Chapter 1

Software Test and Analysis in a Nutshell

Before considering individual aspects and techniques of software analysis and testing, it is useful to view the "big picture" of software quality in the context of a software development project and organization. The objective of this chapter is to introduce the range of software verification and validation activities and a rationale for selecting and combining them within a software development process. This overview is necessarily cursory and incomplete, with many details deferred to subsequent chapters.

1.1 Engineering Processes and Verification

Engineering disciplines pair design and construction activities with activities that check intermediate and final products so that defects can be identified and removed. Software engineering is no exception: Construction of high quality software requires complementary pairing of design and verification activities throughout development.

Verification and design activities take various forms ranging from those suited to highly repetitive construction of non-critical items for mass markets to highly customized or highly critical products. Appropriate verification activities depend on the engineering discipline, the construction process, the final product, and quality requirements.

Repetition and high levels of automation in production lines reduce the need for verification of individual products. For example, only a few key components of products like screens, circuit boards, and toasters are verified individually. The final products are tested statistically. Full test of each individual product may not be economical, depending on the costs of test, the reliability of the production process, and the costs of field failures.

Even for some mass market products, complex processes or stringent quality requirements may require both sophisticated design and advanced product verification procedures. For example, computers, cars, and aircraft, despite being produced in series, are checked individually before release to customers. Other products are not built in series, but are engineered individually through highly evolved processes and tools. Custom houses, race cars, and software are not built in series. Rather, each house, each racing car, and each software package is at least partly unique in its design and functionality. Such products are verified individually both during and after production to identify and eliminate faults.

Verification of goods produced in series, e.g., screens, boards, or toasters, consists in repeating a predefined set of tests and analyses that indicate whether the products meet the required quality standards. In contrast, verification of a unique product, such as a house, requires the design of a specialized set of tests and analyses to assess the quality of that product. Moreover, the relationship between the test and analysis results and the quality of the product cannot be defined once for all items, but must be assessed for each product. For example, the set of resistance tests for assessing the quality of a floor must be customized for each floor, and the resulting quality depends on the construction methods and the structure of the building.

The difficulty of verification grows with the complexity and variety of the products. Small houses built with comparable technologies in analogous environments can be verified with standardized procedures. The tests are parameterized to the particular house, but are nonetheless routine. Verification of a skyscraper or of a house built in an extreme seismic area, on the other hand, may not be easily generalized, instead requiring specialized tests and analyses designed particularly for the case at hand.

Software is among the most variable and complex of artifacts engineered on a regular basis. Quality requirements of software used in one environment may be quite different and incompatible with quality requirements of a different environment or application domain, and its structure evolves and often deteriorates as the software system grows. Moreover, the inherent non-linearity of software systems and uneven distribution of faults complicates verification. If an elevator can safely carry a load of 1000 kg, it can also safely carry any smaller load, but if a procedure correctly sorts a set of 256 elements, it may fail on a set of 255 or 53 or 12 elements, as well as on 257 or 1023.

The cost of software verification often exceeds half the overall cost of software development and maintenance. Advanced development technologies and powerful supporting tools can reduce the frequency of some classes of errors, but we are far from eliminating errors and producing fault-free software. In many cases new development approaches introduce new subtle kinds of faults, which may be more difficult to reveal and remove than classic faults. This is the case, for example, with distributed software, which can present problems of deadlock or race conditions that are not present in sequential programs. Likewise, object-oriented development introduces new problems due to the use of polymorphism, dynamic binding and private state that are absent or less pronounced in procedural software.

The variety of problems and the richness of approaches make it challenging to choose and schedule the right blend of techniques to reach the required level of quality within cost constraints. There are no fixed recipes for attacking the problem of verifying a software product. Even the most experienced specialists do not have pre-cooked solutions, but need to design a solution that suits the problem, the requirements, and the development environment.

1.2 Basic Questions

To start understanding how to attack the problem of verifying software, let us consider a hypothetical case. The Board of Governors of Chipmunk Computers, an (imaginary) computer manufacturer, decides to add new on-line shopping functions to the company web presence to allow customers to purchase individually configured products. Let us assume the role of quality manager. To begin, we need to answer a few basic questions:

- When do verification and validation start? When are they complete?
- What particular techniques should be applied during development of the product to obtain acceptable quality at an acceptable cost?
- How can we assess the readiness of a product for release?
- How can we control the quality of successive releases?
- How can the development process itself be improved over the course of the current and future projects to improve products and make verification more cost-effective?

1.3 When do verification and validation start and end?

Although some primitive software development processes concentrate test and analysis at the end of the development cycle, and the job title "tester" in some organizations still refers to a person who merely executes test cases on a complete product, today it is widely understood that execution of tests is a small part of the verification and validation process required to assess and maintain the quality of a software product.

Verification and validation start as soon as we decide to build a software product, or even before. In the case of Chipmunk Computers, when the Board of Governors asks the IT manager for a feasibility study, the IT manager considers not only functionality and development costs, but also the required qualities and their impact on the overall cost.

The Chipmunk software quality manager participates with other key designers in the feasibility study, focusing in particular on risk analysis and the measures needed to assess and control quality at each stage of development. The team assesses the impact of new features and new quality requirements on the full system and considers the contribution of quality control activities to development cost and schedule. For example, migrating sales functions into the Chipmunk web site will increase the criticality of system availability and introduce new security issues. A feasibility study that ignored quality could lead to major unanticipated costs and delays and very possibly to project failure.

The feasibility study necessarily involves some tentative architectural design, e.g., a division of software structure corresponding to a division of responsibility between a human interface design team and groups responsible for core business software ("business logic") and supporting infrastructure, and a rough build plan breaking the project into a series of incremental deliveries. Opportunities and obstacles for cost-effective verification are important considerations in factoring the development effort into subsystems and phases, and in defining major interfaces.

Overall architectural design divides work and separates qualities that can be verified independently in the different subsystems, thus easing the work of the testing team as well as other developers. For example, the Chipmunk design team divides the system into a presentation layer, back-end logic, and infrastructure. Development of the three subsystems is assigned to three different teams with specialized experience, each of which must meet appropriate quality constraints. The quality manager steers the early design toward a separation of concerns that will facilitate test and analysis.

In the Chipmunk web presence, a clean interface between the presentation layer and back end logic allows a corresponding division between usability testing (which is the responsibility of the human interface group, rather than the quality group) and verification of correct functioning. A clear separation of infrastructure from business logic serves a similar purpose. Responsibility for a small kernel of critical functions is allocated to specialists on the infrastructure team, leaving effectively checkable rules for consistent use of those functions throughout other parts of the system.

Taking into account quality constraints during early breakdown into subsystems allows for a better allocation of quality requirements and facilitates both detailed design and testing. However, many properties cannot be guaranteed by one subsystem alone. The initial breakdown of properties given in the feasibility study will be detailed during later design and may result in "cross quality requirements" among subsystems. For example, to guarantee a given security level, the infrastructure design team may require the verification of absence of some specific security holes, e.g., buffer overflow, in other parts of the system.

The initial build plan also includes some preliminary decisions about test and analysis techniques to be used in development. For example, the preliminary prototype of Chipmunk online sales functionality will not undergo complete acceptance testing, but will be used to validate the requirements analysis and some design decisions. Acceptance testing of the first release will be primarily based on feedback from selected retail stores, but will also include complete checks to verify absence of common security holes. The second release will include full acceptance test and reliability measures.

If the feasibility study leads to a project commitment, verification and validation activities will commence with other development activities, and like development itself will continue long past initial delivery of a product. Chipmunk's new web-based functions will be delivered in a series of phases, with requirements reassessed and modified after each phase, so it is essential that the V&V plan be cost-effective over a series of deliveries whose outcome cannot be fully known in advance. Even when the project is "complete," the software will continue to evolve and adapt to new conditions, such as a new version of the underlying data base, or new requirements, such as the opening of a European sales division of Chipmunk. V&V continues through each small or large change to the system.

Why Combine Techniques?

No single test or analysis technique can serve all purposes. The primary reasons for combining techniques, rather than choosing a single "best" technique, are

- Effectiveness for different classes of faults. For example, race conditions are very difficult to find with conventional testing, but they can be detected with static analysis techniques.
- Applicability at different points in a project. For example, we can apply inspection techniques very early to requirements and design representations that are not suited to more automated analyses.
- Differences in purpose. For example, systematic (non-random) testing is aimed at maximizing fault detection, but cannot be used to measure reliability; for that, statistical testing is required.
- Tradeoffs in cost and assurance. For example, one may use a relatively expensive technique to establish a few key properties of core components (e.g., a security kernel) when those techniques would be too expensive for use throughout a project.

1.4 What techniques should be applied?

The feasibility study is the first step of a complex development process that should lead to delivery of a satisfactory product through design, verification and validation activities. Verification activities steer the process towards the construction of a product that satisfies the requirements by checking the quality of intermediate artifacts as well as the ultimate product. Validation activities check the correspondence of the intermediate artifacts and the final product to users' expectations.

The choice of the set of test and analysis techniques depends on quality, cost, scheduling and resource constraints in development of a particular product. For the business logic subsystem, the quality team plans to use a preliminary prototype for validating requirements specifications. They plan to use automatic tools for simple structural checks of the architecture and design specifications. They will train staff for design and code inspections, which will be based on company checklists that identify deviations from design rules for ensuring maintainability, scalability, and correspondence between design and code.

Requirements specifications at Chipmunk are written in a structured, semi-formal format. They are not amenable to automated checking, but like any other software artifact they can be inspected by developers. The Chipmunk organization has compiled a check-list based on their rules for structuring specification documents and on experience with problems in requirements from past systems. For example, the check-list for inspecting requirements specifications at Chipmunk asks inspectors to confirm that each specified property is stated in a form that can be effectively tested.

The analysis and test plan requires inspection of requirements specifications, design specifications, source code, and test documentation. Most source code and test documentation inspections are a simple matter of soliciting an offline review by one other developer, though a handful of critical components are designated for an additional review and comparison of notes. Component interface specifications are inspected by small groups that include a representative of the "provider" and "consumer" sides of the interface, again mostly offline with exchange of notes through a discussion service. A larger group and more involved process, including a moderated inspection meeting with three or four participants, is used for inspection of a requirements specification.

Chipmunk developers produce functional unit tests with each development work assignment, as well as test oracles and any other scaffolding required for test execution. Test scaffolding is additional code needed to execute a unit or a subsystem in isolation. Test oracles check the results of executing the code and signal discrepancies between actual and expected outputs.

Test cases at Chipmunk are based primarily on interface specifications, but the extent to which unit tests exercise the control structure of programs is also measured. If less than 90% of all statements are executed by the functional tests, this is taken as an indication that either the interface specifications are incomplete (if the missing coverage corresponds to visible differences in behavior), or else additional implementation complexity hides behind the interface. Either way, additional test cases are devised based on a more complete description of unit behavior.

Integration and system tests are generated by the quality team, working from a catalog of patterns and corresponding tests. The behavior of some subsystems or components is modeled as finite-state machines, so the quality team creates test suites that exercise program paths corresponding to each state transition in the models.

Scaffolding and oracles for integration testing are part of the overall system architecture. Oracles for individual components and units are designed and implemented by programmers using tools for annotating code with conditions and invariants. The Chipmunk developers use a home-grown test organizer tool to bind scaffolding to code, schedule test runs, track faults, and organize and update regression test suites.

The quality plan includes analysis and test activities for several properties distinct from functional correctness, including performance, usability, and security. Although these are an integral part of the quality plan, their design and execution is delegated in part or whole to experts who may reside elsewhere in the organization. For example, Chipmunk maintains a small team of human factors experts in its software division. The human factors team will produce look-and-feel guidelines for the web purchasing system, which together with a larger body of Chipmunk interface design rules can be checked during inspection and test. The human factors team also produces and executes a usability testing plan.

Parts of the portfolio of verification and validation activities selected by Chipmunk are illustrated in Figure 1.1. The quality of the final product and the costs of the quality assurance activities depend on the choice of the techniques to accomplish each activity. Most important is to construct a coherent plan that can be monitored. In addition to monitoring schedule progress against the plan, Chipmunk records faults found during each activity, using this as an indicator of potential trouble spots. For example, if the number of faults found in a component during design inspections is high, additional dynamic test time will be planned for that component.

1.5 How can we assess the readiness of a product?

Analysis and testing activities during development are intended primarily to reveal faults so that they can be removed. Identifying and removing as many faults as possible is a useful objective during development, but finding all faults is nearly impossible and seldom a cost-effective objective for a non-trivial software product. Analysis and test cannot go on forever: Products must be delivered when they meet an adequate level of functionality and quality. We must have some way to specify the required level of dependability, and to determine when that level has been attained.

Different measures of dependability are appropriate in different contexts. *Availability* measures the quality of service in terms of running versus down time; *mean time between failures (MTBF)* measures the quality of the service in terms of time between failures, i.e., length of time intervals during which the service is available. *Reliability* is sometimes used synonymously with availability or MTBF, but usually indicates the fraction of all attempted operations (program runs, or interactions, or sessions) that complete successfully.

Both availability and reliability are important for the Chipmunk web presence. The availability goal is set (somewhat arbitrarily) at an average of no more than 30 minutes of down time per month. Since 30 one minute failures in the course of a day would be much worse than a single 30 minute failure, mean time between failures is separately specified as at least one week. In addition, a reliability goal of less than 1 failure per 1000 user sessions is set, with a further stipulation that certain critical failures (e.g., loss of data) must be vanishingly rare.

Having set these goals, how can Chipmunk determine when it has met them? Monitoring systematic debug testing can provide a hint, but no more than a hint. A product with only a single fault could be have a reliability of zero if that fault results in a failure on every execution, and there is no reason to suppose that a test suite designed for finding faults is at all representative of actual usage and failure rate.

From the experience of many previous projects, Chipmunk has empirically determined that in their organization, it is fruitful to begin measuring reliability when debug testing is yielding less than one fault ("bug") per day of tester time. For some application domains, Chipmunk has gathered a large amount of historical usage data from which to define an operational profile, and these profiles can be used to generate large, statistically valid sets of randomly generated tests. If the sample thus tested is a valid model of actual executions, then projecting actual reliability from the failure rate of test cases is elementary. Unfortunately, in many cases such an operational profile is not available.

Chipmunk has an idea of how the web sales facility will be used, but it cannot construct and validate a model with sufficient detail to obtain reliability estimates from a randomly generated test suite. They decide, therefore, to use the second major approach to verifying reliability, using a sample of real users. This is commonly known as *alpha testing* if the tests are performed by users in a controlled environment, observed by the development organization. If the tests consist of real users in their own envi-

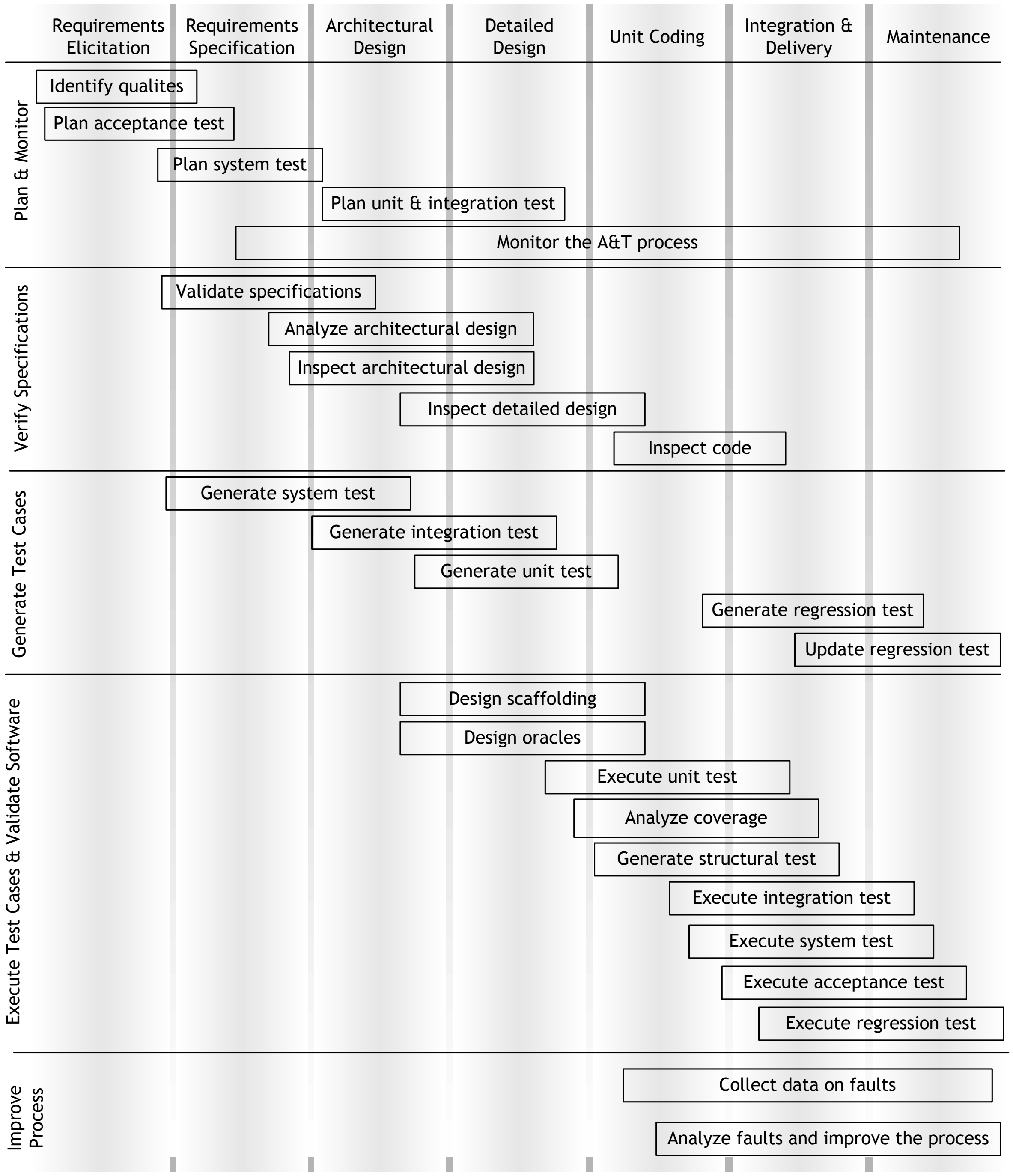

Figure 1.1: Main analysis and testing activities through the software life cycle.

ronment, performing actual tasks without interference or close monitoring, it is known as *beta testing*. The Chipmunk team plans a very small alpha test, followed by a longer beta test period in which the software is made available only in retail outlets. To accelerate reliability measurement after subsequent revisions of the system, the beta test version will be extensively instrumented, capturing many properties of a usage profile.

1.6 How can we assure the quality of successive releases?

Software test and analysis does not stop at the first release. Software products often operate for many years, frequently much beyond their planned life cycle, and undergo many changes. They adapt to environment changes, e.g., introduction of new device drivers, evolution of the operating system, and changes in the underlying data base. They also evolve to serve new and changing user requirements. Ongoing quality tasks include test and analysis of new and modified code, re-execution of system tests, and extensive record-keeping.

Chipmunk maintains a database for tracking problems. This database serves a dual purpose of tracking and prioritizing actual, known program faults and their resolution and managing communication with users who file problem reports. Even at initial release, the database usually includes some known faults, because market pressure seldom allows correcting all known faults before product release. Moreover, “bugs” in the database are not always and uniquely associated with real program faults. Some problems reported by users are misunderstandings and feature requests, and many distinct reports turn out to be duplicates and are eventually consolidated.

Chipmunk designates relatively major revisions, involving several developers, as “point releases,” and smaller revisions as “patch level” releases. The full quality process is repeated in miniature for each point release, including everything from inspection of revised requirements to design and execution of new unit, integration, system, and acceptance test cases. A major point release is likely even to repeat a period of beta testing.

Patch level revisions are often urgent for at least some customers. For example, a patch level revision is likely when a fault prevents some customers from using the software, or when a new security vulnerability is discovered. Test and analysis for patch level revisions is abbreviated, and automation is particularly important so that a reasonable level of assurance can be obtained with very fast turnaround. In particular, when fixing one fault, it is all too easy to introduce a new fault or re-introduce faults that have occurred in the past. Chipmunk maintains an extensive suite of *regression tests* to detect these. The Chipmunk development environment supports recording, classification, and automatic re-execution of test cases. Each point release must undergo complete regression testing before release, but patch level revisions may be released with a subset of regression tests that run unattended overnight. Developers add new regression test cases as faults are discovered and repaired.

1.7 How can the development process be improved?

As part of an overall process improvement program, Chipmunk has implemented a quality improvement program. In the past, the quality team encountered the same defects in project after project. The quality improvement program tracks and classifies faults to identify the human errors that cause them and weaknesses in test and analysis that allow them to remain undetected.

Chipmunk quality improvement group members are drawn from developers and quality specialists on several project teams. The group produces recommendations that may include modifications to development and test practices, tool and technology support, and management practices. The explicit attention to buffer overflow in networked applications at Chipmunk is the result of failure analysis in previous projects.

Fault analysis and process improvement comprise four main phases: defining the data to be collected and implementing procedures for collecting it; analyzing collected data to identify important fault classes; analyzing selected fault classes to identify weaknesses in development and quality measures; and adjusting the quality and development process.

Collection of data is particularly crucial, and often difficult. Earlier attempts by Chipmunk quality teams to impose fault data collection practices were a dismal failure. The quality team possessed neither carrots nor sticks to motivate developers under schedule pressure. An overall process improvement program undertaken by the Chipmunk software division provided an opportunity to better integrate fault data collection with other practices, including the normal procedure for assigning, tracking, and reviewing development work assignments. Quality process improvement is distinct from the goal of improving an individual product, but initial data collection is integrated in the same bug tracking system, which in turn is integrated with the revision and configuration control system used by Chipmunk developers.

The quality improvement group defines the information that must be collected for faultiness data to be useful, and the format and organization of that data. Participation of developers in designing the data collection process is essential to balance the cost of data collection and analysis with its utility, and to build acceptance among developers.

Data from several projects over time are aggregated and classified to identify classes of faults that are important because they occur frequently, because they cause particularly severe failures, or because they are costly to repair. These faults are analyzed to understand how they are initially introduced and why they escape detection. The improvement steps recommended by the quality improvement group may include specific analysis or testing steps for earlier fault detection, but they may also include design rules and modifications to development and even management practices. An important part of each recommended practice is an accompanying recommendation for measuring the impact of the change.

Summary

The quality process has three distinct goals: Improving a software product (by preventing, detecting, and removing faults), assessing the quality of the software product (with respect to explicit quality goals), and improving the long-term quality and cost-effectiveness of the quality process itself. Each goal requires weaving quality assurance and improvement activities into an overall development process, from product inception through deployment, evolution, and retirement.

Each organization must devise, evaluate, and refine an approach suited to that organization and application domain. A well-designed approach will invariably combine several test and analysis techniques, spread across stages of development. An array of fault detection techniques are distributed across development stages so that faults are removed as soon as possible. The overall cost and cost-effectiveness of techniques depends to a large degree on the extent to which they can be incrementally re-applied as the product evolves.

Further Reading

This book deals primarily with software analysis and testing to improve and assess the dependability of software. That is not because qualities other than dependability are unimportant, but rather because they require their own specialized approaches and techniques. We offer here a few starting points for considering some other important properties that interact with dependability. Norman's *Design of Everyday Things* [Nor02b] is a classic introduction to design for usability, with basic principles that apply to both hardware and software artifacts. A primary reference on usability for interactive computer software, and particularly for web applications, is Nielsen's *Design Web Usability* [Nie99]. Bishop's text [Bis02] is a good introduction to computer security. The most comprehensive introduction to software safety is Leveson's *Safeware* [Lev95].

Exercises

1.1. Philip has studied "just-in-time" industrial production methods, and is convinced that they should be applied to every aspect of software development. He argues that test case design should be performed just before the first opportunity to execute the newly designed test cases, never earlier. What positive and negative consequences do you foresee for this just-in-time test case design approach?

1.2. A newly hired project manager at Chipmunk questions why the quality manager is involved in the feasibility study phase of the project, rather than joining the team only when the project has been approved, as at the new manager's previous company. What argument(s) might the quality manager offer in favor of her involvement in the feasibility study?

1.3. Chipmunk procedures call for peer review not only of each source code module, but also of test cases and scaffolding for testing that module. Anita argues that inspecting test suites is a waste of time; any time spent on inspecting a test case designed to detect a particular class of fault could more effectively be spent inspecting the source code to detect that class of fault. Anita's project manager, on the other hand, argues that inspecting test cases and scaffolding can be cost-effective when considered over the whole lifetime of a software product. What argument(s) might Anita's manager offer in favor of this conclusion?

1.4. The spiral model of software development prescribes sequencing incremental prototyping phases for risk reduction, beginning with the most important project risks. Architectural design for testability involves, in addition to defining testable interface specifications for each major module, establishing a build order that supports thorough testing after each stage of construction. How might spiral development and design for test be complementary or in conflict?

1.5. You manage an on-line service that sells down-loadable video recordings of classic movies. A typical down-load takes one hour, and an interrupted down-load must be restarted from the beginning. The number of customers engaged in a download at any given time ranges from about 10 to about 150 during peak hours. On average, your system goes down (dropping all connections) about two times per week, for an average of three minutes each time. If you can double availability or double mean time between failures, but not both, which will you choose? Why?

1.6. Having no a priori operational profile for reliability measurement, Chipmunk will depend on alpha and beta testing to assess the readiness of their on-line purchase functionality for public release. Beta testing will be carried out in retail outlets, by retail store personnel and then by customers with retail store personnel looking on. How might this beta testing still be misleading with respect to reliability of the software as it will be used at home and work by actual customers? What might Chipmunk do to ameliorate potential problems from this reliability mis-estimation?

1.7. The junior test designers of Chipmunk Computers are annoyed by the procedures for storing test cases together with scaffolding, test results and related documentation. They blame the extra effort needed to produce and store such data for delays in test design and execution. They argue for reducing the data to store to the minimum required for re-executing test cases, eliminating details of test documentation and limiting test results to the information needed for generating oracles.

What argument(s) might the quality manager use to convince the junior test designers of the usefulness of storing all this information?

Chapter 2

A Framework for Test & 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 and combine from 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,[1] and that proposed solution may or may not achieve its goals. Moreover, specifications are written by people,

Δ validation

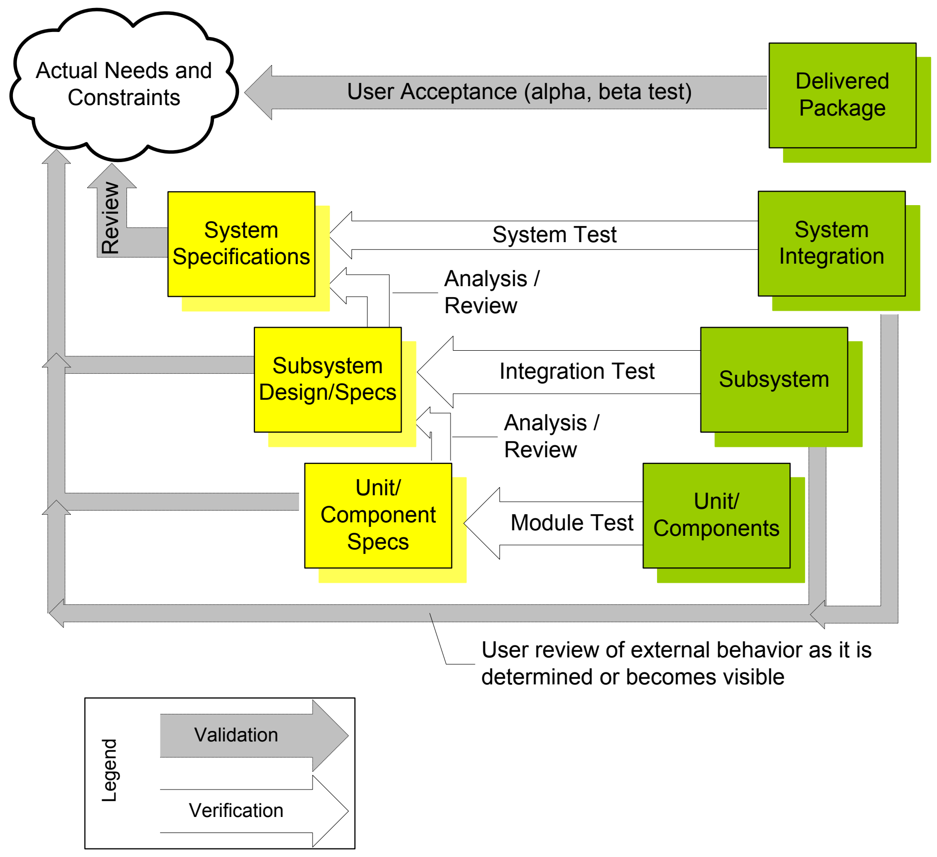

Figure 2.1: Validation activities check work products against actual user requirements, while verification activities check consistency of work products.

and therefore contain mistakes. A system that meets its actual goals is *useful*, while a system that is consistent with its specification is *dependable*.

Δ dependable

Δ verification

"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 non-trivial 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 further discussed 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, i.e., 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 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.

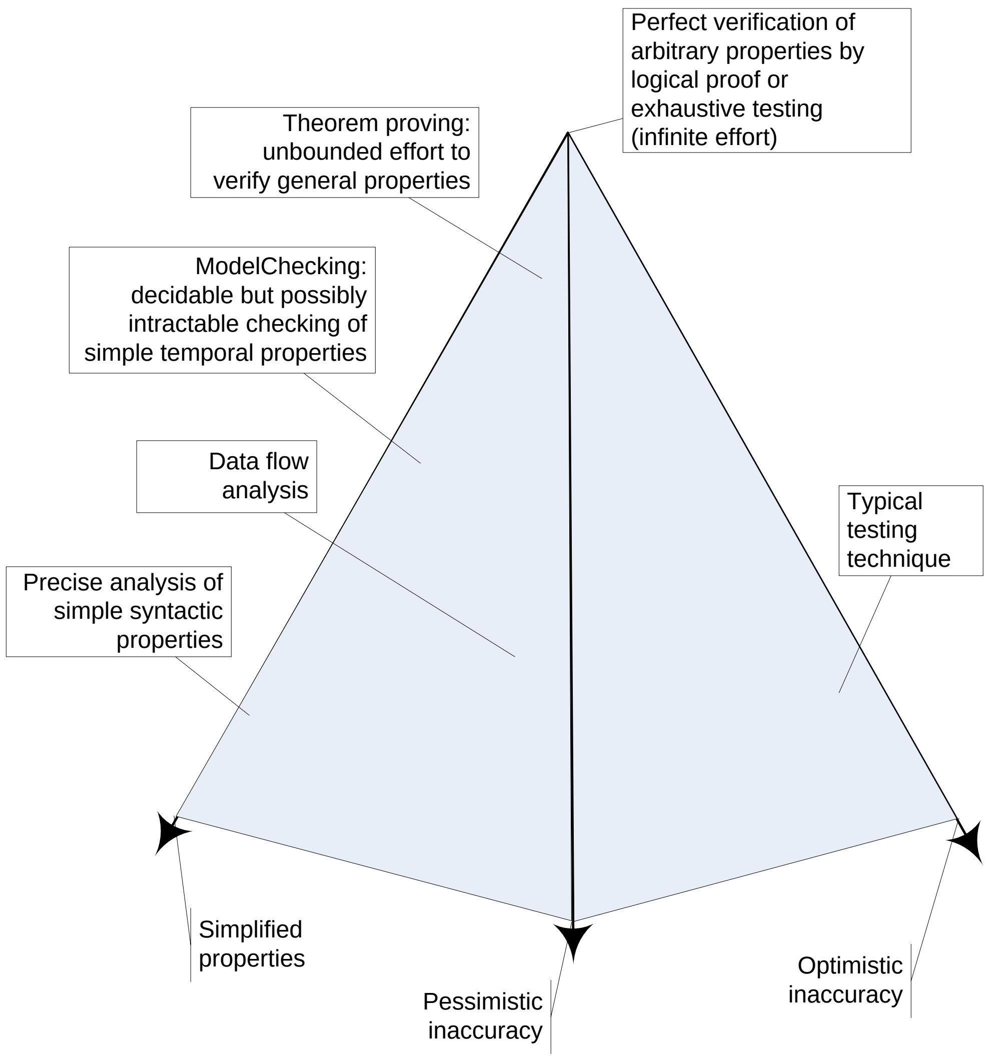
Perfect verification of arbitrary properties by logical proof or exhaustive testing (infinite effort)
Theorem proving: unbounded effort to verify general properties
ModelChecking: decidable but possibly intractable checking of simple temporal properties
Data flow analysis
Precise analysis of simple syntactic properties
Typical testing technique
Simplified properties
Pessimistic inaccuracy
Optimistic inaccuracy

Figure 2.2: Verification tradeoff dimensions

Program testing is a verification technique, and is as vulnerable to undecidability as other techniques. Exhaustive testing, i.e., 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:

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

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 which 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, i.e., 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 which admits only optimistic inaccuracy. An analysis which 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:

```

int i, sum;
int first=1;
for (i=0; i<10; ++i) {
    if (first) {
        sum=0; first=0;
    }
    sum += i;
}
```

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, i.e., 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 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 online 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 which 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, i.e., 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 which 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 non-determinism, 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, etc., 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 $\frac{1}{4}$ 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 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 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 which 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 was 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' 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 tradeoffs 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 current version of the RodentOS operating system (an (imaginary) operating system of Chipmunk), and requires that the new version of RodentOS should not annoy the user before allowing to start working.

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

2.2. Let us considered a simple specification language *SL* that describes systems diagrammatically in terms of `functions`, that represent data transformations and correspond to nodes of the diagram, and `flows`, that 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, e.g., 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 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 programmer could maintain very coarse-grain locking, preventing any interleaving of accesses to the shared data structure, even which 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 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 of Structured Analysis may have noticed that *SL* resembles a simple Structured Analysis specification

Compare these approaches by their relative positions 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 re-execute the test cases designed for the former versions of the program. Re-executing test cases designed for previous versions is called regression testing. When testing large complex programs, the number of regression test cases may be too large, and thus test designers may need to select a subset of test cases among all test cases that can be re-executed on the new version.

Subsets of test cases can be selected according to different criteria. An interesting property of criteria for selecting subset of regression test cases is not to exclude any test that may reveal a possible fault.

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