Chapter 2

A Framework for Test and Analysis

The purpose of software test and analysis is either to assess software qualities or else to make it possible to improve the software by finding defects. Of the many kinds of software qualities, those addressed by the analysis and test techniques discussed in this book are the *dependability* properties of the software *product*.

There are no perfect test or analysis techniques, nor a single "best" technique for all circumstances. Rather, techniques exist in a complex space of trade-offs, and often have complementary strengths and weaknesses. This chapter describes the nature of those trade-offs and some of their consequences, and thereby a conceptual framework for understanding and better integrating material from later chapters on individual techniques.

It is unfortunate that much of the available literature treats testing and analysis as independent or even as exclusive choices, removing the opportunity to exploit their complementarities. Armed with a basic understanding of the trade-offs and of strengths and weaknesses of individual techniques, one can select from and combine an array of choices to improve the cost-effectiveness of verification.

2.1 Validation and Verification

While software products and processes may be judged on several properties ranging from time-to-market to performance to usability, the software test and analysis techniques we consider are focused more narrowly on improving or assessing dependability.

Assessing the degree to which a software system actually fulfills its requirements, in the sense of meeting the user's real needs, is called *validation.* Fulfilling requirements is not the same as conforming to a requirements specification. A specification is a statement about a particular proposed solution to a problem, and that proposed solution may or may not achieve its goals. Moreover, specifications are written by people, and therefore contain mistakes. A system that meets its actual goals is *useful*, while a

Δ validation

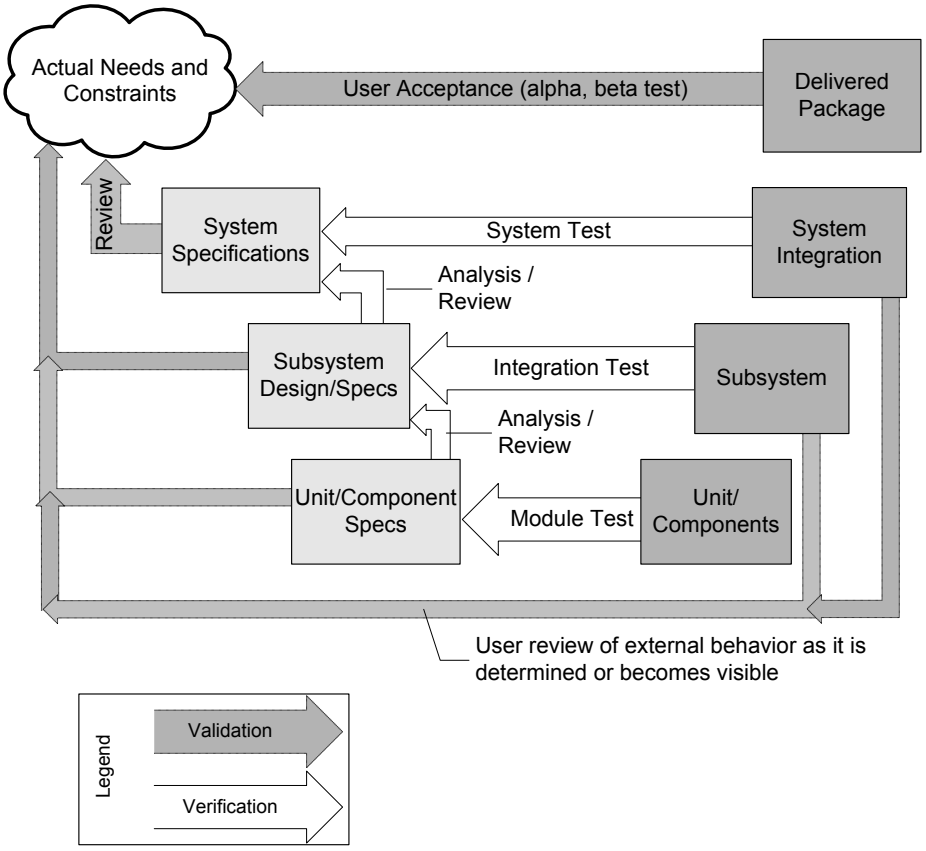

Figure 2.1: Validation activities check work products against actual user requirements, while verification activities check consistency of work products.

Δ dependable

Δ verification

system that is consistent with its specification is *dependable*.[1]

"Verification" is checking the consistency of an implementation with a specification. Here, "specification" and "implementation" are roles, not particular artifacts. For example, an overall design could play the role of "specification" and a more detailed design could play the role of "implementation"; checking whether the detailed design is consistent with the overall design would then be verification of the detailed design. Later, the same detailed design could play the role of "specification" with respect to

[1] A good requirements document, or set of documents, should include both a requirements analysis and a requirements specification, and should clearly distinguish between the two. The requirements analysis describes the problem. The specification describes a proposed solution. This is not a book about requirements engineering, but we note in passing that confounding requirements analysis with requirements specification will inevitably have negative impacts on both validation and verification.

source code, which would be verified against the design. In every case, though, verification is a check of consistency between two descriptions, in contrast to validation which compares a description (whether a requirements specification, a design, or a running system) against actual needs.

Figure 2.1 sketches the relation of verification and validation activities with respect to artifacts produced in a software development project. The figure should not be interpreted as prescribing a sequential process, since the goal of a consistent set of artifacts and user satisfaction are the same whether the software artifacts (specifications, design, code, etc.) are developed sequentially, iteratively, or in parallel. Verification activities check consistency between descriptions (design and specifications) at adjacent levels of detail, and between these descriptions and code.[2] Validation activities attempt to gauge whether the system actually satisfies its intended purpose.

Validation activities refer primarily to the overall system specification and the final code. With respect to overall system specification, validation checks for discrepancies between actual needs and the system specification as laid out by the analysts, to ensure that the specification is an adequate guide to building a product that will fulfill its goals. With respect to final code, validation aims at checking discrepancies between actual need and the final product, to reveal possible failures of the development process and to make sure the product meets end-user expectations. Validation checks between the specification and final product are primarily checks of decisions that were left open in the specification (e.g., details of the user interface or product features). Chapter 4 provides a more thorough discussion of validation and verification activities in particular software process models.

We have omitted one important set of verification checks from Figure 2.1 to avoid clutter. In addition to checks that compare two or more artifacts, verification includes checks for self-consistency and well-formedness. For example, while we cannot judge that a program is "correct" except in reference to a specification of what it should do, we can certainly determine that some programs are "incorrect" because they are ill-formed. We may likewise determine that a specification itself is ill-formed because it is inconsistent (requires two properties that cannot both be true) or ambiguous (can be interpreted to require some property or not), or because it does not satisfy some other well-formedness constraint that we impose, such as adherence to a standard imposed by a regulatory agency.

Validation against actual requirements necessarily involves human judgment and the potential for ambiguity, misunderstanding, and disagreement. In contrast, a specification should be sufficiently precise and unambiguous that there can be no disagreement about whether a particular system behavior is acceptable. While the term *testing* is often used informally both for gauging usefulness and verifying the product, the activities differ in both goals and approach. Our focus here is primarily on dependability, and thus primarily on verification rather than validation, although techniques for validation and the relation between the two is discussed further in Chapter 22.

Dependability properties include correctness, reliability, robustness, and safety. Correctness is absolute consistency with a specification, always and in all circumstances. Correctness with respect to nontrivial specifications is almost never achieved.

[2] This part of the diagram is a variant of the well-known "V model" of verification and validation.

Reliability is a statistical approximation to correctness, expressed as the likelihood of correct behavior in expected use. Robustness, unlike correctness and reliability, weighs properties as more and less critical, and distinguishes which properties should be maintained even under exceptional circumstances in which full functionality cannot be maintained. Safety is a kind of robustness in which the critical property to be maintained is avoidance of particular hazardous behaviors. Dependability properties are discussed further in Chapter 4.

2.2 Degrees of Freedom

Given a precise specification and a program, it seems that one ought to be able to arrive at some logically sound argument or proof that a program satisfies the specified properties. After all, if a civil engineer can perform mathematical calculations to show that a bridge will carry a specified amount of traffic, shouldn't we be able to similarly apply mathematical logic to verification of programs?

For some properties and some very simple programs, it is in fact possible to obtain a logical correctness argument, albeit at high cost. In a few domains, logical correctness arguments may even be cost-effective for a few isolated, critical components (e.g., a safety interlock in a medical device). In general, though, one cannot produce a complete logical "proof" for the full specification of practical programs in full detail. This is not just a sign that technology for verification is immature. It is, rather, a consequence of one of the most fundamental properties of computation.

undecidability

Even before programmable digital computers were in wide use, computing pioneer Alan Turing proved that some problems cannot be solved by any computer program. The universality of computers — their ability to carry out any programmed algorithm, including simulations of other computers — induces logical paradoxes regarding programs (or algorithms) for analyzing other programs. In particular, logical contradictions ensue from assuming that there is some program P that can, for some arbitrary program Q and input I, determine whether Q eventually halts. To avoid those logical contradictions, we must conclude that no such program for solving the "halting problem" can possibly exist.

halting problem

Countless university students have encountered the halting problem in a course on the theory of computing, and most of those who have managed to grasp it at all have viewed it as a purely theoretical result that, whether fascinating or just weird, is irrelevant to practical matters of programming. They have been wrong. Almost every interesting property regarding the behavior of computer programs can be shown to "embed" the halting problem, that is, the existence of an infallible algorithmic check for the property of interest would imply the existence of a program that solves the halting problem, which we know to be impossible.

In theory, undecidability of a property S merely implies that for each verification technique for checking S, there is at least one "pathological" program for which that technique cannot obtain a correct answer in finite time. It does not imply that verification will always fail or even that it will usually fail, only that it will fail in at least one case. In practice, failure is not only possible but common, and we are forced to accept a significant degree of inaccuracy.

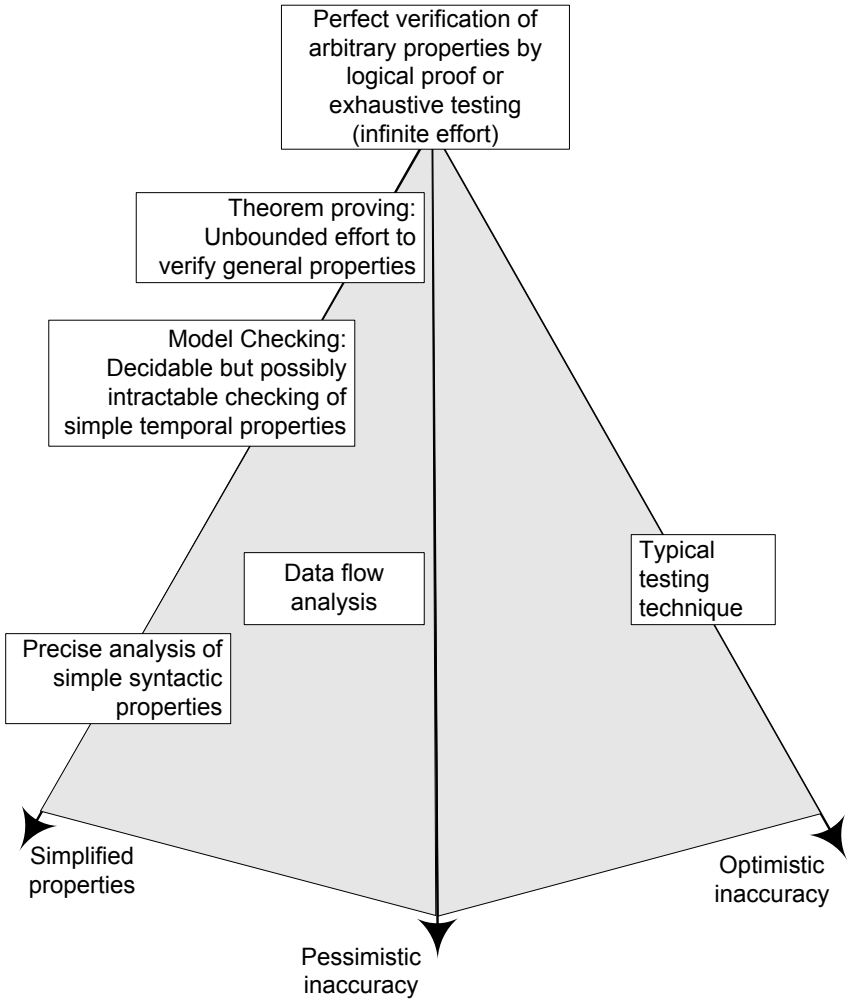

Figure 2.2: Verification trade-off dimensions

Program testing is a verification technique and is as vulnerable to undecidability as other techniques. Exhaustive testing, that is, executing and checking every possible behavior of a program, would be a "proof by cases," which is a perfectly legitimate way to construct a logical proof. How long would this take? If we ignore implementation details such as the size of the memory holding a program and its data, the answer is "forever." That is, for most programs, exhaustive testing cannot be completed in any finite amount of time.

Suppose we do make use of the fact that programs are executed on real machines with finite representations of memory values. Consider the following trivial Java class:

```
1  class Trivial{
2     static int sum(int a, int b) { return a + b; }
3  }
```

The Java language definition states that the representation of an int is 32 binary digits, and thus there are only $2^{32} \times 2^{32} = 2^{64} \approx 10^{21}$ different inputs on which the method Trivial.sum() need be tested to obtain a proof of its correctness. At one nanosecond (10^{-9} seconds) per test case, this will take approximately 10^{12} seconds, or about 30,000 years.

Δ pessimistic
Δ optimistic

A technique for verifying a property can be inaccurate in one of two directions (Figure 2.2). It may be *pessimistic*, meaning that it is not guaranteed to accept a program even if the program does possess the property being analyzed, or it can be *optimistic* if it may accept some programs that do not possess the property (i.e., it may not detect all violations). Testing is the classic optimistic technique, because no finite number of tests can guarantee correctness. Many automated program analysis techniques for properties of program behaviors[3] are pessimistic with respect to the properties they are designed to verify. Some analysis techniques may give a third possible answer, "don't know." We can consider these techniques to be either optimistic or pessimistic depending on how we interpret the "don't know" result. Perfection is unobtainable, but one can choose techniques that err in only a particular direction.

A software verification technique that errs only in the pessimistic direction is called a *conservative* analysis. It might seem that a conservative analysis would always be preferable to one that could accept a faulty program. However, a conservative analysis will often produce a very large number of spurious error reports, in addition to a few accurate reports. A human may, with some effort, distinguish real faults from a few spurious reports, but cannot cope effectively with a long list of purported faults of which most are false alarms. Often only a careful choice of complementary optimistic and pessimistic techniques can help in mutually reducing the different problems of the techniques and produce acceptable results.

In addition to pessimistic and optimistic inaccuracy, a third dimension of compromise is possible: substituting a property that is more easily checked, or constraining the class of programs that can be checked. Suppose we want to verify a property *S*, but we are not willing to accept the optimistic inaccuracy of testing for *S*, and the only

[3] Why do we bother to say "properties of program behaviors" rather than "program properties?" Because simple syntactic properties of program text, such as declaring variables before they are used or indenting properly, can be decided efficiently and precisely.

A Note on Terminology

Many different terms related to *pessimistic* and *optimistic* inaccuracy appear in the literature on program analysis. We have chosen these particular terms because it is fairly easy to remember which is which. Other terms a reader is likely to encounter include:

Safe: A *safe* analysis has no optimistic inaccuracy; that is, it accepts only correct programs. In other kinds of program analysis, safety is related to the goal of the analysis. For example, a safe analysis related to a program optimization is one that allows that optimization only when the result of the optimization will be correct.

Sound: Soundness is a term to describe evaluation of formulas. An analysis of a program P with respect to a formula F is *sound* if the analysis returns *True* only when the program actually does satisfy the formula. If satisfaction of a formula F is taken as an indication of correctness, then a *sound* analysis is the same as a *safe* or *conservative* analysis.

If the sense of F is reversed (i.e., if the truth of F indicates a fault rather than correctness) then a *sound* analysis is not necessarily *conservative*. In that case it is allowed optimistic inaccuracy but must not have pessimistic inaccuracy. (Note, however, that use of the term *sound* has not always been consistent in the software engineering literature. Some writers use the term *unsound* as we use the term *optimistic*.)

Complete: Completeness, like soundness, is a term to describe evaluation of formulas. An analysis of a program P with respect to a formula F is *complete* if the analysis always returns *True* when the program actually does satisfy the formula. If satisfaction of a formula F is taken as an indication of correctness, then a *complete* analysis is one that admits only optimistic inaccuracy. An analysis that is sound but incomplete is a conservative analysis.

available static analysis techniques for S result in such huge numbers of spurious error messages that they are worthless. Suppose we know some property S' that is a sufficient, but not necessary, condition for S (i.e., the validity of S' implies S, but not the contrary). Maybe S' is so much simpler than S that it can be analyzed with little or no pessimistic inaccuracy. If we check S' rather than S, then we may be able to provide precise error messages that describe a real violation of S' rather than a potential violation of S.

Many examples of substituting simple, checkable properties for actual properties of interest can be found in the design of modern programming languages. Consider, for example, the property that each variable should be initialized with a value before its value is used in an expression. In the C language, a compiler cannot provide a precise static check for this property, because of the possibility of code like the following:

```
1    int i, sum;
2    int first=1;
3    for (i=0; i<10; ++i) {
4        if (first) {
5            sum=0; first=0;
6        }
7        sum += i;
8    }
```

It is impossible in general to determine whether each control flow path can be executed, and while a human will quickly recognize that the variable sum is initialized on the first iteration of the loop, a compiler or other static analysis tool will typically not be able to rule out an execution in which the initialization is skipped on the first iteration. Java neatly solves this problem by making code like this illegal; that is, the rule is that a variable must be initialized on *all* program control paths, whether or not those paths can ever be executed.

Software developers are seldom at liberty to design new restrictions into the programming languages and compilers they use, but the same principle can be applied through external tools, not only for programs but also for other software artifacts. Consider, for example, the following condition that we might wish to impose on requirements documents:

(1) Each significant domain term shall appear with a definition in the glossary of the document.

This property is nearly impossible to check automatically, since determining whether a particular word or phrase is a "significant domain term" is a matter of human judgment. Moreover, human inspection of the requirements document to check this requirement will be extremely tedious and error-prone. What can we do? One approach is to separate the decision that requires human judgment (identifying words and phrases as "significant") from the tedious check for presence in the glossary.

(1a) Each significant domain term shall be set off in the requirements document by the use of a standard style *term*. The default visual representation of the

term style is a single underline in printed documents and purple text in on-line displays.

(1b) Each word or phrase in the *term* style shall appear with a definition in the glossary of the document.

Property (1a) still requires human judgment, but it is now in a form that is much more amenable to inspection. Property (1b) can be easily automated in a way that will be completely precise (except that the task of determining whether definitions appearing in the glossary are clear and correct must also be left to humans).

As a second example, consider a Web-based service in which user sessions need not directly interact, but they do read and modify a shared collection of data on the server. In this case a critical property is maintaining integrity of the shared data. Testing for this property is notoriously difficult, because a "race condition" (interference between writing data in one process and reading or writing related data in another process) may cause an observable failure only very rarely.

Fortunately, there is a rich body of applicable research results on concurrency control that can be exploited for this application. It would be foolish to rely primarily on direct testing for the desired integrity properties. Instead, one would choose a (well-known, formally verified) concurrency control protocol, such as the two-phase locking protocol, and rely on some combination of static analysis and program testing to check conformance to that protocol. Imposing a particular concurrency control protocol substitutes a much simpler, *sufficient* property (two-phase locking) for the complex property of interest (serializability), at some cost in generality; that is, there are programs that violate two-phase locking and yet, by design or dumb luck, satisfy serializability of data access.

It is a common practice to further impose a global order on lock accesses, which again simplifies testing and analysis. Testing would identify execution sequences in which data is accessed without proper locks, or in which locks are obtained and relinquished in an order that does not respect the two-phase protocol or the global lock order, even if data integrity is not violated on that particular execution, because the locking protocol failure indicates the potential for a dangerous race condition in some other execution that might occur only rarely or under extreme load.

With the adoption of coding conventions that make locking and unlocking actions easy to recognize, it may be possible to rely primarily on flow analysis to determine conformance with the locking protocol, with the role of dynamic testing reduced to a "back-up" to raise confidence in the soundness of the static analysis. Note that the critical decision to impose a particular locking protocol is *not* a post-hoc decision that can be made in a testing "phase" at the end of development. Rather, the plan for verification activities with a suitable balance of cost and assurance is part of system design.

2.3 Varieties of Software

The software testing and analysis techniques presented in the main parts of this book were developed primarily for procedural and object-oriented software. While these

"generic" techniques are at least partly applicable to most varieties of software, particular application domains (e.g., real-time and safety-critical software) and construction methods (e.g., concurrency and physical distribution, graphical user interfaces) call for particular properties to be verified, or the relative importance of different properties, as well as imposing constraints on applicable techniques. Typically a software system does not fall neatly into one category but rather has a number of relevant characteristics that must be considered when planning verification.

As an example, consider a physically distributed (networked) system for scheduling a group of individuals. The possibility of concurrent activity introduces considerations that would not be present in a single-threaded system, such as preserving the integrity of data. The concurrency is likely to introduce nondeterminism, or else introduce an obligation to show that the system is deterministic, either of which will almost certainly need to be addressed through some formal analysis. The physical distribution may make it impossible to determine a global system state at one instant, ruling out some simplistic approaches to system test and, most likely, suggesting an approach in which dynamic testing of design conformance of individual processes is combined with static analysis of their interactions. If in addition the individuals to be coordinated are fire trucks, then the criticality of assuring prompt response will likely lead one to choose a design that is amenable to strong analysis of worst-case behavior, whereas an average-case analysis might be perfectly acceptable if the individuals are house painters.

As a second example, consider the software controlling a "soft" dashboard display in an automobile. The display may include ground speed, engine speed (rpm), oil pressure, fuel level, and so on, in addition to a map and navigation information from a global positioning system receiver. Clearly usability issues are paramount, and may even impinge on safety (e.g., if critical information can be hidden beneath or among less critical information). A disciplined approach will not only place a greater emphasis on validation of usability throughout development, but to the extent possible will also attempt to codify usability guidelines in a form that permits verification. For example, if the usability group determines that the fuel gauge should always be visible when the fuel level is below a quarter of a tank, then this becomes a specified property that is subject to verification. The graphical interface also poses a challenge in effectively checking output. This must be addressed partly in the architectural design of the system, which can make automated testing feasible or not depending on the interfaces between high-level operations (e.g., opening or closing a window, checking visibility of a window) and low-level graphical operations and representations.

Summary

Verification activities are comparisons to determine the consistency of two or more software artifacts, or self-consistency, or consistency with an externally imposed criterion. Verification is distinct from validation, which is consideration of whether software fulfills its actual purpose. Software development always includes some validation and some verification, although different development approaches may differ greatly in their relative emphasis.

Precise answers to verification questions are sometimes difficult or impossible to

obtain, in theory as well as in practice. Verification is therefore an art of compromise, accepting some degree of optimistic inaccuracy (as in testing) or pessimistic inaccuracy (as in many static analysis techniques) or choosing to check a property that is only an approximation of what we really wish to check. Often the best approach will not be exclusive reliance on one technique, but careful choice of a portfolio of test and analysis techniques selected to obtain acceptable results at acceptable cost, and addressing particular challenges posed by characteristics of the application domain or software.

Further Reading

The "V" model of verification and validation (of which Figure 2.1 is a variant) appears in many software engineering textbooks, and in some form can be traced at least as far back as Myers' classic book [Mye79]. The distinction between validation and verification as given here follow's Boehm [Boe81], who has most memorably described validation as "building the right system" and verification as "building the system right."

The limits of testing have likewise been summarized in a famous aphorism, by Dijkstra [Dij72] who pronounced that "Testing can show the presence of faults, but not their absence." This phrase has sometimes been interpreted as implying that one should always prefer formal verification to testing, but the reader will have noted that we do not draw that conclusion. Howden's 1976 paper [How76] is among the earliest treatments of the implications of computability theory for program testing.

A variant of the diagram in Figure 2.2 and a discussion of pessimistic and optimistic inaccuracy were presented by Young and Taylor [YT89]. A more formal characterization of conservative abstractions in static analysis, called abstract interpretation, was introduced by Cousot and Cousot in a seminal paper that is, unfortunately, nearly unreadable [CC77]. We enthusiastically recommend Jones's lucid introduction to abstract interpretation [JN95], which is suitable for readers who have a firm general background in computer science and logic but no special preparation in programming semantics.

There are few general treatments of trade-offs and combinations of software testing and static analysis, although there are several specific examples, such as work in communication protocol conformance testing [vBDZ89, FvBK+91]. The two-phase locking protocol mentioned in Section 2.2 is described in several texts on databases; Bernstein et al. [BHG87] is particularly thorough.

Exercises

2.1. The Chipmunk marketing division is worried about the start-up time of the new version of the RodentOS operating system (an (imaginary) operating system of Chipmunk). The marketing division representative suggests a software requirement stating that the start-up time shall not be annoying to users.

Explain why this simple requirement is not verifiable and try to reformulate the requirement to make it verifiable.

2.2. Consider a simple specification language *SL* that describes systems diagrammatically in terms of *functions*, which represent data transformations and correspond to nodes of the diagram, and *flows*, which represent data flows and correspond to arcs of the diagram.[4] Diagrams can be hierarchically refined by associating a function F (a node of the diagram) with an *SL* specification that details function F. Flows are labeled to indicate the type of data.

Suggest some checks for self-consistency for *SL*.

2.3. A calendar program should provide *timely* reminders; for example, it should remind the user of an upcoming event early enough for the user to take action, but not too early. Unfortunately, "early enough" and "too early" are qualities that can only be validated with actual users. How might you derive verifiable dependability properties from the timeliness requirement?

2.4. It is sometimes important in multi-threaded applications to ensure that a sequence of accesses by one thread to an aggregate data structure (e.g., some kind of table) appears to other threads as an atomic transaction. When the shared data structure is maintained by a database system, the database system typically uses concurrency control protocols to ensure the atomicity of the transactions it manages. No such automatic support is typically available for data structures maintained by a program in main memory.

Among the options available to programmers to ensure serializability (the illusion of atomic access) are the following:

- The programmer could maintain very coarse-grain locking, preventing any interleaving of accesses to the shared data structure, even when such interleaving would be harmless. (For example, each transaction could be encapsulated in an single synchronized Java method.) This approach can cause a great deal of unnecessary blocking between threads, hurting performance, but it is almost trivial to verify either automatically or manually.
- Automated static analysis techniques can sometimes verify serializability with finer-grain locking, even when some methods do not use locks at all. This approach can still reject some sets of methods that would ensure serializability.
- The programmer could be required to use a particular concurrency control protocol in his or her code, and we could build a static analysis tool that checks for conformance with that protocol. For example, adherence to the common two-phase-locking protocol, with a few restrictions, can be checked in this way.
- We might augment the data accesses to build a *serializability graph* structure representing the "happens before" relation among transactions in testing. It can be shown that the transactions executed in serializable manner if and only if the serializability graph is acyclic.

[4] Readers expert in Structured Analysis may have noticed that *SL* resembles a simple Structured Analysis specification

Compare the relative positions of these approaches on the three axes of verification techniques: pessimistic inaccuracy, optimistic inaccuracy, and simplified properties.

2.5. When updating a program (e.g., for removing a fault, changing or adding a functionality), programmers may introduce new faults or expose previously hidden faults. To be sure that the updated version maintains the functionality provided by the previous version, it is common practice to reexecute the test cases designed for the former versions of the program. Reexecuting test cases designed for previous versions is called regression testing.

When testing large complex programs, the number of regression test cases may be large. If updated software must be expedited (e.g., to repair a security vulnerability before it is exploited), test designers may need to select a subset of regression test cases to be reexecuted.

Subsets of test cases can be selected according to any of several different criteria. An interesting property of some regression test selection criteria is that they do not to exclude any test case that could possibly reveal a fault.

How would you classify such a property according to the sidebar of page 21?