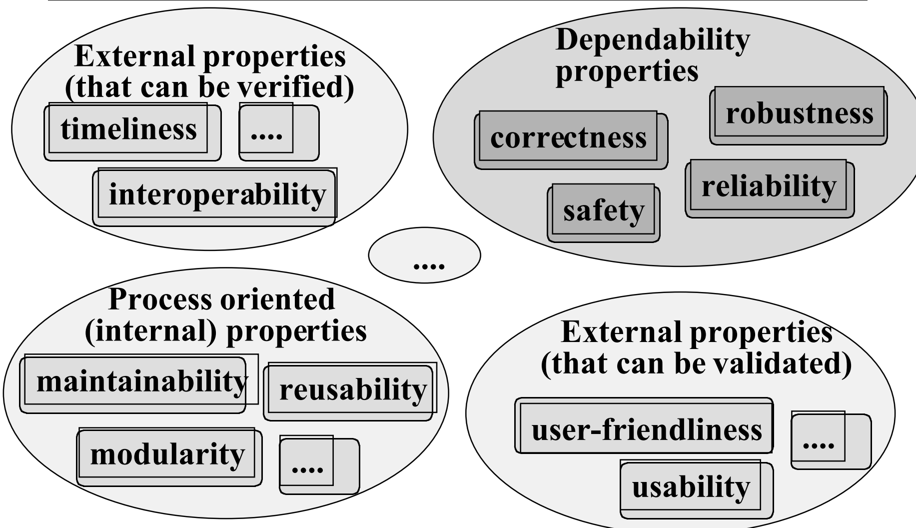
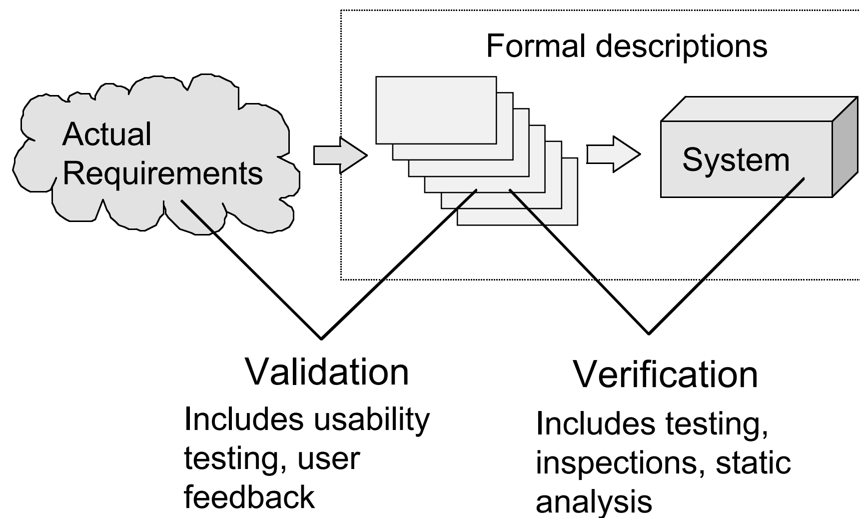
OMSE 525/ CIS 610
Software Qualities



Software is characterized by many properties, that are measured and studied with different techniques. This tutorial focuses on techniques for assessing dependability properties, i.e., correctness, robustness, safety, and reliability.

Validation vs. Verification



CIS 160 SQA / F99 / © Michal Young

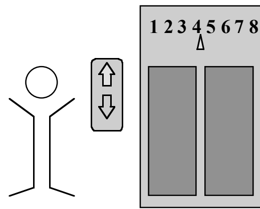
We use the term *verification* to describe a check for consistency between two formal representations. We will use this term in a wide sense: for example, testing is verification when it is used to check program behavior against a specification.

While we cannot conclusively show correspondence between an informal representation and a formal representation, there are several things we can do to check either a specification or an actual system against even the most informal requirements. These activities are called *validation*, as versus *verification*. In the terms introduced by Boehm, verification is "*building the system right*" while validation is "*building the right system*." Both are important, but they require quite different measures.

Validation is largely subjective (although it may also include objective measures, such as usability testing with objective performance measures). Verification is objective, in the sense that a program behavior is unambiguously permitted or not permitted by the specification (this is what we mean by the specification being precise.) Verification of an implementation against a specification is valuable only to the extent that we have done a good job of validating the specification against intentions. In practice these are usually not sequential steps, but are intertwined.

Verification or validation depend on the specification

Example: elevator response

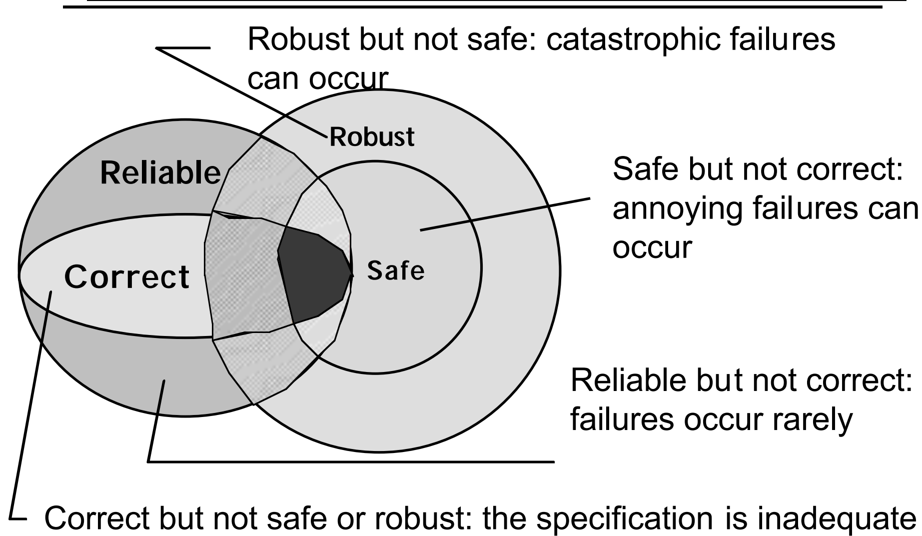


... if a user press a request button at floor i, an available elevator must arrive at floor i soon... ⇒ this property can be validated,NOT verified (SOON depends on the feeling of the user)

... if a user press a request button at floor i, an available elevator must arrive at floor i within 30 seconds... ⇒ this property can be verified (30 seconds is a precise quantity)

Verification and validation depend on the way the properties are defined. Qualitative specifications often require the validation of an external user, while quantitative properties can often be verified independently.

Dependability Properties



A program is reliable but not correct when failures occur rarely. (A “failure” is any behavior that is not permitted by the specification.) As discussed on the previous page, “rarely” may depend on factors other than the program, including the way a program is used and even its environment. These factors are often not described explicitly, in which case they implicitly mean “under normal conditions” or “in typical usage” (whether or not such a thing makes sense).

A program may be correct without being safe or robust if the specification is inadequate, in the sense that the specification does not rule out some undesirable behaviors. The phrase “that’s not a bug” is sometimes associated with inadequate specifications (the behavior in question is not a bug because the specification doesn’t prohibit it, but it *ought* to be a bug.)

A particularly common way in which a program can be correct (or at least reliable) without being safe or robust is when the specification is only partly defined. For example, a specification could describe what the program should do when an existing file is opened for reading, but not say what the program should do if the file doesn’t exist. In principle, the program can be correct regardless of *what* it does in this case — even if it opens a completely different file — but this would not be a very robust (and possibly not a safe) behavior.

Correctness

- Correctness is a consistency relation
  - Meaningless without a specification
- Absolute correctness is nearly impossible to establish
  - seldom a cost-effective goal for a non-trivial software product
- Approximations are difficult to define and measure
  - e.g., faults per 1000 lines of code

A program is correct if it obeys its specification. (There are many alternative ways of formalizing this, e.g., we can consider a specification to denote a set of acceptable implementations and say that a program is correct if it belongs to that set, called the "specific and set.")

It is meaningless to talk about correctness of a program without reference to a specification. And yet we seem to do this all the time ... If a program crashes, or corrupts a file, or just creates garbage, I say that it is unreliable (and therefore incorrect), although we have never seen a specification for the program in question. How can we reconcile this common usage with the definition of correctness as a consistency relation? When we talk of bugs and reliability in this manner, we are appealing to implicit and usually informal specifications. Later we'll make a distinction between verification of explicit, formally specified properties and validation of other properties.

Establishing correctness — i.e., "proving" that a program is correct — is almost never a practical, cost-effective goal. Algorithms can be proved correct, as can protocols, and sometimes small but crucial bits of code can be proved correct; all these are useful, but they are not the same as proving correctness of a whole system. So, can we get an "engineering approximation" of correctness, e.g., "this program has fewer than 1 bug per 1000 lines of code?" Such approximations are sometimes used, but they are difficult to define unambiguously or measure precisely.

Correctness and Specification

- Example: for a "correct" language interpreter, we require
  - Precise grammar (e.g., BNF)
  - Precise semantics
    - Hoare-style proof rules, or denotational semantics, or ...
    - or operational semantics from model implementation (bugs and all)
- Few application domains are well enough understood for full specifications

Demonstrating correctness depends as much on our ability to specify intended behavior as on our ability to verify consistency with actual behavior. Our ability to specify, in turn, depends a great deal on how well we have built up a "domain theory."

Language interpreters are a good example of the relation between verification and the existence of a mature domain theory. There is a mature theory of syntactic structure, and formal notations (regular expressions and BNF) for specifying exactly the texts that a language interpreter should accept. Moreover, this specification method is associated with a well-developed theory of parsing, which makes it easy to verify that a parser accepts the language specified by a grammar. The same theory allows us to (mostly) use "proof by construction," automatically deriving a parser from the grammar.

The theory of programming language semantics, in contrast to syntax, is not nearly adequate for specifying the intended meaning of programming languages. A formal semantic specification for a typical programming language (say, Java) would be much larger than the corresponding syntactic specification. In practice, it is a book — semi-formal, and despite a lot of work, almost certainly ambiguous and incomplete. Since the only available specification is informal, there is no way to obtain a formal demonstration of correctness.

Mature domain theories exist for several domains, but far more application domains lack the kind of formalization that would be required to produce concise formal specifications.

- Quantifiable: Mean time between failures, availability, etc.
  - describes behavior, not the product itself, e.g., fewer bugs ° higher reliability
- Still relative to a specification
  - but perhaps a simple one
- Relative to a usage profile
  - often difficult to obtain in advance
  - may not be static, or even well-defined
    - how reliable are programs with the year 2000 bug?

So we can’t (usually) achieve correctness ... who cares? Our cars, televisions, and even our airplanes don’t work correctly 100% of the time. What we really care about is reliability, right? *(Well, not quite ... more on that later.)*

Reliability is a way of statistically approximating correctness. Reliability can be stated in different ways. Classical reliability is often stated in terms of time, e.g., mean time between failures (MTBF) or availability (likelihood of correct functioning at any given time). Time-based reliability measures are often used for continuously functioning software (e.g., an operating system or network interface), but for other software “time” is often replaced by a usage-based measure (e.g., number of executions).

For example, mean time between failures (MTBF) is a statement about the likelihood of failing before a given point in time (but “time” may be measured in number of uses or some other way). Availability is the likelihood of correct functioning at any particular point in time.

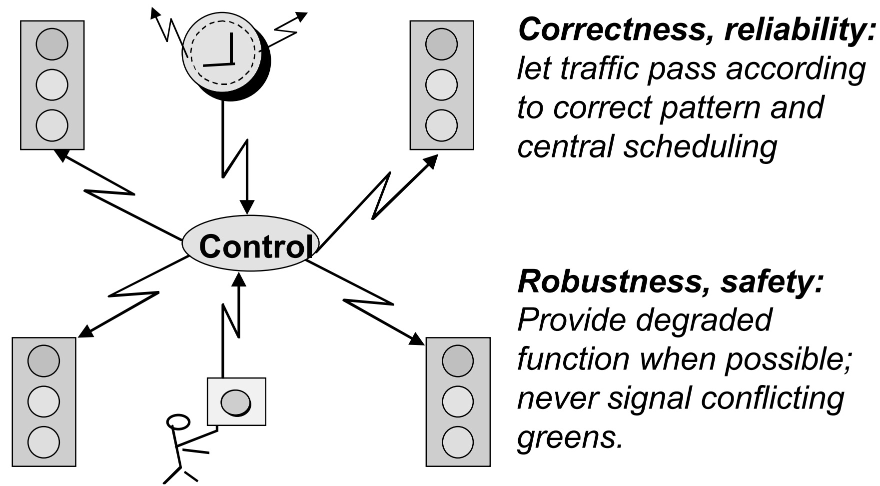
Reliability describes the behavior of a program, which may not be correlated to structural measures of quality. For example, a program with 1 fault (bug) per 1000 lines of code may be less reliable than another program with 5 faults per 1000 lines of code, depending on how often those faults result in program failures. How often a fault causes a failure depends, in turn, on how a program is used. Therefore, reliability is relative to a usage profile; in fact, a program may be highly reliable when used by one group of users in one way, and very unreliable when used by another group of users in another way. Accurate usage profiles can be obtained for some kinds of embedded software, or when one program replaces another in an existing domain (e.g., we have good usage profiles for telephone switching systems). For novel applications, it is difficult to obtain accurate usage profiles in advance. In some cases, reliability may not even be well-defined. Consider: What is the reliability of a payroll program that runs correctly 100% of the time until Jan 1, 2000, and then crashes on every use?

Robustness

- Beyond correctness: A property of both specifications and implementations
  - A robust system has specified behavior in unexpected and severe conditions
- Orthogonal to reliability
  - concerned with *unusual* conditions
- Often concerned with partial functionality:
  - graceful degradation

A system is robust if it acts reasonably in severe or unusual conditions. It is not possible to give a precise definition of robustness, but one characteristic of robust systems is that their specifications include "desired reactions to undesirable situations" [see. Henninger & Parnas].

Robustness is often (but not always) concerned with partial functionality, also called "graceful degradation." An example of this is a phone system, which distinguishes "plain old telephone service" (abbreviated POTS) from advanced services like call waiting, call forwarding, etc. The phone system is designed to keep POTS operational even when advanced services cannot be maintained.



We will illustrate the relation between correctness and reliability on the one hand, and robustness and safety on the other, using the example of a traffic light at a four-way intersection with pedestrian crossings. We'll imagine a sophisticated traffic light system with timing partly controlled by a central system, which sets the pattern differently according to traffic patterns at different times of day. For purposes of the example, we'll consider only the lights at a single intersection (i.e., we'll treat the central scheduling facility as an external entity).

A goal of the traffic light system is to let traffic pass as efficiently as possible, according to the timing pattern set by the central scheduler. To be robust, it should also provide some minimal function even when full functionality is not possible. Above all, it must avoid the hazard of collisions between cars or between cars and pedestrians.

Software Safety

Preventing bad things from happening
(robustness/ negative specifications)

- Not concerned with maintaining function
  - Simple, incomplete, often redundant specifications of hazard prevention
- Adaptation of system safety
  - An established engineering field
- May depend on reliability, or conflict with it

There is an established field of system safety engineering concerned with preventing unsafe behavior; software safety is an extension and adaptation of system safety principles and techniques to software engineering. System safety begins with hazard analysis; hazards are the unsafe situations or behaviors that we must avoid. The parts of a specification concerned with safety are not concerned at all with maintaining functionality, only with avoiding these hazards.

Safety specifications are typically

- Simple: Even if the system as a whole is very complex, the safety properties should be very simple so that they are easy to establish and verify.
- Incomplete: They don't specify all the behavior of a system, which is how simplicity can be achieved.
- Redundant: Even if a safety property should follow logically from correct functional behavior, safety properties are specified independently (just to be sure!).

Safety vs. Reliability

- Interdependent when safety depends on continued (perhaps degraded) function
  – example: flight control of fly-by-wire aircraft
- Conflicting when function does not contribute to safety
  – example: an automobile that does not start is safe, but unreliable
  – example: the safest torpedo never explodes

In some cases, safety will depend at least partly on reliability. For example, a fly-by-wire aircraft such as the Airbus A320 or the Boeing 777 is safe only if the avionics software provides at least some level of functionality. In many cases, though, safety can conflict with reliability, because the "safe state" of a system is a non-functional state. For example, a nuclear plant control system is safest (but not very reliable) if it shuts down at the least hint of irregularity. An automobile that tests its turn signal lights and refuses to start if any are burnt out would be safer, but we would probably not accept the reduction in reliability.

Nancy Leveson, a pioneer in software safety research, often tells a story of her experience as a consultant to a company developing a torpedo. The intended behavior of a torpedo is to reach an enemy ship and explode; the relevant hazard is to return to the firing ship and explode. After many measures were taken to avoid the hazard, the torpedo was tested in a lake. It consistently floated to the bottom and disarmed itself. It was a very safe torpedo — but very unreliable.

Why negative properties?

- Formal descriptions are incomplete
  - positive specifications may not rule out dangerous behaviors
- Implementations are imperfect
  - and more complexity => more errors
  - a redundant, simple specification may facilitate stronger assurance of critical properties

Why do we explicitly state negative properties (what the software should not do), rather than depending on a statement of the positive properties (what the software should do) and showing that they imply the safety conditions? In a word, fallibility. We make mistakes. Formal descriptions of the intended behavior of software systems are complex, and we make mistakes. We may think they positive properties rule out the hazardous behavior, but we may be wrong. A simple, redundant statement of behavior to be avoided reduces the likelihood that we fail to adequately specify it.

Implementations, too, are complex and imperfect. Where there is complexity, there will be errors. We want to have much higher assurance of critical safety properties than of overall correct functioning, and the only way we can achieve this is to keep safety properties extremely simple. There should be equally simple measures in the system (software and/or hardware) to ensure these properties.