can be bound to more than one value, symbols are required to represent the value it had at a particular point in execution. For example, if we assign the value of x to y, and then change x, y still holds a representation o the *previous* value of x rather than its new value. If we never change x, then we never need to represent prior values.

7.2. We can observe that the variables key and dictKeys[] are not changed in the program, and that the statement at line 27 is control dependent on the statement at line 17. Thus, if we demonstrate that the condition for traversing the sub-path from statement comparison = strcmp(dictKeys[mid], key); to the return statement requires variable comparison be equal zero (i.e., dictKeys[mid] = key) and that variable mid does not change in the subpath, we demonstrated the property.

Let us start from the statement following comparison = strcmp(dictKeys[mid], key); with with symbolic values M for variable mid and C for variable comparison, and let us symbolically execute the subpath that leads to the return statement.

The execution of the false branch of statement if (comparison $<$ 0) returns the state

$$\begin{aligned} & \text{mid} = M \\ \wedge \quad & \text{comparison} = C \\ \wedge \quad & not(C < 0) \end{aligned}$$

the execution of the false branch of statement if (comparison $>$ 0) returns the state

$$\begin{aligned} & \text{mid} = M \\ \wedge \quad & \text{comparison} = C \\ \wedge \quad & not(C < 0) \\ \wedge \quad & not(C > 0) \end{aligned}$$

which simplifies to

$$\begin{aligned} & \text{mid} = M \\ \wedge \quad & \text{comparison} = C \\ \wedge \quad & C = 0 \end{aligned}$$

which demonstrate that we can go from statement

$$comparison = strcmp(dictKeys[mid], key);$$

to the return statement only when variable comparison = 0. It also shows that the value of variable mid does not change when walking through the subpath.

If we assume that strcmp returns 0 only when the two parameters are equal, we have demonstrated the property.

7.3. The while loop of the binary search program of Figure 7.1 contains 4 simple paths. Two of them reach return statements and thus terminate the execution of the loop. If we demonstrate that the remaining two paths reduce the number of iterations of the loop, we can compute an upper bound to the number of iterations through the loop.

To show that the two paths reduce the amount of iterations in the loop, we symbolically execute each of them. If we execute the two paths as illustrate in the chapter,

starting with symbolic values L and H for variables low and high, we obtain the following states for the two paths, respectively:

$$\begin{aligned} & low = \tfrac{L+H}{2} + 1 \\ \wedge \quad & high = H \\ \wedge \quad & mid = \tfrac{L+H}{2} \end{aligned}$$

and

$$\begin{aligned} & low = L \\ \wedge \quad & high = (\tfrac{L+H}{2}) - 1 \\ \wedge \quad & mid = \tfrac{L+H}{2} \end{aligned}$$

We can thus conclude that each iteration of the loop either terminates (the two paths that lead to a return statement) or reduces the distance between the values of variables low and high by $\frac{1}{2}$ (the paths executed symbolically).

If we consider that the execution condition of the loop depends only from the relative values of variables low and high, we can conclude that the iterations of the loop will terminate when the value of variable low exceeds the value of high, and this happen after a number of iterations sufficient to eliminate the distance between two integer values iteratively dividing it by a factor of 2, i.e., $\log 2(H-L)$.

7.4. We can execute the path starting with symbolic values C, L, H and M for variables comparison, low, high and mid, respectively.

The execution of the false branch of the first if statement leads to the symbolic state:

$$\begin{aligned} & low = L \\ \wedge \quad & high = H \\ \wedge \quad & mid = M \\ \wedge \quad & comparison = C \\ \wedge \quad & not(C < 0) \end{aligned}$$

the execution of the false branch of the second if statement leads to the symbolic state:

$$\begin{aligned} & low = L \\ \wedge \quad & high = H \\ \wedge \quad & mid = M \\ \wedge \quad & comparison = C \\ \wedge \quad & not(C < 0) \\ \wedge \quad & not(C > 0) \end{aligned}$$

the false branch of the third if statement cannot be executed under these conditions, in fact the condition: $comparison = C \wedge not(C < 0) \wedge not(C > 0)$ implies $comparison = 0$ and thus prevents the execution of the false branch, which is infeasible.

7.5. The precondition must indicate that the array is not empty, as required by the specification. If we assume that initial value of setSize indicates the size of the array, we will have the term ($setSize > 0$).

We must also decide if to interpret the word *set* in a strict mathematical sense, i.e., without repeated elements, or not. In the first case, we must add the term:

$$\forall i, j\ 0 \leq i < j < \text{setSize} : \text{set}[i] \neq \text{set}[j]$$

The postcondition must indicate that the result is the index of the maximum value. If we indicate the result with result, the condition can be expressed as:

$$\forall i\ 0 \leq i < \text{setSize} : \text{set}[result] \geq \text{set}[j]$$

notice that the condition of non-empty set is required for the validity of the predicate. Notice also that in case of repeated elements, we do not indicate a privileged index as result.

This term can be expressed also as:

$$\forall i\ 0 \leq i < \text{setSize}, i \neq \text{result} : \text{set}[result] > \text{set}[j]$$

but in this case the condition holds only if the set does not contain repeated elements.

The condition should be completed asserting that the result belongs to the array: $0 \leq result < \text{setSize}$

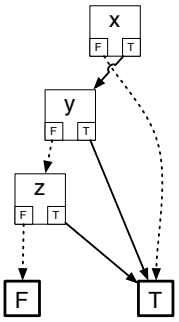
To avoid undesirable side effects, the postcondition should also state that the set is not changed by the method:

$$\forall i\ 0 \leq i < \text{setSize} : \text{set}[i] > \text{set'}[i]$$

Chapter 8

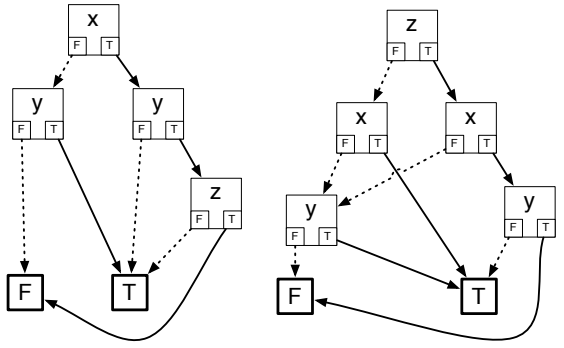
8.1. The statements are consistent. A conservative or safe finite state verification technique really is tantamount to formal proof that a certain model has the verified property when it "succeeds" in the sense of finding no violations of the property. It is less powerful than formal verification, nonetheless, because the properties it can verify are more limited, the models it can verify those properties against are likewise more limited, and it may fail (often producing spurious error reports) in cases where a more expensive formal verification could have succeeded. In terms of Figure 2.2, page 19, such a technique has no optimistic inaccuracy, but it has greater pessimistic inaccuracy (but less cost) than theorem proving.

8.2. The OBDD for $x \Rightarrow y \vee z$ is shown below with variable ordering x, y, z.



8.3.

(a) The OBDD is larger when z is first in the variable ordering than when x or y is first.



Since x and y appear only together with commutative operations (logical *and* and *or*), there is no difference in shape between the OBDD in which x precedes y and the OBDD in which y precedes x; we can just swap variable labels in the OBDD diagram.

(b) In the expression

$$(x \vee y \vee z) \wedge \neg(x \wedge y \wedge z)$$

the variables always appear together in the same combinations with commutative operators (logical *and* and logical *or*), so their ordering cannot matter.

8.4. A safety property asserts that "nothing bad happens," while a liveness property asserts that "something good eventually happens." The real-time property ("within 30 seconds") creates a "bad" thing that can happen, in this case passage of 30 seconds before arrival of the elevator.

For readers familiar with Büchi automata, a more formal and precise characterization follows the same outline but characterizes "good things" and "bad things" in terms of recognizing infinite strings. Violation of a safety property is recognized by

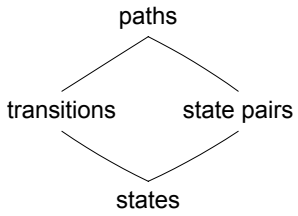
transition to a trap state (once the transition on the "bad thing" is made, no transition to an accepting state is possible). Violation of a liveness property corresponds to an infinite loop through a set of non-accepting states. Linear temporal logic (LTL) model-checking tools, including Spin, typically construct and interpret Büchi automata, and violation of liveness properties are reported as "non-progress cycles."

Chapter 9

9.1. Transition coverage subsumes state coverage, under the (reasonable) assumption that all FSM states are reachable.

Path coverage subsumes transition coverage, under the slightly stronger (but still reasonable) assumption that all states are reachable on paths from the distinguished start state. (A model that violates this assumption is not very useful.)

Path coverage also subsumes state-pair coverage: At least one of the sub-paths from r to s must be covered. However, state-pair coverage does not subsume transition coverage, nor vice versa, as illustrated by the following simple example.[3]



For this FSM, each of the labeled states is reachable from all other labeled states, so there are $3^2 = 9$ state pairs, including pairs like $\langle r,r\rangle$ and $\langle s,s\rangle$. All of these state pairs can be covered with one looping path, so we can satisfy the state-pairs criterion with a test suite containing just one test case:

$$\{\langle t1,t3,t4,t1,t3\rangle\}$$

This test suite does not satisfy the transition coverage criterion, since transition $t2$ is not covered. Therefore state-pair coverage does not subsume transition coverage.

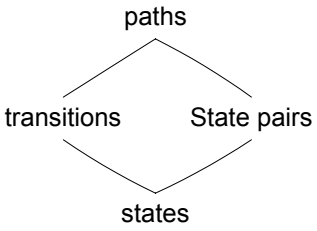
On the other hand, we can satisfy the transition coverage criterion with the following test suite containing just two test cases:

$$\{\langle t2,t4,t2\rangle,\langle t1,t3\rangle\}$$

This second test suite does not cover several of the state-pairs, such as $\langle s,x\rangle$, $\langle x,x\rangle$, and $\langle x,r\rangle$. Therefore transition coverage does not subsume state-pair coverage.

We conclude that the subsumes hierarchy, for these criteria, is

[3] We are using notational conventions as in UML statecharts to mark the initial and final state as if starting and ending were special transitions, but they do not correspond to program inputs and so are not considered in the adequacy criterion.



9.2. The feasibility of logical conditions is undecidable in general, as long as terms in our specification language are at least rich enough to describe integer arithmetic or something equivalent to it. Therefore we cannot escape the infeasibility problem even with adequacy criteria based on specifications.

One might reasonably argue, however, that the practical impact of the infeasibility problem is smaller with logical specifications, because

- A well-crafted logical specification should be smaller and simpler than the program it specifies (though this is not always true in practice).
- A test obligation based on a specification is typically a fairly simple combination of parts of the specification, while the logical conditions under which a program element executes can be extremely complex.

9.3. We cannot be certain that faults revealed by A are also revealed by B. In fact, as the question is stated, it is even possible that A also satisfies the stronger criterion C_2, or an even stronger criterion C_3. Even if we knew that A did not satisfy C_2, it would not follow that all the faults revealed by A would also be revealed by B. A might be an exceptionally good test suite for reasons unrelated to the adequacy criterion, or we might just have been lucky.

Chapter 10

10.1. This aspect of the XP does not contradict the assertion that test cases are a formalization of specifications. On the contrary, in XP the role of test cases as a form of requirements statement is explicit.

10.2.

(a) The fault is revealed if $q = 0 \wedge a = 0$; this will be true only if both a and b are zero. The probability of either being zero is 1 in 2^{32}. The probability of both being zero is the product of the individual probabilities, 1 in $2^{32} \times 2^{32}$, or 1 in 2^{64}.

(b) When a is zero, q must be greater than or equal to zero; therefore the fault will be revealed whenever $a = 0$. The probability of a being zero in a randomly selected set of values is 1 in 2^{32}.

10.3. One potential breakdown is to separate the graphical interface, the textual interface, and the core functionality. It may be difficult to test core functionality apart from one of the interfaces, though. If core functionality is grouped with one of the interfaces, for practical reasons it is better to group it with the textual interface, keeping the graphical interface separate because of the greater expense of producing and initially executing test cases for the graphical interaction. (Capture and replay tools can automate re-execution of a test case involving graphical interaction, but manual operation is typically required the first time a test case involving graphical interaction is executed.)

The core functionality could be further divided into basic arithmetic functions and the memory functions (including with the memory functions checks that the arithmetic functions operate on the the correct values).

Chapter 11

11.1.

(a) If parameter characteristics and value classes are given without constraints, the number of test case specifications is the product of the number of value classes in each category. For example, if the categories are A, B, and C, category A has 5 value classes, category B has 6, and category C has 3, then the number of test cases is $5 \times 3 \times 6$.

(b) If only *error* and *single* clauses are used, we just treat value classes with those constraints separately from unconstrained value classes. The number of test case specifications is the sum of the *error* and *single* classes, plus the product of the numbers of unconstrained value classes in each category. For example, if again the categories are A, B, and C, and A has 5 values classes, 1 of which is an error class, B has 6, 1 of which is an error class and 1 of which is a single class, and C has 3 classes, all unconstrained, then the number of test case specifications is $(1+2+0)+(5-1)\times(6-2)\times 3$.

(c) First, we separate *single* and *error* choices from other parameters. Each *single* or *error* choice contributes exactly one test case (and we do not use them to cover other combinations).

For the remaining enumeration, we consider a set P of parameters (categories) whose remaining choices may have *if* and *if-property* constraints. Let N be the number of parameters, and let $n_1 \ldots n_N$ be the number of (non-error, non-single) choices in each category $P_1 \ldots P_n$.

Our approach is a simple recursive enumeration of possibilities, essentially similar to what we would use to generate a set of test case specifications. For each possible property, we need two flags: One to indicate if the property *must* be a constraint associated with one of the value choices (because it appears in an *if-property* constraint associated with another value choice) and another flag to indicate if the property *has* been indicated in a *property* constraint on a value

choice.[4] We begin with an empty set of constraints, and add to it as we traverse a tree of possibilities:

```
1   /** Count the number of test cases specifications
2    *  that would be generated from N categories, where
3    *  the value choices in each category may have
4    *  "property p" or "ifproperty p" constraints (but
5    *  with no "single" or "error" constraints.
6    */
7   def enumerate( int i, constraints c ) is
8       if i > N then
9           /* Tuple is complete. Is it consistent? */
10          for each "must p" constraint in c do
11              /* Consistent only if another value
12               * had property p
13               */
14              if "has p" not in c then return 0;
15          od;
16          /* If all the "ifproperty p" constraints were matched
17           * by "property p" constraints, then the case is ok.
18           */
19          return 1;
20      fi;
21      /* Still need to fill in choices for categories i..N.
22       * For each choice in category i, we'll count the ways
23       * to fill in categories i+1 .. N.
24       */
25      count = 0;
26      for each choice v in P[i] do
27          childConstraints = c;
28          for each [property p] constraint on v do
29              add "has p" to childConstraints;
30          od;
31          for each [if p] constraint on v do
32              add "must p" to childConstraints;
33          od;
34          count = count + enumerate ( i+1, childConstraints );
35      od;
36      return count;
37  end enumerate;
38
```

[4] Note that we cannot be certain of encountering the categories with *property* constraints before having to select among choices for another category with corresponding *if-property* choices. If we could (e.g., if we accepted only specifications with parameters that could be sorted so that no property is tested before it is set), then only one flag per property would be needed, and the enumerative algorithm could be somewhat faster.

11.2. No, pairwise coverage will never produce a larger test suite than category partition testing, provided properties, constraints, and single and error cases are treated the same in both approaches. Both approaches should generate the same number of error and single cases (though they may choose other values in the single and error test case specifications differently). For cases made up of normal (that is, not error or single) values, every test case specification that can be created by a pairwise combinatorial testing tool is also a combination of values that should be created by the category partition testing tool.

11.3.

(a) The value V5 is not possible because it conflicts with V2. The value V6 is not possible because it conflicts with V3. After eliminating incompatible choices from category C3, there are no choices available.

(b) We should not treat erroneous values as candidates for completing a normal test case specification. The point of testing each pairing of normal values is to check for interactions between particularly pairs, and errror handling is likely to mask any such interaction.

(c) If we marked V7 as *single* because it is (or should be) treated as a special case, then the same argument applies as in part (b): The *single* value is a poor choice to complete a test case specification. If *single* has been specified simply to reduce the size of a generated test suite, then we might be willing to relax this rule. One might also reasonably argue, though, that undisciplined use of *single* is a symptom of sloppy test design, and that it is therefore better to receive a warning about pairs that cannot be covered than to supporess that warning by overriding the *single* constraint.

11.4. Acceptable answers can vary. A good answer should describe parameter characteristics, not concrete values. Note, for example, that in our sample solution, times of the outbound departure are classified by their relation to the inbound arrival, rather than being selected from a set of concrete times. The distinction between parameter values and parameter characteristics may be slight when parameters are orthogonal or nearly so, but becomes important when the specification describes relations among parameters.

Parameter: Arriving flight

Arriving flight code

malformed	[error]
not in database	[error]
valid	

Originating airport code
- malformed [error]
- not in database [error]
- foreign city (relative to transfer airport)
- domestic city (relative to transfer airport)

Scheduled departure time
- syntactically malformed [error]
- out of legal range [error]
- legal

Destination airport (transfer airport)
- malformed [error]
- not in database [error]
- valid city

Note: we are using the destination aiport of the first flight as a base for calling other airports "domestic" or "foreign," so we don't classify the destination airport itself except as to being domestic or foreign.

Scheduled arrival time (t0)
- syntactically malformed [error]
- out of legal range [error]
- legal

Parameter: Departing flight

Departing flight code
- malformed [error]
- not in database [error]
- valid

Originating airport code
- malformed [error]
- not in database [error]
- differs from transfer airport [error]
- same as transfer airport

Scheduled departure time
- syntactically malformed [error]
- out of legal range [error]
- before arriving flight time (t0)
- between t0 and t0 + DCT [if ICT-greater]
- between DCT and t0 + ICT [if ICT-greater]
- more than t0 + max(ICT,DCT)
- equal to t0 + DCT
- equal to t0 + ICT

Destination airport code

malformed	[error]
not in database	[error]
foreign city (relative to transfer airport)	
domestic city (relative to transfer airport)	

Here "foreign" means the two cities (transfer airport and destination airport) have unequal zones, and "domestic" means they have the same zone.

Scheduled arrival time

syntactically malformed	[error]
out of legal range	[error]
legal	

Parameter: Airport connect time record

This parameter refers to the domestic and international connect time record corresponding to the transfer airport.

Domestic connect time (DCT)

not found	[error]
invalid	[error]
valid	

International connect time (ICT)

not found	[error]
invalid	[error]
not allowed	
less than domestic	[single]
same as domestic	
greater than domestic	[property ICT-greater]

11.5. As in the previous exercise, acceptable answers can vary. Function *Airport Connection Check* involves two flights which are composed of two airports, a departing and an arrival time. Here we consider only the inputs used by the function. Other solutions may consider all the details.

New catalog entries The specification refers to universal and connecting time.

Time

[in/out]	valid
[in]	invalid

Step 1: Identify elementary items

Validated preconditions

pre-1 arr_fl.dest.code is valid

pre-2 dep_fl.orig.code is valid

Assumed preconditions

pre-3 arr_fl.arr_time is a valid universal time

pre-4 dep_fl.dep_time is a valid universal time

Postconditions

post-1 if arr_fl.dest.code == dep_fl.orig.code and [(arr_fl.dest.zone == dep_fl.orig.zone imply arr_fl.arr_time - dep_fl.dep_time $\geq$ arr_fl.dest.dom_conn_time) or (arr_fl.dest.zone $\neq$ dep_fl.orig.zone imply arr_fl.arr_time - dep_fl.dep_time $\geq$ arr_fl.dest.int_conn_time)], then val_code == 0

post-2 if arr_fl.arr.code is not in Airport DB, then val_code == 10

post-3 if dep_fl.dep.code is not in Airport DB, then val_code == 10

post-4 if arr_fl.arr.zone == dep_fl.dep.zone and arr_fl.arr_time - dep_fl.Dep_time $<$ arr_fl.dest.dom_conn_time, then val_code == 15

post-5 if arr_fl.arr.zone $\neq$ dep_fl.dep.zone and arr_fl.arr_time - dep_fl.Dep_time $<$ arr_fl.dest.int_conn_time, then val_code == 15

post-6 if arr_fl.arr.code $\neq$ dep_fl.dep.code, then val_code == 16

post-7 if other error, then val_code == 20

Variables

var-1 arr_fl is a flight

var-2 dep_fl is a flight

var-3 val_code is a validity code

Definitions

def-1 flight is a tuple $\langle fl\_id, orig, dest, dep\_time, arr\_time \rangle$

def-2 orig is an airport

def-3 dest is an airport

def-4 dep_time is a universal time

def-5 arr_time is a universal time

def-6 airport is a tuple $\langle code, zone, dom\_conn\_time, int\_conn\_time \rangle$

def-7 code is a string of three alphanumeric characters

def-8 zone is a string or two alphanumeric characters

def-9 dom_conn_time is a connecting time

def-10 int_conn_time is a connecting time

def-11 validity code can be {0, 10, 15, 16, 20}

We do not consider fl_id (the flight identifier), since it is not used by the valid connection function.

Operations none

Step 2: Derive a first set of test case specifications from preconditions, postconditions and definitions

Validated preconditions

TC-pre-1.1 arr_fl.dest.code: valid code

TC-pre-1.2 arr_fl.dest.code: not valid code

TC-pre-2.1 dep_fl.orig.code: valid code

TC-pre-2.2 dep_fl.orig.code: not valid code

Assumed preconditions no test cases

Postconditions

TC-post-1.1 arr_fl.dest.code == dep_fl.orig.code and arr_fl.dest.zone == dep_fl.orig.zone and arr_fl.arr_time - dep_fl.dep_time $\geq$ arr_fl.dest.dom_conn_time

TC-post-1.2 arr_fl.dest.code == dep_fl.orig.code and arr_fl.dest.zone $\neq$ dep_fl.orig.zone and arr_fl.arr_time - dep_fl.dep_time $\geq$ arr_fl.dest.int_conn_time

TC-post-1.3 arr_fl.dest.code $\neq$ dep_fl.orig.code

TC-post-1.4 arr_fl.dest.code == dep_fl.orig.code and arr_fl.dest.zone == dep_fl.orig.zone and arr_fl.arr_time - dep_fl.dep_time $<$ arr_fl.dest.dom_conn_time

TC-post-1.5 arr_fl.dest.code == dep_fl.orig.code and arr_fl.dest.zone $\neq$ dep_fl.orig.zone and arr_fl.arr_time - dep_fl.dep_time $<$ arr_fl.dest.int_conn_time

TC-post-1.6 arr_fl.arr.code $\neq$ dep_fl.dep.code

TC-post-2.1 arr_fl.arr.code not in Airport DB

TC-post-3.1 dep_fl.dep.code not in Airport DB

TC-post-7.1 other error

We do not explicitly add redundant test cases

Definitions no test cases

STEP 3: Complete the test case specifications using catalogs We use the catalog from Table 11.7 augmented with entry time.

Boolean no test cases

Enumeration it applies to def-11 (validity code), but value outside the enumerated set does not apply, since validity code is only an output value and all other test cases are redundant

Range $L \ldots U$ no test cases

Numeric Constant C no test cases

Non-Numeric Constant C no test cases

Sequence no test cases

Scan with action on elements P no test cases

Time

TC-pre-3.1 arr_fl.arr_time: invalid time

TC-pre-4.1 dep_fl.dep_time: invalid time

Chapter 12

12.1. The compound condition adequacy criterion and the modified condition adequacy criterion would both be satisfied by the following set of test cases:

Test Case	n < max_size	c = getc(yyin) != EOF	c != '\n'
(1)	*False*	–	–
(2)	*True*	*False*	–
(3)	*True*	*True*	*False*
(4)	*True*	*True*	*True*

12.2.

(a) The following set of test case specifications satisfies the MC/DC criterion for the statement. Underlined values indicate the values covered by each test case, i.e., the values that modify the decision outcome if changed.

	Room	Open	Close	Bar	Decision
(1)	*True*	*True*	*False*	*False*	*True*
(2)	*True*	*False*	*True*	*False*	*True*
(3)	*True*	*False*	*False*	*True*	*True*
(4)	*False*	*True*	*False*	*False*	*False*
(5)	*True*	*False*	*False*	*False*	*False*

Notice that test case specification (4) requires a combination of values for terms Open, Close and Bar that produces a *True* result. We can chose combinations where only Close is *True*, or only Bar is *True*. Other combinations may be possible when considering the expressions as Boolean combinations of uninterpreted terms, but are impossible when we consider the semantic interpretation of those terms (see next item).

(b) The compound condition coverage would require all $2^4 = 16$ combinations of Boolean values, some of which are not compatible with the semantics of the terms occurring in the condition. In particular all combinations where more than one of the terms Open, Close and Bar is *True* are impossible, since parseArray[pos] cannot be '}' and '{' and '|' at the same time.

12.3. The number of test cases required to satisfy the modified condition/decision coverage criterion (MC/DC) can be calculated by induction on the number of elementary conditions.

If there is only one element, then the minimum and maximum number of test cases required is 2, one evaluating to *True* and one evaluating to *False*.

Suppose the predicate is $E_1 \vee E_2$ (or, in C notation, E1 || E2). Let N_1 be the number of elementary conditions in E_1, and let N_2 be the number of elementary conditions in E_2. Assume (by the induction hypothesis) the number of test cases required to satisfy the modified condition coverage criterion for E_1 alone is $N_1 + 1$, and similarly $N_2 + 1$ for E_2. Let T_1 be a test suite which would be adequate for the predicate E_1 alone, and let T_2 be a test suite which would be adequate for the predicate E_2 alone, and let T_1 and T_2 each be as small as possible, i.e., $N_1 + 1$ and $N_2 + 1$ test cases respectively. Now we will construct a satisfactory test suite T for E by combining T_1 and T_2.

At least one of the test cases in T_1 causes E_1 to evaluate to *True*, and at least one of the test cases in T_1 causes E_1 to evaluate to *False*; T_2 likewise contains test cases that evaluate to *True* and *False* for E_2, else it would not demonstrate the effect of the elementary conditions.

For each test case in T_1, we construct a corresponding test case in T by combining it with a case from T_2 that causes E_2 evaluate to *False*. The overall truth value of E for each of these constructed test cases is the same as the value of E_1 for the corresponding test case, and pairs of test cases in T_1 that showed the effective of elementary condition e in E_1 correspond to pairs of test cases that show the effectiveness of e in E. Note that we can always use the same test case from T_2 to extend each of the cases in T_1; we never have to extend some test case in more than one way. We similarly extend the test cases T_2 for E, using some test case in T_1 that causes E_1 to evaluate to *False*.

The total number of test cases produced by extending cases in T_1 and T_2 is $N_1 + N_2 + 2$. However, note that one test case, which evaluates to *False* for both E_1 and E_2, has been introduced twice. Removing this redundant test case, the number of test cases required to satisfy modified condition coverage for E is $N_1 + N_2 + 1 = N + 1$.

Test cases for $E_1 \wedge E_2$ can be constructed similarly, except that we extend T_1 and T_2 with partial truth-value assignments that evaluate to *True* rather than false.

12.4. Splitting a single node n into a series of nodes and edges $n_1 e_1 n_2 e_2 n_3 \ldots n_k$ adds k nodes and k edges to the graph, leaving the total $e - n + 2$ unchanged. Therefore the cyclomatic complexity is the same whether the nodes are basic blocks or individual statements. This is intuitively consistent with the idea of using the cyclomatic number as a kind of "complexity" measure for program source code.

12.5. The two call graph coverage criteria described in this chapter (procedure entry and exit testing and procedure call testing) are not ordered by the subsume relation. Procedure entry and exit testing requires us to exercise all entry and exit points of the procedure, while procedure call testing requires us to exercise all entry points of the procedure in all the actual contexts in which the procedure is used. Let us consider a procedure P with two return statements R_1 and R_2, and suppose P is called from two different points within Q. Some test suites might exercise all entry points in all actual contexts without exercising all exit points (P is called from both places in Q, but always returns throuugh R_1), thus satisfying procedure call testing, but not procedure entry and exit testing. Other test suites can exercise all entry and exit points of the procedure without calling the procedure in all contexts (all calls from the same point in Q, with some returning through R_1 and some returning through R_2), thus satisfying

procedure entry and exit testing, but not procedure call testing.

Procedure call testing subsumes procedure entry and exit testing for programs that do not have multiple exit points, since exercising all entry points in all contexts, exercises all entry points and the only exit point.

12.6. The MC/DC criterion will demand an infeasible test case when there is a basic condition that is always *True* or always *False* when evaluated in a larger conditional expression. In many cases, one might reasonably argue that such an expression ought to be simplified, omitting the useless part. However, reasonable defensive programming practices may also lead one to use conditions that *should* always evaluate to the same value. For example, we might be able to reason from program logic and from knowledge of the problem domain that a door is closed, and yet choose to test that it is closed before engaging a motor to move the vehicle, as defense against a future change elsewhere in the program or in its environment.

Chapter 13

13.1.

(a) **all p-use some c-use:** A test suite T for a program P satisfies the all p-use some c-use adequacy criterion iff, for each (def, p-use) pair dpu of P, at least one test case in T exercises dpu, and for each set of (def, c-use) pairs dcu with the same definition as first element, at least one test case in T exercises dcu.

all c-use some p-use: A test suite T for a program P satisfies the all c-use some p-use adequacy criterion iff, for each (def, c-use) pair dcu of P, at least one test case in T exercises dcu, and for each set of (def, p-use) pairs dpu with the same definition as first element, at least one test case in T exercises dpu.

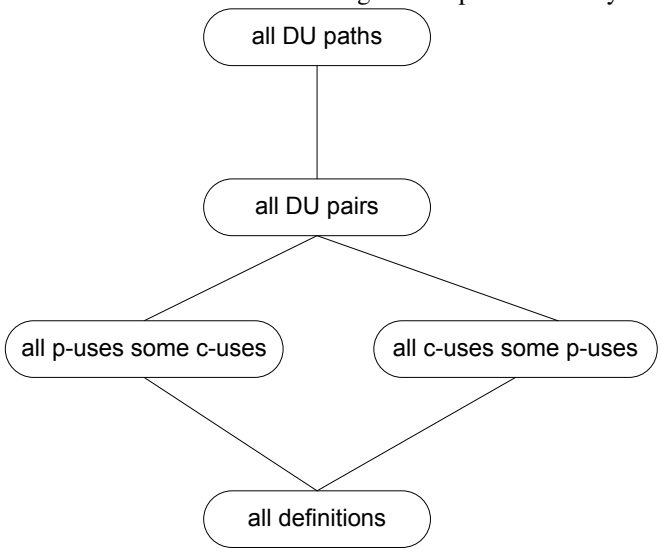
(b) The two criteria do not differ with respect to covering def-use pairs involving variables encoded, decoded, dptr*, digit_high, or digit_low, since for each definition of those variables, there is at most one c-use and one p-use (see Table 13.2).

Variable ok is used only in computational statements (c-uses), while variable c is used only in predicates (p-uses). Thus, the all-c-use some-p-use criterion will require the same elements as the all-def-use-pairs criterion with respect to ok, while the all-p-use, some-c-use criterion will permit fewer pairs to be covered. The converse holds for variable c, with the all-p-use, some-c-use criterion requiring all def-use pairs and the all-c-use, some-p-use criterion permitting smaller test suites.

Variables eptr and *eptr are used in multiple computational statements and each is used in two predicates. The all c-uses some p-uses criterion requires covering all but one def-use pair for each of the two variables, since only one of the two p-uses must be covered. The all p-uses, some c-uses criterion requires covering

at least three pairs for each variable: the two pairs that include a p-use and at least one of the pairs that include a c-use.

13.2. All p-use some c-use, all c-use some p-use, all DU pairs, all DU paths and all definitions are related in the following subsumption hierarchy.



Both "all p-uses some c-uses" and "all c-uses some p-uses" criteria subsume "all definitions," because both criteria include at least one DU pair for each definition. Strictly speaking, the subsume relation does not hold for definitions that do not belong to any DU pair. However, such definitions represent anomalous situations (usually faults) that can be easily identified and removed statically.

The "all p-uses some c-uses" and "all c-uses some p-uses" criteria select different subsets of DU pairs and thus neither subsumes the other, as shown in the former exercise.

The "all p-uses some c-uses" and "all c-uses some p-uses" criteria select subsets of DU pairs, so they are subsumed by the The "all DU pairs" criterion, which covers all of them.

The "all DU paths" criterion covers all simple DU paths, and thus includes all DU pairs, but a test suite satisfying the "all DU pairs" criterion may not satisfy "all DU paths" when there is more than one simple path from a definition to a use.

13.3. It would be too expensive to treat each element of the buf data structure as a separate variable for definitions and uses, so (as with most composite data structures) we will treat it as a single object.

One approach would be to consider definitions and uses in terms of a logical "value" consisting of pos and buf together, even though they are not grouped as a C language struct (the closest C equivalent of records or objects in other languages). It is clear that reading or writing either of these two physical variables is essentially a read or write of this logical value, e.g., setting pos to zero makes the effective content of buf

be the empty string, even though no characters are changed. We would treat buf[pos++] = inChar as both a use and a definition, since it is creating a new logical value from the old logical value. In this program, treating buf and pos together in this way for data flow testing will have the same effect as data flow testing with respect to pos alone.

If we treat pos and buf as separate variables, the emit function is a use, and the statement buf[pos++] = inChar can be treated as both a use and a definition, or as a definition that does not "kill" (in the dataflow analysis sense) preceding definitions.

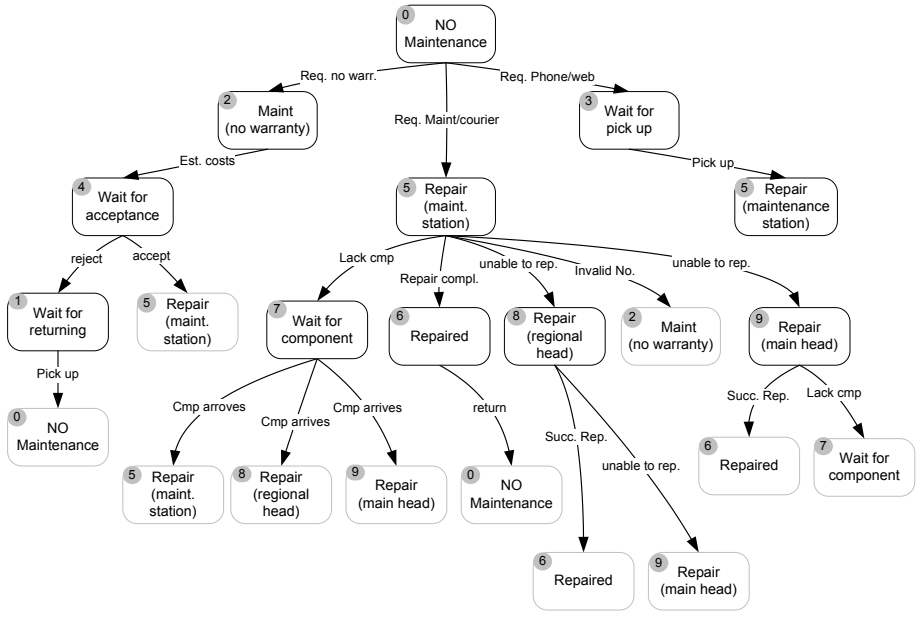
Chapter 14

14.1. Test cases for the FSM specification of Figure 14.2.

(a) Transition Coverage The test cases must collectively *traverse each transition at least once*. The following set of test cases satisfy the criterion (States numbers refer to Figure 14.2)

TC-t1	0 – 2 – 4 – 1 – 0
TC-t2	0 – 2 – 4 – 5 – 6 – 0
TC-t3	0 – 5 – 7 – 9 – 7 – 8 – 6 – 0
TC-t4	0 – 3 – 5 – 8 – 9 – 6 – 0
TC-t5	0 – 5 – 9 – 6 – 0
TC-t6	0 – 5 – 7 – 5 – 8 – 7 – 9 – 6 –0

(b) Single State Path Coverage The test cases must collectively *include each subpath that traverses states at most once in a path that is exercised*. The following tree includes all subpaths that traverse different states at least once. A set of test cases that contains all subpaths from the root to the leaves of the tree satisfies the criterion.



(c) Single transition path coverage must *include each subpath that traverses transitions at most once.* A subpath that traverses both transitions $\langle 5,7\rangle$ and $\langle 7,5\rangle$ need not be included in a test suite satisfying single state path coverage, since it reaches the same states twice, but such a subpath must be included in a test suite satisfying single transition path coverage. One possible test case covering such a subpath is TC-t7:

TC-t7 0 – 5 – 7 – 5 – 7 – 9 – 7 – 9 – 6 – 0

(d) The *Boundary Interior Loop Coverage* criterion *exercises each distinct loop of the state machine the minimum, an intermediate, and the maximum or a large number of times*. This should include covering the loop $\langle 5,7,5\rangle$ an intermediate number of times, which can be accomplished with the test case TC-t8:

TC-t8 0 – 5 – 7 – 5 – 7 – 5 – 7 – 9 – 6 – 0

14.2.

(a) Assuming that implicit transitions are error transitions is equivalent to adding to each state of the FSM of Figure 14.2, page 248, the transitions not already exiting that state. Each added transition leads to the the same new *error* state. To satisfy the *Transition Coverage* criterion, we need to add a new test case for each new transition.

(b) If we assume that implicit transitions are self transitions, we can satisfy the *Transition Coverage* criterion by extending test cases already in the suite with self transitions. With this approach, we do not need to add new test cases, since self transitions do not lead to a terminal *e*rror state, but leave the state unchanged.

14.3.

(a) A sequence of inputs and observable actions for one version of the finite state machine should correspond to a sequence of inputs and observable actions for the other. State and transition labels may differ, but we can simply trace equivalent paths by considering the event sequence. This procedure should always work if both versions of the state machine are deterministic. If one or both versions are nondeterministic, a test case for one version may not map uniquely to a test case in the other, although there should be at least one corresponding test case.

(b) A test suite that exercises all transitions in a version of the test suite with fewer variables and more states will also cover all the transitions in the more compact state machine description with variables. The transformation in part (a) induces a mapping from states and transitions in the former model to states and transitions in the latter, more compact model, and this mapping is onto (surjective).

(c) If we map test cases the other direction, from the more compact state machine model to the machine in which more states are distinguished, a test suite that satisfies the transition coverage criterion for the former may be transformed to a test suite that does not satisfy the transition coverage criterion for the latter. For example, consider this test suite for the compact FSM:

Case	state-(input)-state ...
Tc-1	*p* –(f)– *p* –(a)– *p* –(c)– *l* –(a)– *p* –(c)– *l* –(f)– *p* –(e)– *d*
Tc-2	*p* –(c)– *l* –(e)– *d*
	Key: (f) = LF, (c) = CR, (e) = EOF, (a) = other character

This test suite never makes a transition from state l to state d with empty $=$ *True*, i.e., it never exercises a scenario in which the file does not end with a carriage return or line feed. The corresponding test suite for the original state machine would not satisfy the transition coverage criterion, because it would not exercise the transition from state w to state d.

(d) Combining FSM coverage with MC/DC will generate test suites that do not depend on the number of explicit states. For example, in the case of transition EOF from Gathering to Done, MC/DC requires covering both *True* and *False* outcome of predicate not empty, thus covering the cases represented with two transitions in the original FSM. For this simple machine, the same effect could be obtained with a test suite that covers branches; need and usefulness of more complex structural coverage criteria like MC/DC depends on the way extra variables are used in state transitions.

Chapter 15

15.1.

(a) The *transition coverage* criterion for the Statechart of Table 15.1 can be satisfied by test suites with a single test case that traverses all transitions, for example:

Display Mode
full-graphics
text-only
limited-bandwidth

Test Case TC_X

select_model(M1)	
deselect_model()	
select_model(M2)	
add_component(S1,C1)	*from workingConfiguration to workingConfiguration*
remove_component(S1)	*from workingConfiguration to workingConfiguration*
isLegalConfiguration()	
add_component(S2,C2)	*from workingConfiguration to validConfiguration*
isLegalConfiguration()	
add_component(S1,C3)	*from validConfiguration to validConfiguration*
remove_component(S1)	*from validConfiguration to validConfiguration*
add_component(S3,C4)	*from validConfiguration to workingConfiguration*
add_component(S1,C5)	*from workingConfiguration to validConfiguration*
remove_component(S2)	*from validConfiguration to workingConfiguration*
add_component(S2,C6)	*from workingConfiguration to validConfiguration*
deselect_model()	
select_model(M2)	
add_component(S2,C2)	*from workingConfiguration to validConfiguration*
isLegalConfiguration()	

Notice that the structure of the test case depends on the test environment, which

determines whether a configuration is valid. The annotations indicate the assumptions for the selected case.

(b) The smaller test suite executes faster than the larger one, but the larger test suite provides additional information to localize faults, and reduces the likelihood of masking faults. A failure of the only test case indicates a fault somewhere in the code, while a failure of a simple test case, e.g., TC_A of Table 15.1, indicates that the fault is probably located in one of the three methods called by the test case. Moreover, a failure of a method called in a specific state may be masked by following calls in the big test case.

(c) We can transform the test suite in Table 15.1 into a suite that satisfies the the *simple transition coverage* but not the the *transition coverage* criterion, by substituting test case TC_B with:

Test Case $TC_{B'}$
selectModel(M2)
addComponent(S1,C1)
addComponent(S2,C2)
removeComponent(S1)
isLegalConfiguration()

obtained from TC_B by removing the first two calls to *selectModel(M1)* and *deselectModel()*. The new test suite covers all transitions covered by the test suite in Table 15.1, except for transition *deselectModel()* from *workingConfiguration* to *noModelSelected*, which was covered only by the initial calls of test case TC_B.

15.2. To cover "error" transitions, we need to add test cases that cover calls of methods deselectModel(), addComponent(), removeComponent() and isLegalConfiguration() in state noModelSelected, and calls to method selectModel() in both states workingConfiguration and legalConfiguration, e.g. by adding the test cases:

Test Case TC_{Err1}
addComponent(S1,C1)

Test Case TC_{Err2}
removeComponent(S1)

Test Case $TCErr3$
isLegalConfiguration()

Test Case TC_{Err4}
deSelectModel(M1)

Test Case TC_{Err5}
selectModel(M1)
selectModel(M2)

Test Case TC_{Err6}
selectModel(M1)
addComponent(S1,C1)
selectModel(M2)

Since these test cases exercise "error" transitions, we expect that the executions will reach a final error state and stop. When considering the omitted transitions as "self" transitions, they will not necessarily lead to a terminal state, and thus we need to complete the call with a sequence that leads to a terminal state. We can complete the set of test cases of Table 15.1 with the following cases:

Test Case TC_{Self1}
addComponent(S1,C1)
selectModel(M1)
addComponent(S2,C2)
isLegalConfiguration()

Test Case TC_{Self2}
removeComponent(S1)
selectModel(M1)
addComponent(S1,C1)
addComponent(S2,C2)
isLegalConfiguration()

Test Case $TCSelf3$
isLegalConfiguration()
selectModel(M1)
addComponent(S1,C1)
addComponent(S2,C2)
isLegalConfiguration()

Test Case TC_{Self4}
deSelectModel(M1)
selectModel(M1)
addComponent(S1,C1)
addComponent(S2,C2)
isLegalConfiguration()

Test Case TC_{Err5}
selectModel(M1)
selectModel(M2)
addComponent(S1,C1)
addComponent(S2,C2)
isLegalConfiguration()

Test Case TC_{Err6}
selectModel(M1)
addComponent(S1,C1)
selectModel(M2)
addComponent(S1,C1)
addComponent(S2,C2)
isLegalConfiguration()

15.3. Here we have an equivalent and non-equivalent scenario for each of the test cases in Table 15.1.

Test Case $TC_A^{=}$
selectModel(M1)
addComponent(S1,C1)
addComponent(S2,C2)
addComponent(S3,C3)
removeComponent(S3)
isLegalConfiguration()

Test Case $TC_A^{\neq}$
selectModel(M1)
addComponent(S1,C1)
addComponent(S2,C2)
addComponent(S3,C3)
isLegalConfiguration()

Test Case $TC_B^{=}$
selectModel(M2)
addComponent(S2,C2)
isLegalConfiguration()

Test Case $TC_B^{\neq}$
selectModel(M1)
addComponent(S1,C1)
addComponent(S2,C2)
removeComponent(S1)
isLegalConfiguration()

Test Case $TC_C^{=}$
selectModel(M1)
addComponent(S1,C2)
isLegalConfiguration()

Test Case $TC_C^{\neq}$
selectModel(M1)
addComponent(S1,C1)
addComponent(S1,C2)
removeComponent(S1)
isLegalConfiguration()

Test Case $TC_D^{=}$
selectModel(M1)
addComponent(S1,C1)
isLegalConfiguration()

Test Case $TC_D^{\neq}$
selectModel(M1)
addComponent(S1,C1)
addComponent(S2,C2)
addComponent(S3,C3)
isLegalConfiguration()

Test Case $TC_E^{=}$
selectModel(M1)
addComponent(S1,C1)
addComponent(S3,C3)
addComponent(S2,C4)
isLegalConfiguration()

Test Case $TC_E^{\neq}$
selectModel(M1)
addComponent(S1,C1)
addComponent(S2,C2)
addComponent(S3,C3)
removeComponent(S2)
isLegalConfiguration()

15.4. If we are using the equivalent scenarios approach to test a hash table, among the properties that we will test is that, for distinct keys j and k, the state of the table does not depend on whether we first add j with value v and then k with value w, or first add k with value w and then j with value v. Although we might devise scenarios to check equivalence of behavior without making inspecting the abstract state, it is convenient to be able to write a pair of test cases like the following:

Test Case $order_1$	Test Case $order_2$
T1 = HTable.new()	T1 = HTable.new()
T1.add("k1", "v")	T1.add("k2", "w")
T1.add("k2", "w")	T1.add("k1", "v")

The equivalence of the two scenarios could be established by calling an equal method that compares the canonical representation of T1 at the end of each test case.

Chapter 16

16.1. Weak mutation "kills" a mutant when a difference between the state of the original and mutated program is detected, whether or not this difference would eventually show up in program output. Strong mutation succeeds in "killing" a mutation only when program output is different. Therefore, a test case that kills this mutant under weak mutation but not under strong mutation is one that causes some differences in internal state but eventually produces the same output as the original program.

In the example program, mutation of the first of three if statements has made it identical in effect to the third if statement. A test case that *should* return *True* from the third if statement, but instead returns *True* from the first if statement, will therefore kill the mutant under weak mutation but not under strong mutation. If s1 is "cat" and s2 is "caat" we will obtain this outcome, because we cannot observe from the output that the program correctly judged that "cat" is within edit distance 1 of "caat" by incorrectly concluding that "cat" is transformed to "caat" by substituting one letter for another.

16.2. Weak mutation kills a mutant when it detects inconsistencies in the program state. The corrupted state invalidates the execution that follows the corruption. To continue to execute safely, we would need to save a consistent state and resume the execution from it.

16.3. Mutation analysis works with small syntactic differences, which mimic the effects of small design and implementation errors. The competent programmer hypothesis states that these are reasonably representative of actual faults found in programs. Rejection of the competent programmer hypothesis would mean that these syntactic variants are not sufficiently similar to faults produced by programmers. The coupling effect hypothesis states that a test suite thorough enough to detect all or most of one set of faults is also thorough enough to detect all or most of another set of faults; in particular, a test suite thorough enough to detect all or most faults introduced by program mutation is also thorough enough to detect all or most real program faults introduced by programmers. Mutation analysis could still make sense to the extent that either of these two hypotheses held, or each held to some extent. If both were utterly false — if syntactic mutants were utterly unrepresentative of real faults, and effective detection of these unrepresentative faults indicated nothing about effective detection of real faults — then the approach would be without merit.

16.4.

Invalid: Mutants that are not syntactically correct. For example, we can apply the operator sdl (statement deletion) to remove the declaration int pos = 0, making uses of of pos syntactically incorrect.

Valid but not useful: Mutants whose behavior differs in a large fraction of possible executions, so that they are killed almost immmediately even with very weak test suites. For example the mutant obtained by applying the operator aor (arithmetic operator replacement) to transform the statement pos++, into pos−−, or the mutant obtained by applying the operator ror (relational operator replacement) to transform statement pos>0 into pos<0, will not be useful in gauging or improving thoroughness of the test suite.

Valid and equivalent: Mutants that cannot be distinguished from the original program by any execution. For example, we can apply the operator abs (absolute value insertion) to replace pos (which is always non-negative) with abs(pos), so that if (pos $>$ 0) becomes if (abs(pos) $>$ 0).

Valid and non-equivalent: Useful mutants, which are valid and not equivalent to the original program, have behaviors that differ from the original program in only a small fraction of possible executions. For example the mutant obtained by applying the operator sdl (statement deletion) to remove the statement if (pos $>=$ BUFLEN−2) fail("Buffer overflow"); will fail in where the buffer overflow check should be triggered, and is therefore useful in determining whether the test suite includes this important case.

Chapter 17

17.1. Two possible approaches, each appropriate in some circumstances, are fingerprints and on-line comparison.

The fingerprint approach stores a value computed from the output, rather than the output itself. Cryptographic hash functions such as MD5, for example, produce a compact and seemingly random signature corresponding to an arbitrary length text block, such as a suitable text representation of the output of a trusted implementation. If the same signature is obtained by applying the cryptographic hash function to the program under test, with very high probability the complete outputs are identical. It does not matter whether the hash function is secure in the sense that it is difficult for an adversary to purposely produce a hash collision, so cheap and common cryptographic hashes are apt to be perfectly adequate even if they are obsolete for applications in computer security. A weakness of this approach is that a test failure (a difference between computed hashes) provides very little diagnostic information. It does not, for example, indicate whether the difference between predicted and observed output is small or large, or whether it appears near the beginning of the output or near the end.

An on-line comparison oracle does not store the output of the trusted alternate version, but rather runs the trusted version in tandem with the version under test, incrementally comparing output as it is produced. All but a small portion of the output can

be discarded as soon as it has been used in the comparison. On-line comparison oracles avoid some of the shortcomings of fingerprints, and in particular they can provide better diagnostic information from a failed test case. However, they are considerably more complex to implement, and require more execution resources (which is seldom a property if the only resources required are memory and computation time, but becomes more problematic if the necessary resources include an external device such as a robotic arm).

17.2. The "test first" approach associated with extreme programming aligns test oracles with program specification by using test cases (which include pass/fail criteria) as a form of specification. In this sense it meets our stated goal of unifying specification and test oracle. However, it bridges the gap in expressiveness by imposing the limitations of run-time checking on the specification statement. Thus the underlying problem of tension between expressiveness and efficient checking has not really gone away, but its cost has been moved from a gap between specification and oracle to a gap between the specification we would prefer to state and the specification we can express through test cases.

17.3. Advantages of off-line checks may include:

- A computationally expensive analysis may be performed with limited perturbation of the running program.
- The same event log may be used for several different checks.
- Because logs may be retained between runs, it is possible to design a test case that involves more than one execution, e.g., comparing logs produced under different circumstances.
- Some problems of on-line self-checks (e.g., retention of "before" values) become trivial when analyzing a log file.

Advantages of on-line self-checks over off-line log analysis may include:

- Direct access to program state is sometimes much simpler and more efficient than recording an adequate log record. For example, recording the state of a collection data structure after each operation is likely to be much more expensive than checking a structural invariant during execution.
- Production of a log record is essentially a side effect of program execution. Log recording may itself cause program failure, for example when available storage for log retention is exhausted. If logging is activated for testing and disabled in the production environment, we have introduced a potentially significant difference between the tested program and the program in use.

Chapter 18

18.1. Local face-to-face meetings are easier to organize. Asynchronous "meetings" (really an exchange of messages) are less dynamic and sometimes less efficient than synchronous communication, but opportunity for synchronous meetings at a distance (teleconferencing) may be limited by time zone differences (e.g., 8am in Eugene is 5pm in Milan) and are more dependent than local meetings on communication technology (which may range from instant messaging or telephone to video conferencing). Thus, distributed inspection requires careful organization of synchronous and asynchronous work to use meeting opportunities and support effectively.

Distributed meetings are also more taxing and than face-to-face meetings, and less effective at building and maintaining personal relationships among team members, though the gap is likely to narrow as communication technology support improves. Even the best technology support is unlikely to completely erase the difference; for example, cookies brought in by one team member cannot be effectively shared through an audio or video link.

On the other hand, local inspection can take advantage only of the personnel at a single location, while the most appropriate personnel may be distributed across locations. A local inspection meeting in Eugene would not be able to include experts in device interfacing situated in Milan, and the Milan team would be without the mapping and geographic information system expertise of their Eugene teammates. This is particularly apt to be problematic for inspection of requirements and overall design documents, which are typically wider in scope than detailed design and code. And while a face-to-face local meeting may be more effective than meeting at a distance for team-building, the need for building and maintaining camaraderie may be greater between members across sites, e.g., between test designers in Eugene and Milan.

Because both local and distributed forms of meetings have value and limitations, a hybrid approach may be employed. Simple inspection sessions may be carried out asynchronously by individual inspectors without regard to location, with local local face-to-face meetings for checking complex properties when the necessary expertise is available locally, and distributed asynchronous work and synchronous meetings for the checks that involve experts spread across locations.

18.2.

1. A useful study would focus on the ongoing cost of inspection. Start-up costs of designing the inspection process, acquiring tools, constructing initial checklists, training, etc., are less significant if amortized over the long term, and can be ignored in an initial evaluation. The most significant costs by far are human resource (time spent by people on inspections and preparation for inspections) and, in some cases, impact on project schedule (where difficulty scheduling meetings leads to a schedule impact out of proportion to the actual time spent on inspection). Time spent in inspection meetings and on particular inspection activities (e.g., individual reading and review) is an incomplete record of cost, but is not too difficult to measure in most organizations, and is a reasonable choice for a pilot study.

While we may anticipate long term benefits like spread of knowledge through the organization, we should concentrate on the most unambiguous and easily measurable benefits in the pilot study. Faults found in inspection can be counted, and we may characterize them depending on whether and how they would have been detected without inspection. We may look for these effects in the number and kinds of faults found in each stage of testing and in bug reports for a fielded system. We may observe an actual decrease in testing effort, or we may be able to impute a potential decrease in testing effort if the number and seriousness of some class of faults is decreased far enough to justify a reduction in testing for that class of faults. Where inspection at an early stage reduces faults detected at a later stage, we can also infer a benefit in the difference between the cost of early repair and the much higher cost of repairing that fault later.

To quantify benefits of revealing faults that would otherwise remain undiscovered until the product is delivered, we need to take into account both the effort saved in fixing the faults (analogous to the difference between the cost of a fault discovered during unit development and test and a similar fault discovered during system test), and the impact on customers. Assessing potential impact on customers may be difficult to do with much precision and credibility. An alternative is to characterize the cost of detection in some other manner. For example, if inspection reveals some deadlocks that would otherwise go uncaught, we could try to identify the same class of faults with another technique such as model checking, and evaluate the benefit of inspection as the difference in effort between inspection and the other technique.

2. It is notoriously difficult to perform sound studies of human performance in a work setting. Projects are never identical (particularly in software, which does not involve people in the mass duplication portion of manufacturing), individuals and teams are never identical in skills and background, and performance may depend on novelty and practice. With unlimited resources, we might control for individual and project variation by having each of several teams perform each of several projects under different conditions.

   Since we never have the luxury of running each of several real projects several times, we must try to obtain credible comparison within or across projects and control as well as possible for confounding effects, knowing that both approaches have certain risks (or "threats to validity"). These include positive and negative effects of change (the Hawthorne effect, in which novelty leads to expectation and realization of improvement, and learning effects, in which people become better at an activity through practice and familiarity). We can typically do a little better at controlling for these effects across projects than within a single project, where for example the costs and benefits of inspection might vary greatly between the business logic subsystem and the human interface, and the experience of an individual with inspection in one part of the system might later impact that person's performance on another part. On the other hand, the cost of a project-to-project comparison may be higher, and in smaller organizations we may seldom have two sufficiently similar projects, with sufficiently similar teams, running concurrently. A typical approach (also imperfect but sometimes

the best we can do, at least initially) is to compare a pilot project to the record of a similar project in the recent past.

18.3. The cost of inspection can be approximated as the effort required for inspection and meetings. The cost of an analysis tool includes costs of purchasing, personnel, formalization of properties, execution of analysis and interpretation of results. The purchasing cost is the cost of the licences. Personnel cost may derive from the need of hiring people with specific skills. Formalization cost is the effort to express properties as required by the analysis tools. Execution cost is the effort required to execute the analysis. Interpretation cost is the effort required to interpret the results of analysis and identify the related faults.

Some costs may be negligible, depending on the analysis. For example, a simple lockset analyzer described in the next chapter does not require specific skills, the definition of the properties to be checked is built in the tool and interpretation of error messages is easy. Thus relevant costs are purchasing and execution. On the other hand, model checkers ask for specific skills that impact on the costs of the personnel involved in the analysis, require properties to be formalized in a specific way, and results may not be easily interpreted in terms of source code.

18.4. Variations on the following list are possible. A good solution to this exercise should have good justifications for the selected ...

The three classes of tools that can optimize inspection effort are:

Feature Highlighter that scans the checklist and identifies and highlights the part of the artifact that correspond to the features and the items currently inspected. For example, when using the Java checklist in Table 18.4 at page 346, a Feature Highlighter would highlight the header of the files, when considering the entry FILE HEADER of the checklist, the footer of the file when considering the entry FILE FOOTER, etc.

A Feature Highlighter reduces the inspection and meeting overhead, and maintains the focus of the inspectors on the creative aspects of the inspection (check if items are satisfied and not look for items in the artifact to be inspected).

Result Tracker: A *Result Tracker* keeps track of the results of the inspection and relates them to the inspected artifact. For example, referring to the Java checklist in Table 18.4, a Result Tracker Would indicate the parts of the code that violate the items of the checklist, and would link them to the comments of the inspectors.

A Result Tracker help both inspector and developers to quickly locate problems, identify faults and keep track of progresses.

Meeting Manager: A *Meeting Manager* supports meeting organization and execution with features to quickly plan meetings according to the schedules of the participants, automatically distribute information required for meetings, meet synchronously and asynchronously in local or distributed environments.

A Meeting Manager reduces the effort waste in organizing meetings and avoid inconveniences that may derive from distraction.

18.5. We can turn on the attention and participation of inspectors by (1) making them active part of the process and (2) define suitable rewarding mechanisms. Passive participation of inspectors can be avoided by assigning inspector specific (individual) roles according to their expertise, and giving them responsibility for the assigned roles. We can for example identify the expert of some classes of design patterns, who will be in charge of double checking items related to those patterns, and of speaking up when such items occur in the artifact to be inspected. The necessity of triggering the discussion according to the assigned role forces inspectors to pay attention to the whole inspection sessions, and increases active participation in the process. Passive participation of inspectors to meetings can also be discouraged by rewarding mechanisms that for example depend on the contribution of inspectors to fault identification. As often happens, rewarding mechanisms may bring unexpected side-effects in the overall process, and must be carefully evaluated in the specific context.

Chapter 19

19.1. A variety of examples are possible.

If we consider only intraprocedural analysis, then one way that lock(l) ... unlockl might present extra challenges is by involving control flow, especially with correlation of conditions:

```
if (i < 20) {
   lock(l);
}
...
if (i <= 21) {
   x = f(y); // x is variable to be protected
}
...
if (i <= 21) {
   unlock(l);
}
```

Because synchronized blocks are syntactic structures, the locking and unlocking operations they imply cannot be intermixed with conditional control flow in this manner, and it is simple to determine that the lock is held everywhere within the block.

Another problem that arises when lock(l) and unlock(l) are not grouped together syntactically is that they might not appear in the same procedure (method or function). Rather, the lock is acquired in one procedure, and then control is passed to another procedure which is required to release the lock after accessing the object. For example, code that attempts to implement the two-phase locking protocol to ensure serializable transactions may acquire a lock for each data item before accessing it, but defer releasing those locks until a transaction unit is committed.

19.2. A lock is identified with an object, but it is referenced through a variable, so determining whether the *same* lock is always held at a particular program point re-

quires determining whether the lock variable l in a surrounding synchronized(l) { ... } always refers to the same object. This can be arbitrarily hard (i.e., undecidable in general), because the variable binding may be determined through an arbitrarily complex computation:

```
l = f( v, w, x );
synchronized(l) {
   x = g(v, w);
}
```

19.3. The simplest approach may be to provide a user option to revive all previously suppressed error reports. The advantage of simplicity is balanced against the serious disadvantage that this approach is unselective, and lets the user see potentially important error reports only at the cost of wading through many reports that remain irrelevant. Allowing the user to indicate particular error reports that should no longer be suppressed does not solve this dilemma, because the user gains no information if reactivating error reports X, Y, and Z results in seeing error reports X, Y, and Z.

More value may be obtained by letting reactivation be triggered (always, or only optionally) by conditions. For example, when suppressing a particular error report, the user might be given a chance to provide rationale in the form of a program assertion, and the error report might be revived when violations of the assertion are observed. An approach that would place less burden on the user would be to revive suppressed error reports only when surrounding code has been changed in certain ways. Triggering changes could be detected statically (e.g., checking at a statement might be reactivated if another statement on which it is directly control or data dependent has been changed), or dynamic (e.g., using a run-time analysis tool like Daikon to note that a relevant condition that used to be always true is now sometimes violated).

19.4. For four variables, with three possible values each, there are $(2^3)^4 = 4096$ possible tuples of sets of FSM states; each possible value is representable in 12 bits. There are $2^{(3^4)} = 2^{81}$ possible sets of tuples of FSM states; each set is representable in 81 bits, around 7 times as large as the tuples-of-sets representation. Solution times are also likely to be larger for the sets-of-tuples representation, not only because of more work in each step to manipulate the larger representation, but also because the number of computational steps is related to the number of different approximations encountered before reaching a final value, which is likely to be much greater for the sets-of-tuples representation.[5]

The extra cost of the sets-of-tuples representation buys the ability to correlate values of the variables. For example, suppose we were attempting to verify $(w \vee x) \Rightarrow \neg(y \vee z)$. Suppose x is sometimes true and y is sometimes true, but they are never true at the same

[5] When a powerset lattice is used to iteratively find a fixed point solution, the number of steps is proportional to the height of the lattice, which is equal to the number of bits in the representation. We can think of each refinement to the approximation as flipping at least one bit from zero to one; monotonicity guarantees that we never flip bits from one to zero.

time. Analysis using the tuple-of-sets representation might encounter the values

$$\langle \{False\}, \{True\}, \{False\}, \{False\} \rangle$$

at some program point, and later encounter

$$\langle \{False\}, \{False\}, \{True\}, \{False\} \rangle$$

at the same program point, and merge them to

$$\langle \{False\}, \{False, True\}, \{False, True\}, \{False\} \rangle$$

The fact that the *True* values of x and y were encountered at different times is lost, and the analysis inaccurately reports the possibility of x and y being simultaneously true and violating the property. The more expensive sets-of-tuples representation avoids this inaccuracy.

Given these trade-offs, it is clear that we should prefer the tuples-of-sets representation whenever we do not depend on corrrelating values, but may be forced to use the sets-of-tuples representation when we require its precision in correlating values. A practical application arises in lockset analysis, where we sometimes need to know that read accesses to a variable are protected by at least one of several locks, while write accesses are protected by all of those locks.

Chapter 20

20.1. The architectural design will minimize risk to the extent it separates questions about particular printers or printer drivers from questions about the behavior of the software. All printing functionality must be through a small subsystem with very well-defined interfaces, both the interface to other parts of the software (call it the client interface) and interfaces to printers or printer drivers (call it the printer interface). Moreover, those interface definitions must be independent, meaning that one can effectively make a judgment about correct use of the client interface without reference to the printer interface, and vice versa. With this decomposition, construction and testing of the printing subsystem can be scheduled earlier in the project, and testing with actual printers can begin earlier, long before access to the physical printers becomes a resource bottleneck. If the two interfaces are well-enough specified, risk of finding serious problems when the completed software is tested with actual printers should be greatly reduced.

This modularization is generally in line with good software engineering principles, and should align well with the needs of the development manager. For example, the same separation of concerns protects the development manager from risks associated with requirements changes. However, the degree of specification detail and programming discipline required is likely to be more than would otherwise be justified, and some negotiation and compromise may be required.

20.2. Externalizing software development reduces the control over progress, delivery time and quality. This leads to risks of missing deadlines and delivery of poor quality software. We can reduce the impact of these risks by externalizing the development of modules with low impact on critical paths, anticipate test and analysis activities, agree with the external development team on measurable test and analysis activities to be executed before delivery, define acceptance test suites to be executed on the delivered modules, schedule activities to allow for some extra time to recover from unexpected delays.

20.3. The distribution of fault types versus triggers indicates which types of test cases reveal the different classes of faults. Unexpected distribution of fault types vs triggers may derive from bad design of test cases or erroneous classification of test cases. If test cases are badly designed, test designers should add appropriate test cases; if test cases are badly classified, test designers may revise the test documentation, and define an incentive structure to avoid erroneous classifications.

The distribution of fault types versus impact indicates the impact of the different types of faults on the customers. Test designers may revise the plan, so that test cases that reveal fault types with severe impact are executed early.

20.4. Fault qualifier may be correlated with age and defect type. Test designers can analyze age versus qualifier to monitor development progresses: an increasing number of faults due to missing code in base code may for example indicate problems in specifications and design. Test designers may analyze defect type versus qualifier to relate specific development phases to classes of faults: if for example some defect types relate mostly to missing code, they may derive from poor specifications or design.

Chapter 21

21.1. Failures when attempting to read from an empty file may be *violations of value domains or of capacity or size limits*, caused by implicit assumptions about files. Failures when reading a file containing a syntax error could also be *violations of value domains*, but are probably better classified as *missing or misunderstood functionality*, since the most likely cause is wrongly assuming that files are always checked before being read, e.g., when generated.

21.2. Thread integration is preferable when we can easily identify self-contained functionality that includes many modules, for example, maintenance management, configuration management, customer management, etc. Critical module integration testing is preferable when implementation risks are concentrated in particular components or modules. For example, if we use new technology, e.g., a new middleware layer, or if failures may cause severe damage, e.g., in the security kernel.

21.3. If bottom-up portions of the backbone strategy are in places where stubs would be more expensive to build than drivers, and top-down portions are in places where

drivers are more expensive than stubs, then the overall cost of producing scaffolding may be reduced.

Chapter 22

22.1. Test cases that may serve both during final integration and early system testing typically refer to end-to-end behavior checked while integrating subsystems. For example, many test cases of the test suite designed for the integration of the human interface with the business logic of the Chipmunk Web presence may serve both purposes. Such test cases may be be executed with a partially stubbed infrastructure as integration test cases, and with the whole system as system test cases. For example, we might define a test case for the "check configuration" functionality of the "build your own computer" feature, use it to test integration of the user interface, business logic, and database components, and continue using it in early system testing.

Integration-only test cases typically check interactions of internal subsystems that do not correspond directly to end-to-end behavior. For example, we might define an integration test case that involves adding to a computer a component not found in the database, requiring the check configuration function to react robustly to that situation, though the we also ensure with client-side and server-side data validation that the situation can never arise in actual system execution.

System-only test cases check global properties that characterize the system as a whole, but not partial integration of subsystems. This is the case of many nonfunctional properties, e.g., performance properties that depend on the behavior of the whole system. For example, although we would design module and integration test cases related to response time design goals, the ultimate check of overall response time bounds would be from system-only test cases.

22.2. Testing responsibility should typically be shifted from the development to an independent quality team when it becomes necessary avoid a predominance of development over quality issues as time and effort become critical, and pressure on developers mounts. Training issues may also impact the decision: Not all good developers may have good testing skills or vice versa, so dividing responsibility may simplify identification of staff with suitable skills.

Shifting testing responsibility from a development to a quality team becomes cost-effective when test case design does not require deep knowledge of implementation details. This usually happens when moving from unit and early integration testing to subsystem and system testing. Thus it is often wise to shift responsibility when moving to subsystem integration testing. Quality teams may also be responsible for monitoring early testing performed by developers, e.g., by checking unit test coverage and inspecting the results of early testing.

Shifting testing responsibility to a quality team may be impractical when the size of the project or organization is small or the cost of shifting in terms of training on the product and the process is higher than the anticipated benefits.

22.3. Properties that cannot be efficiently verified with system testing include those that refer to users' perception, those that refer to unexpected behaviors, and many internal properties. The first category, user perceptions, includes many usefulness properties such as ease of reaching some information or navigating in the application, as well as some dependability properties, such as mean time between failure and availability, which depend on usage. The second category, unexpected behaviors, includes security properties regarding protection of the system from malicious attacks. The third category, internal properties, includes maintainability, modularity, etc, which are not primarily properties of system behavior and for which testing is therefore not the appropriate assessment.

For a property of user perception (e.g., ease of navigation), we try to establish a set of concrete and testable conditions early with usability prototyping and assessment, preferably performed by a human interface team. Some of those conditions may be verified through inspection (e.g., checks for compliance with standards for meaningful labels and ordering of fields on a screen), and some through automated testing (e.g., bounds on response latency). The actual usability properties would meanwhile be validated in an ongoing series of user assessments, from the earliest interface mock-ups through the actual system.

Dependability properties like mean time to failure and availability are statistical in nature, and must be measured relative to profiles of actual or presumed usage. Random tests, distinct from systematic system tests, are generated from usage profiles and used to derive statistical estimates and confidence.

One can never thoroughly test for response to completely unexpected events, but one can often make useful assessments of likely response to rare and undesired events. Security testing, for example, is always relative to a threat model which gives some characteristics of the nature of a possible attack. A combination of inspection, static analysis, and testing under extreme or unusual conditions can be used. For security, a "red team" may be employed to devise attacks, enriching the threat model as well as directly testing protection.

Internal properties such as modularity and maintainability may be assessed in design and code inspections and, to some extent, with static program analyses (although one must avoid confusing proxy measures such as coupling and module size with the actual properties of interest).

22.4. A typical case of resource limitations that impact system more than module or integration testing is that of some embedded software systems that execute on expensive equipment. If, for example, we consider the software that drives an aircraft flight simulator, we may be able to run most module and integration test cases with a scaffolding of average complexity, but we may need expensive flight simulator hardware to run many system tests, and scheduling of system testing may suffer much more than module or unit testing from limited availability of the hardware platform. Similarly, software than runs on large and expensive networks may suffer from resource limitations during system testing, but not during module or integration testing. This is the case, for example, of the software of a mobile phone network that needs a whole network for final system testing, but whose modules can be tested with suitable scaffolding

and a small or simulated network.

22.5. The property must be checked as part of acceptance and not system testing, since it is refers to qualitative measures that depend on customers' feelings ("perceive", "convenient", "fast", "intuitive").

To check the property as part of system testing, we would need to reformulate it referring to quantitative metrics strongly associated with customers' perception, but that can be measured independently from them. To formulate the property in quantitative terms, we can rely on usability principles that relate measurable characteristics of the interface to perception and reaction time, e.g., by limiting the choices in each page of the Web application, steps required to complete an operation, and response latency.

> Each window of the purchasing application must include no more than ten choices. Choices must be highlighted always with the same style, which must conform to W3C guidelines. Users must be able to purchase products in any legal configuration in less than 5 steps, where a step involves interaction with a single screen layout. Erroneous choices must be signalled explicitly to users and must indicate possible recovery actions. The response time of the application to each user choice must not exceed 100 milliseconds under peak traffic, where peak traffic is simultaneous access of up to 1000 customers.

The reader should notice that the reformulation is not perfectly equivalent to the original property, and do not substitute for a final acceptance testing phase, but allows many checks to be performed earlier at interface design time and during subsystem and system testing, to reduce testing and rework costs.

Chapter 23

23.1. Some of the possible differences in tooling for each of the pairs of projects are rather obvious, but some less obvious differences are also possible.

- Java-only versus mixed language development: The obvious difference is that we may have tools specific to each development language. For example, we might use one testing framework for Java, and another for Python, even if they provided essentially similar functionality. A more interesting difference is in single-language versus multi-language development — integrating analysis across languages is likely to be a larger challenge than providing analysis for each of the languages individually.

- Larger and longer projects require more thorough (and, unavoidably, more bureaucratic) process and documentation than shorter, smaller projects. This is equally true for managing the quality process as for managing development of product code. Tool support for the larger project will almost certainly involve collecting and integrating much more information, in greater detail, than for the

smaller project. Also, the larger project may reasonably make larger investments in tools specialized to that project than would be justifiable for the smaller project.

- Tooling for the information system will be similar to other information systems. The interesting part of the question is tool support for testing the weather simulation. Test oracles for simulation software pose a challenge in distinguishing correct behavior from other, plausible-looking but flawed results. A typical approach to validation is to "predict the past" by running a model of past known behavior. This is particularly appropriate for weather simulation, and we would expect a testing framework that allowed replaying of models with past data and comparison to a record of actual weather in that period.

23.2. One simple approach is to forbid (and scan for) string constants in source code, or string constants of more than n alphabetic characters. A simple script in Awk, Perl, or Lex can be written to perform this check. Since quoted strings might still be needed in some contexts, one could either provide a way to indicate strings that will definitely not appear in messages, or one could partly automate that task with interprocedural data flow analysis to distinguish string constants that definitely will, may, and definitely will not be propagated to user messages.

23.3. Limitations of human memory make us very bad at the sort of mental bookkeeping needed to check non-local properties, so automating that check is almost certainly the better investment. Exceptions are possible, for example if the local property is safety critical and violations of the global property are merely annoying.

Chapter 24

24.1. Agile software development methods prescribe a precise set of test cases that must be executed during development. For instance, the sidebar on page 381 requires executing unit and integration test suites at each development step. We can derive an analysis and test strategy template from the description of our favorite agile method, and instantiate the template for the specific organization or project. (Readers familiar with a particular agile software method are encouraged to produce an analysis and test strategy template for that method.)

Some elements of a standard test plan can be derived from the organization of the project according to the adopted method: items to be verified (all units corresponding to user stories), suspend/resume criteria (usually all tests must be passed, but we may adopt different criteria in specific cases) and test deliverables (at least the ones that can be produced automatically). Other elements may be deduced only partially. The overall process includes information about resources and responsibilities, but we may need to refine the preliminary information for some methods. Features to be tested, tasks and schedule are partially implied by the method, but may need to be adapted to the specific situation.

A typical test tool, such as JUnit, provides information that could be used to automatically produce test suite and test case specifications as well as detailed test reports. It may be useful to add Javadoc comments to link JUnit tests to user stories (and thereby to the test plan). Summary reports may be generated automatically from detailed reports.

24.2. To define a tabular specification of a suite of test cases, we can distinguish between information common to the whole suite, and information that characterizes each test case. We can keep the information common to the whole suite as "global information" outside the table, and we can define a column for each item that characterizes the test cases.

Referring to the test case specification at page 464, *Test Item* is common to all the cases of the suite, and can be kept outside the table. All other items vary among test cases and thus will correspond to columns of the table.

We will have simple columns for Test Case Identifier (*TC id.*), Output Specification (*Output*), Environmental Needs (*Env. Needs*), Special Procedural Requirements (*Special Proc. Req.*), and Intercase Dependencies (*Intercase Dep.*)

The input specification will correspond to a set of columns, one for each input item: Model number (*Model No.*), Number of required slots for selected model (*#SMRS*), Number of optional slots for selected model (*#SMOS*), ..., Number of components in DB (*No. Comp. in DB*). Each input item is characterized by a Test case specification (*Spec*) and a Test case (*Value*), and we will thus have two columns for each of the items that characterize the input. Some items may not be required: In our example, correspondence of selection with model slots is not necessary, since it can be deduced from the other fields, and we thus did not include it in the table.

Even in a more compact format, tables may grow fairly large. Test Designers will typically use spreadsheets, which can easily accommodate tables larger than a printed page. On the following page we present the table in two levels to fit the page size.

Overall structure of the tabular specification

TC Id.	Test Case Specification and Data						Output	Env. Needs	Special Proc. Req.	Inter Case Dep.
	Model No.		…	…	No. Comp. In DB					
	Spec	Value	…	…	Spec	Value				
WB07-15.01.C09	valid	C20	…	…	many	DBc1	valid	EN1	-	-
WB07-15.01.C10	valid	C22	…	…	many	DBc1	valid	EN1	-	-
…	…	…	…	…	…	…	…	…	…	…

Input Specification

Test Case Specification and Data																	
Model No.		#SMRS		#SMOS		No. Req <> empty		No. Opt <> empty		Req. Sel.		Opt. Sel.		No. Mod. in DB		No. Comp. In	
Spec	Val	Spec	Val	Spec	Val	Spec	Val	Spec	Val	Spec	Val	Spec	Val	Spec	Val	Spec	Val
valid	C20	many	5	none	4	= No. Req. Sl.	5	< No. Opt. Sl.	0	all valid	Req. C09	all valid	Opt. C09	many	DB m1	many	DB c1
valid	C22	many	3	none	6	= No. Req. Sl.	3	< No. Opt. Sl.	2	all valid	Req. C10	all valid	Opt. C10	many	DB m1	many	DB c1
…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…

When the information in a cell is large, we may use a reference to a detailed description, to keep the size of the table manageable.

In the example, the values of Required component selection (*Req. sel.*) and Optional component selection (*Opt. sel.*) are lists of actual components, and include a number of elements that vary from case to case. We use references to lists outside the table (*Req.C09*, *Req.C10*, *Opt.C09*, *Opt.C10*) to keep the table small. The values of Number of models in DB (*No. Mod. In DB*), Number of components in DB (*No. Comp. In DB*), Environmental needs (*Env. Needs*) and Special procedural requirements (*Spec. Proc. Req.*) may be complex descriptions that do not fit reasonably in a table cell. We use references to external descriptions: DB_{m1}, DB_{c1}, EN_1. Intercase dependencies (*Intercase Dep.*) will include a list of table entries. A dash (-) indicates that the value is not relevant for a test case.

24.3. The features of a checklist for inspecting test design specification documents correspond to the sections that characterize the document: Test Suite Identifier, Features To Be Tested, Approach, Procedure, Test Cases and Pass/Fail Criterion. For each feature, the items to be checked depend on the automatic checks available in the company: We may for example check for uniqueness and compliance of the Test Suite Identifier with Company standards, unless the Id is automatically generated or checked.

FEATURES (where to look and how to check): item (what to check)			
TEST SUITE IDENTIFIER: is the ID ...?	yes	no	comments
compliant with Company standards			
unique			
FEATURES TO BE TESTED: is the feature ...?	yes	no	comments
clearly identified or specified			
APPROACH: is the approach...?	yes	no	comments
clearly specified			
compliant with applicable test strategy and standards			
appropriate for application domain and feature to be tested			
applicable with the available tools and knowledge			
PROCEDURE	yes	no	comments
procedure is executable			
procedure is automated to the extent possible			
description is unambiguous			
description is complete and self-contained			
TEST CASES: does the set of test cases satisfy the following properties?	yes	no	comments
meets criterion imposed by the test approach			
PASS/FAIL CRITERION: is the criterion ...?	yes	no	comments
compliant with applicable strategy and standards			
unambiguous			
easily verifiable			

Checks on single test cases do not belong to this checklist, but to a checklist for test case specification documents.

The checklist does not includes checks of sections common to all technical documentation (Title, Approval, History, Table of Contents, Summary, etc), which belong to a general checklist for inspecting documents and not to a checklist to inspect a specific class of documents. (In some cases, general and specific checklists may be merged in a single checklist for practical purposes, but the checklists are conceptually distinct).

24.4. Some design decisions and considerations include

1. A single top-level repository or multiple repositories?

   Distributed development (Milan and Eugene) suggests posssibly using distinct top-level CVS repositories, which are more difficult to manage but may avoid performance and availability issues for distant users. The choice may depend on whether other aspects of the repository organization tend to break down along geographic lines. We could even choose to arbitrarily divide the repository along geographic lines (all Milano-based artifacts in one repository, and all Eugene-based artifacts in another, regardless of document kind or role), but this is likely to conflict with other considerations.

2. Break down by artifact kind (e.g., design document versus source code) or by subsystem (e.g., business logic versus Web interface)? There will be breakdown by several criteria, but which divisions are subordinate to which others?

   Where two sub-hierarchies should mirror each other (e.g., unit test suites associated with source code units), it is helpful for them to be near each other in the organization; inconsistencies are harder to avoid and repair when they involve parts scattered throughout the organization.

3. Separate sub-hierarchies for documentation in each language, or one hierarchy of documents with different language versions kept together.

   This decision has elements of the previous one. It is likely that the organization of documents mirror each other (e.g., chapter 3 of the English-language user manual probably corresponds to chapter 3 of the Italian user manual), which suggests keeping them close together in the organization.

Other design choices and considerations are also possible.