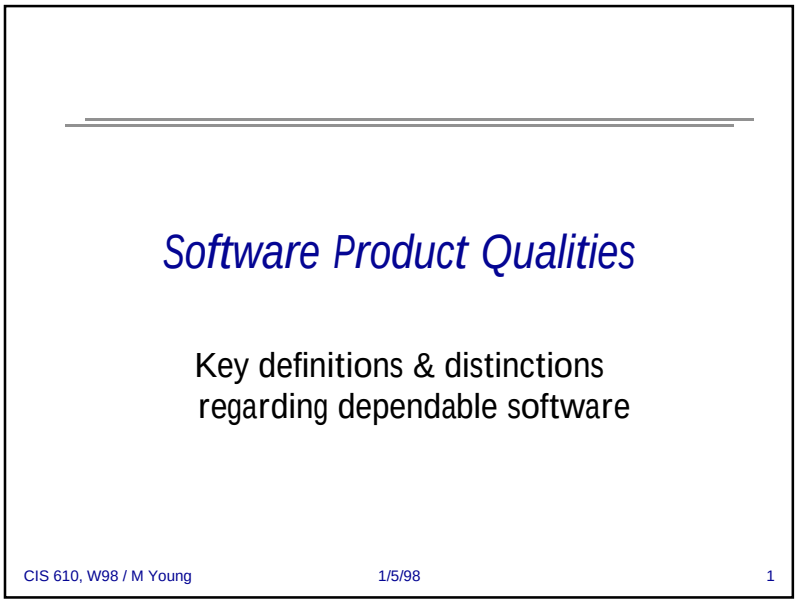


Since the ultimate purpose of testing and analysis is to produce higher quality software, we must begin by considering the quality itself.

For the purposes of this presentation, we will limit ourselves to dependability properties such as correctness, reliability, and safety, and largely ignore other qualities such as usability, maintainability, and efficiency. That is not because other properties are unimportant, but because the techniques needed for ensuring them are quite different from techniques needed for improving dependability.

Dependability Properties

- Correctness
  - Consistency of implementation with specification
- Reliability
  - Likelihood of correct functioning
- Robustness
  - Acceptable behavior in unusual circumstances
- Safety
  - Absence of unacceptable behaviors

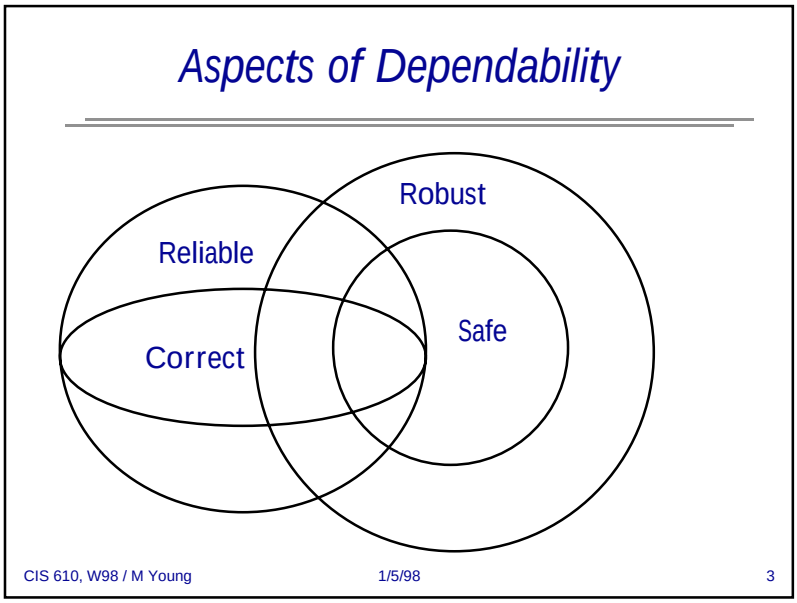
CIS 610, W98 / M Young 1/5/98 2

We will treat "dependability" as a general (and necessarily informal) term, encompassing the whole variety of things we might mean when we say that a piece of software is dependable. We will try to be more precise about four particular kinds of dependability properties: Correctness, reliability, robustness, and safety.

A program is "correct" if it is consistent with its specification, i.e., if it does exactly what the specification says it must do. It is therefore a consistency relation between two things, the specification and the program. It is impossible to say whether a program is correct in the absence of a specification, although the specification may be informal or implicit.

Whereas correctness is an absolute property (a program is either correct with respect to a particular specification, or it is not), reliability is probabilistic measure: What is the likelihood that the program will obey the specification? Reliability is familiar from hardware. We don't ask whether a 1997 Toyota Camry is correct, we ask whether it is dependable (e.g., what is the likelihood that it will get me to the grocery store and back?)

Robustness is concerned with acceptable behavior in unusual circumstances. Unlike correctness, robustness is partly a property of a specification; if the specification is not good, the product may be correct but not robust. Safety is an important sub-category of robustness properties, namely, properties that prescribe what the software *must not* do rather than what it must do.

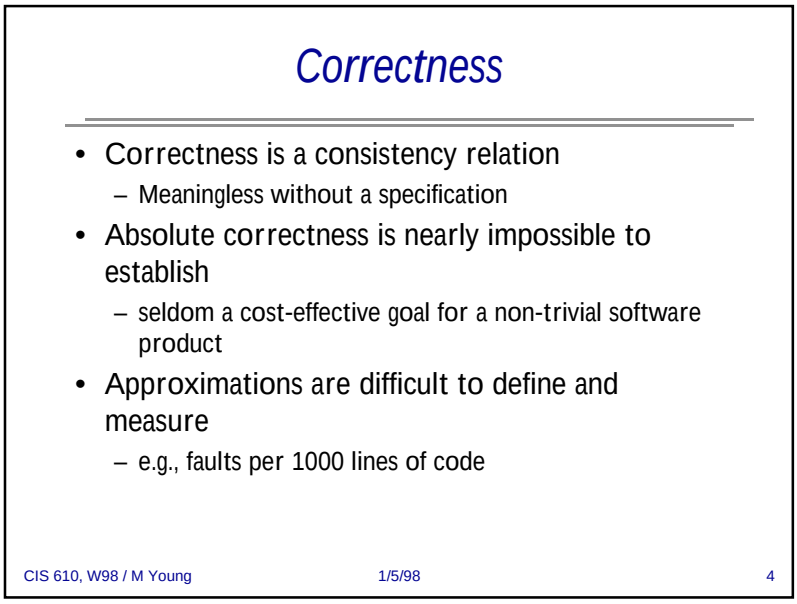


This diagram illustrates the relations among the four kinds of dependability property we are considering.

A program that is “correct” is, by definition, 100% reliable. Thus the correct programs (or more precisely, the correct (spec,program) pairs) are a subset of the reliable programs (those that exceed some statistical criterion of reliability). However, a correct or reliable program need not be robust or safe.

Safety properties are a kind of robustness property, so we draw the safe region within the robust region. Correctness implies safety only when all the needed safety properties have been included in the specification.

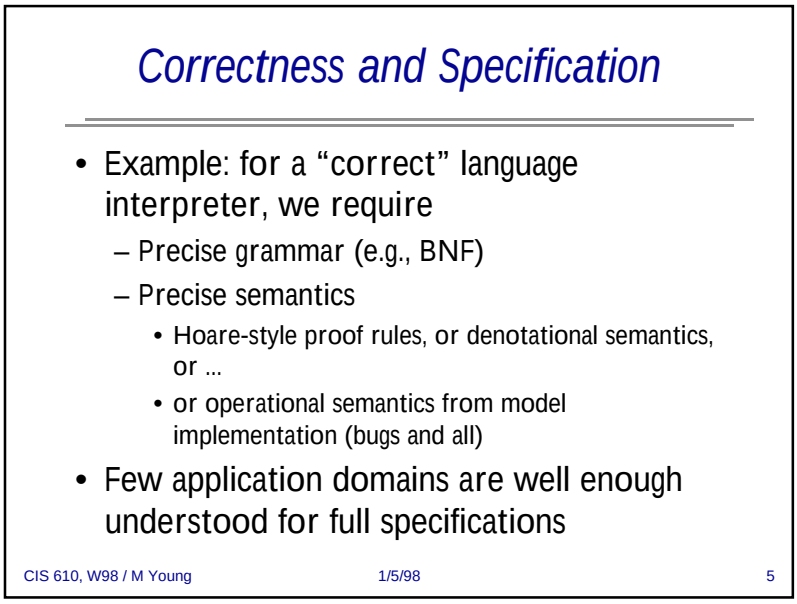
A program need not be correct or reliable to be safe. In fact, a perfectly useless piece of software can be safe, as long as it doesn’t cause any harm. As we will see, reliability properties are sometimes in conflict with safety and robustness.



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 "specificand 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.

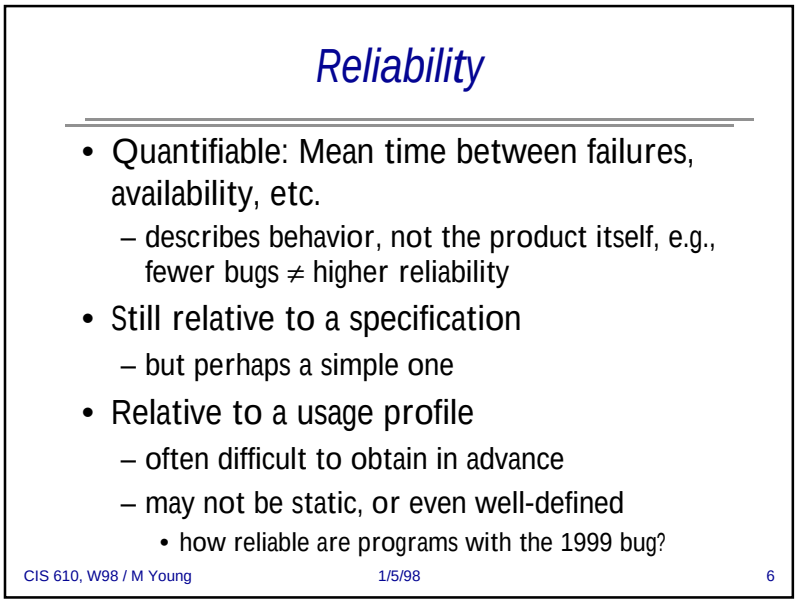


Demonstrating correctness depends as much on our ability to specify intended behavior as on our ability to verify consistency with actual behavior. Our ability to specify, in turn, depends a great deal on how well we have built up a "domain theory."

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.

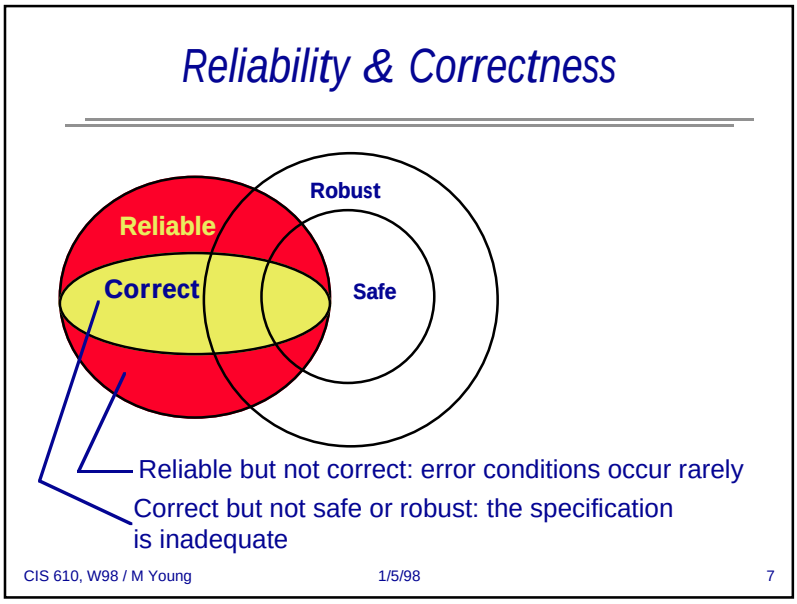


So we can't (usually) achieve correctness ... who cares? Our cars, televisions, and even our airplanes don't work correctly 100% of the time. What we really care about is reliability, right? *(Well, not quite ... more on that later.)*

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?



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.

Reliability Examples

- Telephone switch availability
  - well-defined: minutes of down-time per year
  - based on accessible, stable usage profiles
- Word processor reliability
  - Crashes per hour? Per session? Per document?
  - Inaccessible, poorly defined usage profile (several classes of users and use); may be meaningless

CIS 610, W98 / M Young 1/5/98 8

Sometimes reliability is an appropriate way to talk about dependability, and sometimes not. A telephone switch is an example for which reliability is an appropriate measure of dependability, and for which reliability measurement is feasible. The reliability of a word processor, on the other hand, is difficult to characterize.

It is natural to characterize the reliability of a telephone switch by availability. A typical reliability measure would be "no more than 5 minutes of down-time per year." Note that the complement of down time, "time in which the switch is operating normally," is a very simple kind of specification. We might have a different reliability measure for the same system using a more demanding specification, e.g., probability that any correctly dialed phone call will be connected. (A phone call may fail to be connected because the switch capacity is exceeded, although the switch is "up" because it is maintaining other connections.) Since we have already noted that reliability is relative to both a specification and a usage profile, there is no conceptual problem with having several different reliability measures for the same system.

One of the reasons that reliability is an appropriate way of characterizing dependability of a phone switch system is that we can obtain reliability measures for usage profiles that correspond closely to real use. Although phone use is changing (consider the growth of cellular traffic), it is sufficiently for "reliability" to be meaningful, and there is enough historical data and experience to obtain reliability estimates for a new system even before it is placed in service, through statistical testing.

Unfortunately, the situation for a word processor is quite different. To begin, how should we measure? Not by the hour, probably, but perhaps by the keystroke or operation. More seriously, if the word processor contains any new features, it is very unlikely that we have an accurate model of how they will be used.

Fault density: a static measure

- Reliability is about execution; faults are in code (a static measure)
- Faults/ kLOC
  - May be counted, e.g., in acceptance test phase
- Problems
  - Count reflects only *known* faults, not total
  - Unclear what is a single fault
  - Relation to observable quality is not clear

CIS 610, W98 / M Young 1/5/98 9

Sometimes it is useful to have a static measure, like the number of bugs in a program. This is sometimes measured (or estimated) as "fault density," typically as the number of bugs per 1000 lines of code. (1000 lines of code is commonly abbreviated as kLOC, pronounced "kay-loke").

We intuitively feel that "good" software should have fewer bugs than "bad" software, but there is no necessary relation between fault density and reliability. In principle, if there is a failure (bad behavior), there must be at least one fault (bug in the code), but a single fault could cause many failures or none at all.

Fault density has several problems as a statistical approximation to correctness, besides the fact that it does not necessarily correlate to reliability. First, and most obvious, we can only count the faults that we have found, not the faults remaining in the code. (Remaining faults are sometimes estimated from faults that have been found.) Second, it isn't always clear how to count, i.e., what constitutes a single fault.

Given these problems, why bother with fault density at all? In fact, fault density is a completely meaningless statistic for the user; for all its problems, reliability is at least observable. However, fault density can be a useful measure internally. For example, there may be points in the development process in which there is not enough running code to obtain reliability measures, but fault density measures may be obtained from inspections and unit and subsystem testing. Together with historical data from similar projects, this can provide useful feedback for gauging progress and predicting problems later in development.

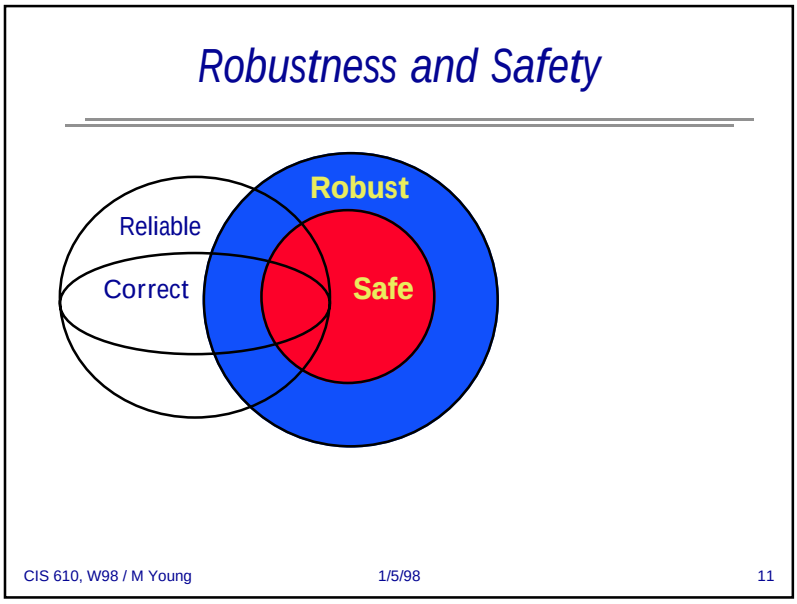
Robustness

- Beyond correctness: A property of specifications and implementations both
  - 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

CIS 610, W98 / M Young 1/5/98 10

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" [cf. Henninger & Parnas].

Robustness is often (but not always) concerned with partial functionality, also called "graceful degradation." An example of this is the 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. *(More ahead ...)*



Robustness is not closely related to reliability or correctness. In particular, a system may be very reliable without being robust at all. As a simple example, consider a sorting program that works perfectly for data sets with less than 64,000 items. If the data set contains more than 64,000 items, it silently discards part of the data (i.e., without any warning to the user). This is a very non-robust behavior, but the program may still be "correct" if the specification states that input data sets must have fewer than 64,000 items. If the specification does not place a limit on the data set size, then the program is incorrect, but it may still be quite reliable if users seldom sort large data sets.

Safety is a sub-category of robustness specifically concerned with avoiding certain very bad behaviors. The undesired outcomes are called "hazards," and safety engineering is concerned with identifying and preventing hazards. Sometimes this literally means "human safety," as when for example the hazard is a nuclear reactor meltdown or a robot crushing a human worker. In other cases we may identify less catastrophic but still undesirable hazards to avoid, such as trashing the file system.

Failure and robustness

- Fail soft: Maintaining partial functionality
  - example: "POTS" service when extra phone services unavailable
- Fail safe: Avoiding unsafe behavior
  - may not provide any useful functionality
  - establish a "safe" state
  - example: engine shut-off in case of self-diagnosed error

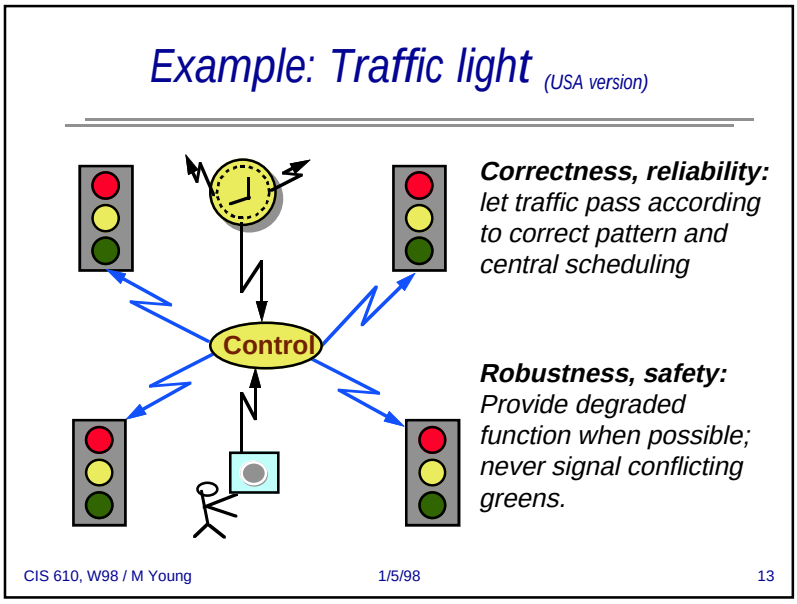
CIS 610, W98 / M Young 1/5/98 12

Recall that robustness is associated with specifications of "desirable reactions to undesirable situations." One "undesirable situation" is failure of a software system (or of a combined hardware/software system). Two particular kinds of robust reactions to failure are commonly distinguished: fail soft and fail safe.

"Fail soft" is the same as the aforementioned "graceful degradation," i.e., maintaining some level of useful functionality despite partial system failure. We have already mentioned the telephone system in this regard. Another example of "fail soft" is an avionics system that maintains enough functionality to allow the pilot to land safely, although it can no longer provide more advanced flight functions.

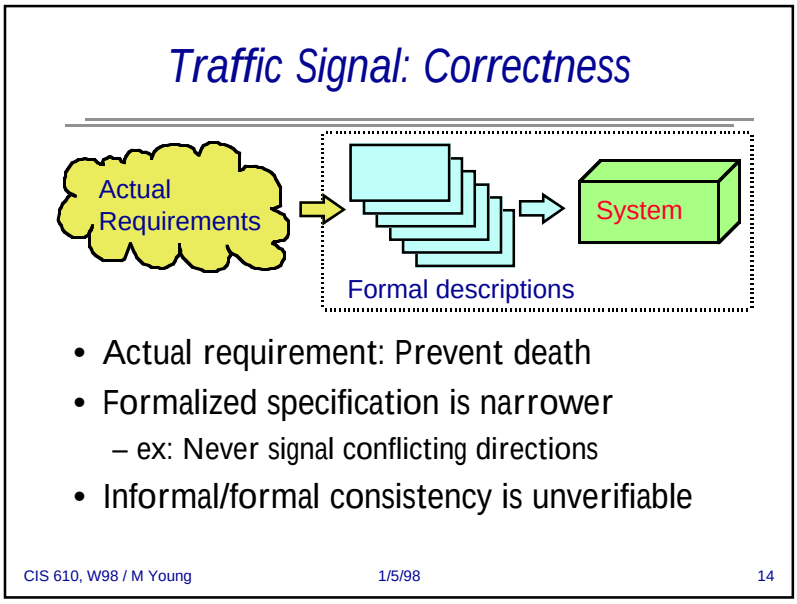
"Fail safe" is avoidance of harmful behavior, perhaps without providing any useful functionality at all. In many cases, this simply means shutting down to avoid doing harm. For example, the "safe state" established by a robot to avoid harming human workers would typically be the "off" state. Sometimes the "safe state" involves some positive action, such as releasing steam pressure from a boiler.

When some positive action is required to maintain safety, it may be hard to distinguish between "fail soft" and "fail safe." For example, when FrameMaker (a word processor) recognizes an inconsistent internal state, it reports an error and terminates, but before terminating it saves a copy of each open file. It probably doesn't matter whether we call this "fail safe" (shutting down to avoid corrupting files) or "fail soft" (providing the very limited functionality of saving one's work before exiting).



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.



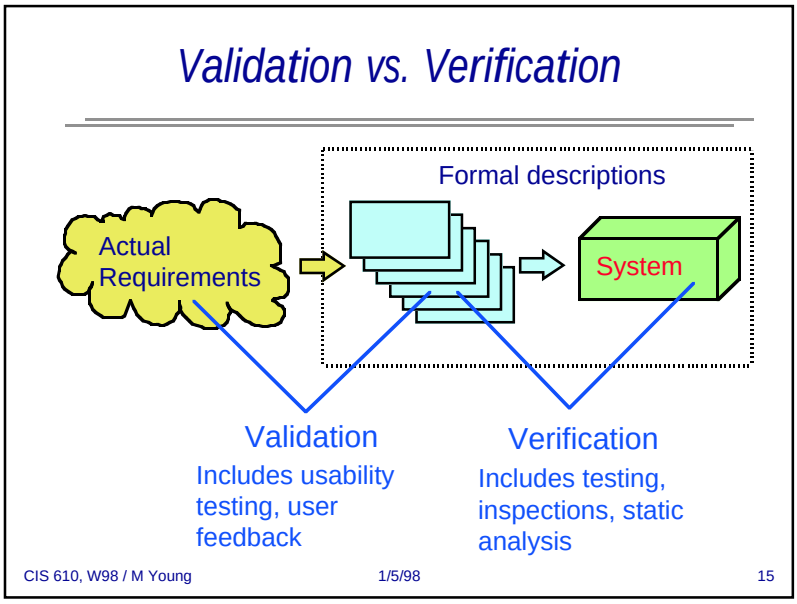
The real safety requirement of the traffic light system is “prevent death,” or more specifically, “never signal in such a way that cars collide with each other or with pedestrians.” It is very difficult to formalize this external requirement in a way that precisely constrains the software system.

This is a very common situation; either the property we really wish to specify is not inherently informal (e.g., “the output should be attractive”), or it is precise (like avoiding collisions) but applies to the system and environment as a whole, and cannot be precisely translated into a sufficient and necessary set of constraints on the software system.

Since correctness is a consistency relation between specification and implementation, it is important that we devise a precise specification of crucial properties even if the specification is an imperfect translation of the actual requirements. Often, moving from an informal statement of a requirement to a specification that is formal enough to be verified involves “narrowing,” that is, reducing the set of acceptable implementations. “Narrowing” is also called “implementation bias,” which is considered an undesirable characteristic of specifications, but it is often the price we must pay for precision.

In addition to the need for narrowing, we are likely to find that we cannot adequately translate a required property into a single specified property. Often we will produce a number of specified properties which (we hope) will together establish the desired property, or at least make it much more likely.

We would like to ensure that the precise specifications of required behavior are consistent with our actual (informal) requirements. Unfortunately, this consistency is unverifiable. At some point in development there is always a transition from informal to formal representations. We can (at least in principle) verify consistency between formal representations, but we cannot verify consistency between an informal representation and a formal representation.



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.

Traffic signal: Robustness

- Specification completeness
  - behavior when a light is disabled
  - behavior when central schedule unavailable
  - self-check and fault recovery
- Negative properties
  - never allow conflicting directions
  - no traffic waits forever

To make the traffic light system robust, we must enhance the specification in ways that are likely to ensure the (informal) requirements, including requirements that may not have been stated explicitly (i.e., things that the user *would* care about if s/he had thought of them).

It is useful to explicitly consider things that can go wrong (undesirable situations for which we should define desirable reactions). We might identify the following problems:

* Light bulbs wear out and fail. Although traffic light bulbs are very high quality and don't burn out often, we should be prepared for the worst.

* The communication line to the central scheduling station could fail. While we cannot maintain full functionality without this central coordination, we may provide some degraded service.

* The software itself, or some other hardware, could fail. We should perform extensive self-checks and try to recover or at least avoid some catastrophic mistake.

Additionally, it is useful to explicitly consider "the worst that could happen," hazards to avoid. These will be stated as negative properties:

* Traffic should never enter from opposing directions (North/South and East/West).

Although less catastrophic, we can also recognize a weakened form of the efficiency requirement which is worth stating as a negative property:

* No traffic waits forever

That is, even if we fail to move traffic efficiently through the intersection, we should at least see to it that no one is stuck at a red light (or a "don't walk" signal) forever.

Word processor: Robustness

- Specification completeness
  - behavior when file system full, memory exhausted, etc. (environment problems)
  - acceptable response to user error
  - crash recovery
- Negative properties
  - never enter “vulnerable” state where crash is unrecoverable

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

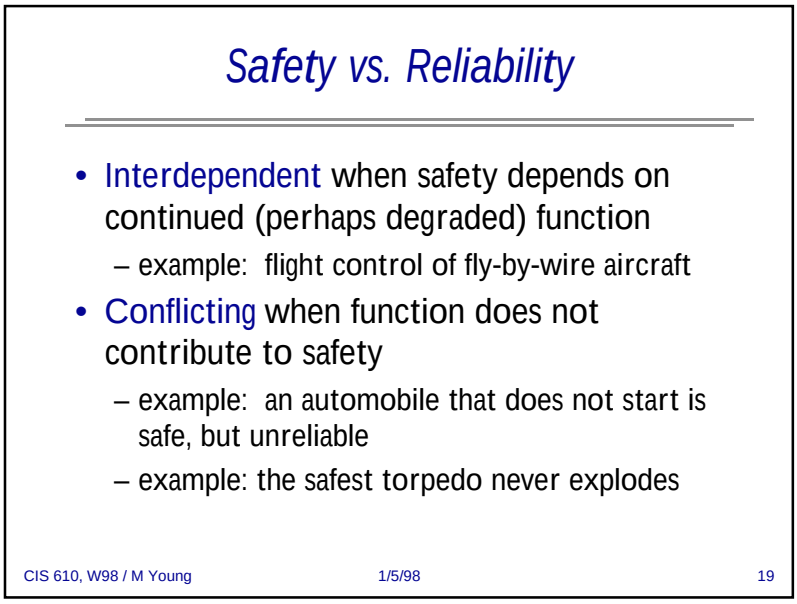
There is an established field of system safety engineering concerned with preventing unsafe behavior; software safety is an extension and adaptation of system safety principles and techniques to software engineering. System safety begins with hazard analysis; hazards are the unsafe situations or behaviors 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!).



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.

Traffic Signal: Fail-soft and Fail-safe

- Elaborate negative requirement: never signal green to conflicting lanes
- If North/South green inoperable, use blinking yellow in its place

  (USA: blinking yellow => "proceed with caution")
- If North/South red inoperable, blink red East/West, yellow North/South

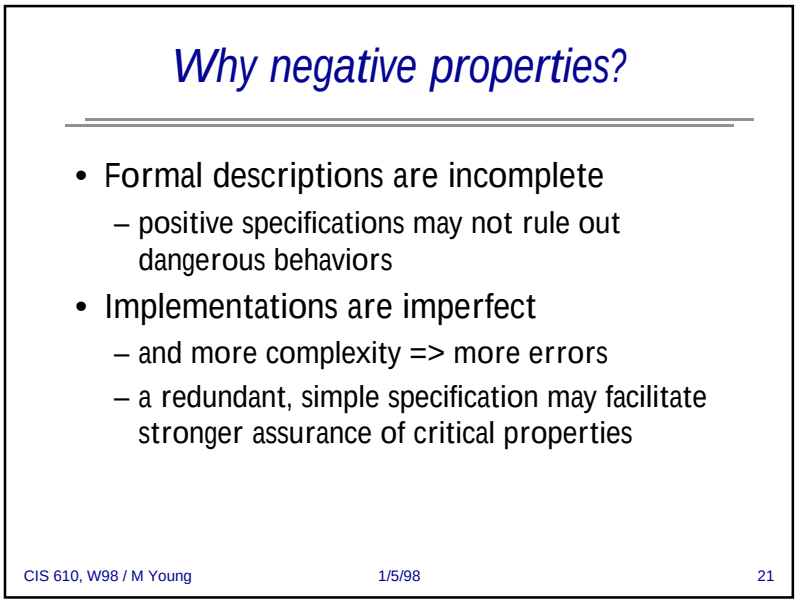
  (USA: blinking red => "stop before proceeding")

We want to ensure the safety requirement of the traffic light system in all circumstances, including the "undesirable situations" we identified.

We will split the "bulb is burnt out" situation into two: A green bulb is burnt out, or a red is burnt out. (We could treat yellow like red).

If a green light is burnt out, we can achieve a "fail soft" state (degraded functionality) by using blinking yellow in its place. All other lights can act as normal.

If a red light is inoperable, we can use a blinking yellow in this direction and a blinking red in the other direction.



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.

Safety property examples from non-life critical applications

- Language interpreter (ex. Java, safe TCL)
  - Prohibit "unsafe" I/O
    - e.g., cannot sneak through a firewall
  - Require predictable type-state (Java)
    - to support the file & network safety assurances
    - includes predictable behavior for *all* errors
- Unix ftp ignores world-readable .netrc
- Operating system: Memory protection prevents system crash

Discussion: Spam Filter

- Hypothetical product: Email filter discards spam (unsolicited bulk email)
- How would you formalize "spam"?
  - Does this suggest something about the relation between validation and verification?
- Identify "hazards" and safety requirements
- How do the safety requirements interact with reliability of the spam filter?