

Unification of Verification and Validation Methods

Overview

Verification and Validation

Properties and Assumptions

Requirements

Property and Assumption Specification

Truth or Falsity of Specifications – Property Evaluation
Methods

Unification of Property Evaluation Methods

Informal Principles

Design for Verification

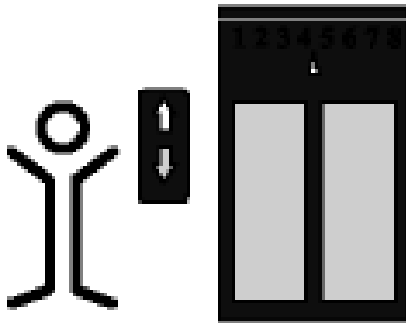
Verification vs Validation

- Verification:
 - "Are we building the product right"
- The software should conform to its specification
- Validation:
 - "Are we building the right product"
- The software should do what the user really requires

Verification or Validation?

Depends on the property

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, but NOT verified (SOON is a subjective quantity)

... 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)

Goal: A software system for which a specified set of *properties* are known to hold given that a set of *assumptions* also hold.

Goal: A software system for which a specified set of *properties* are known to hold given that a set of *assumptions* also hold.

A property is a statement which can be evaluated as either true or false with respect to the software system and the assumptions.

Goal: A software system for which a specified set of *properties* are known to hold given that a set of *assumptions* also hold.

A property is a statement which can be evaluated as either true or false with respect to the software system and the assumptions.

An assumption is a property of the environment in which the system is defined and will execute.

Requirements

A capability for formulating properties and assumptions

The capability for determining the truth or falsity of a set of properties.

Formulation of Properties

Domain knowledge about the software system.

Domain knowledge about the external environment
of the software system.

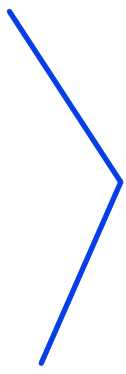
Language in which to specify properties

Example of simplified property: Unmatched Semaphore Operations

original problem

```
if ( .... ) {  
  ...  
  lock(S);  
}  
...  
if ( ... ) {  
  ...  
  unlock(S);  
}
```


Static
checking for
match is
necessarily
inaccurate ...



simplified property

Java prescribes a
more restrictive, but
statically checkable
construct.

```
synchronized(S) {  
  ...  
  ...  
}
```



Property Specification Language

What do we want to be able to establish about programs?

Property Specification Language

What do we want to be able to establish about a program?

The final state resulting from an execution from a given initial state will always conform to program specifications.

It will never be in a given state.

It will always arrive in a given state.

A certain state can only be reached through a specified sequence of other states.

It will reach a certain state within a given time.

What methods are available for establishing the truth or falsity of properties?

Static Analysis – What properties can be evaluated from analysis of that part of program state which is realized without execution.

Test – What properties can be determined by execution from a given initial state and a set of observable states resulting from execution from the given initial state.

Model Checking – What properties can be determined from a complete execution which exhaustively realized the states reachable from a given initial state.

(Initial states for model checking may be non-deterministic.)

Runtime Monitoring – What properties can be established (in principle and in practice) by monitoring of states and behaviors and/or adding redundant computation?

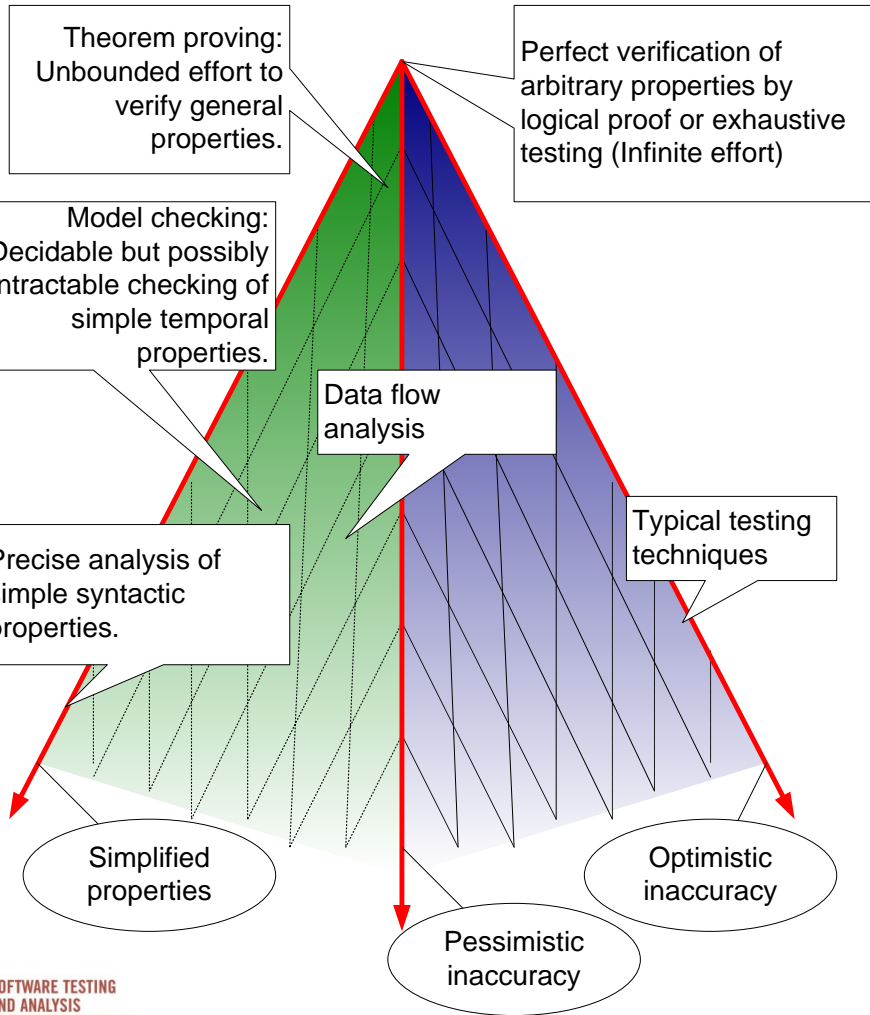
Monitoring – Detection of deviations from specifications

Redundant computation – Detection and sometimes correction of deviations from specifications

Formal Proofs

What properties can be established by deduction or induction to hold under a set of assumptions.

Getting what you need ...



- optimistic inaccuracy: we may accept some programs that do not possess the property (i.e., it may not detect all violations).
 - testing
- pessimistic inaccuracy: it is not guaranteed to accept a program even if the program does possess the property being analyzed
 - automated program analysis techniques
- simplified properties: reduce the degree of freedom for simplifying the property to check

Verification Methods Relationships

- Static Analysis and Model Checking have a common conceptual foundation.
- Model checking is exhaustive testing for a specific property.
- Model checking is equivalent to proof for a specific property.
- Runtime monitor can be based as model checking over traces (or abstractions of traces) as they occur.
- All are some kind of search of a many dimensional space.



Definitions

- **Safe:** A safe analysis has no optimistic inaccuracy, i.e., it accepts only correct programs.
- **Sound:** An analysis of a program P with respect to a formula F is sound if the analysis returns true only when the program does satisfy the formula.
- **Complete:** 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.

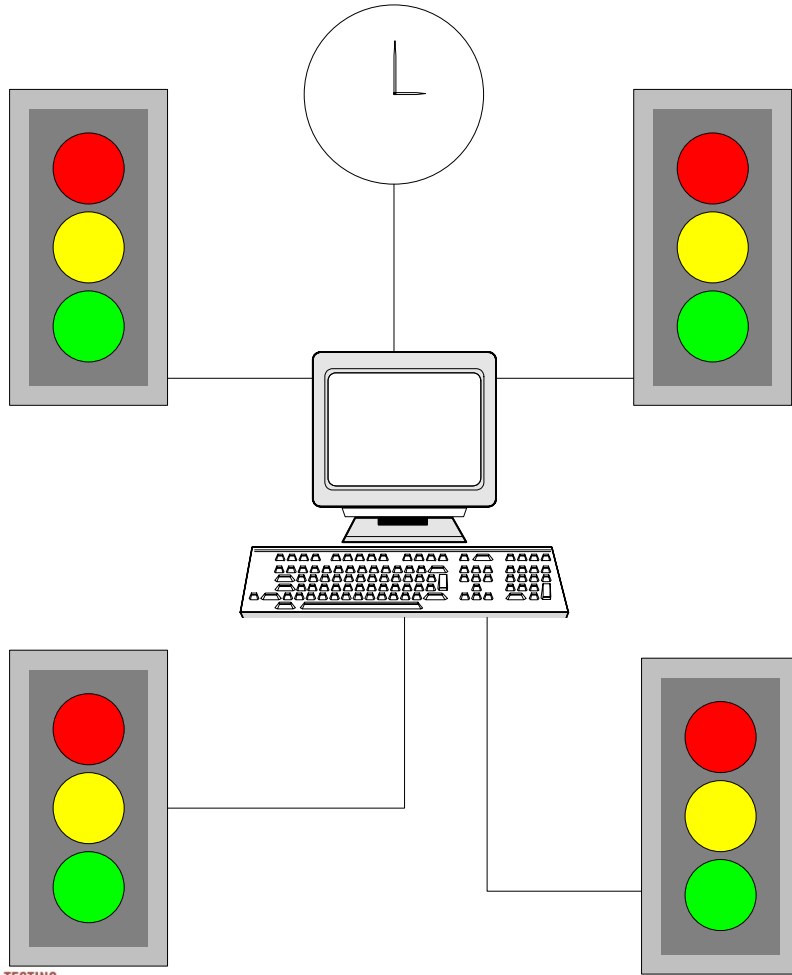


Dependability Qualities

- Correctness:
 - A program is correct if it is consistent with its specification
 - seldom practical for non-trivial systems
- Reliability:
 - likelihood of correct function for some "unit" of behavior
 - relative to a specification and usage profile
 - statistical approximation to correctness (100% reliable = correct)
- Safety:
 - preventing hazards
- Robustness
 - acceptable (degraded) behavior under extreme conditions



Example of Dependability Qualities

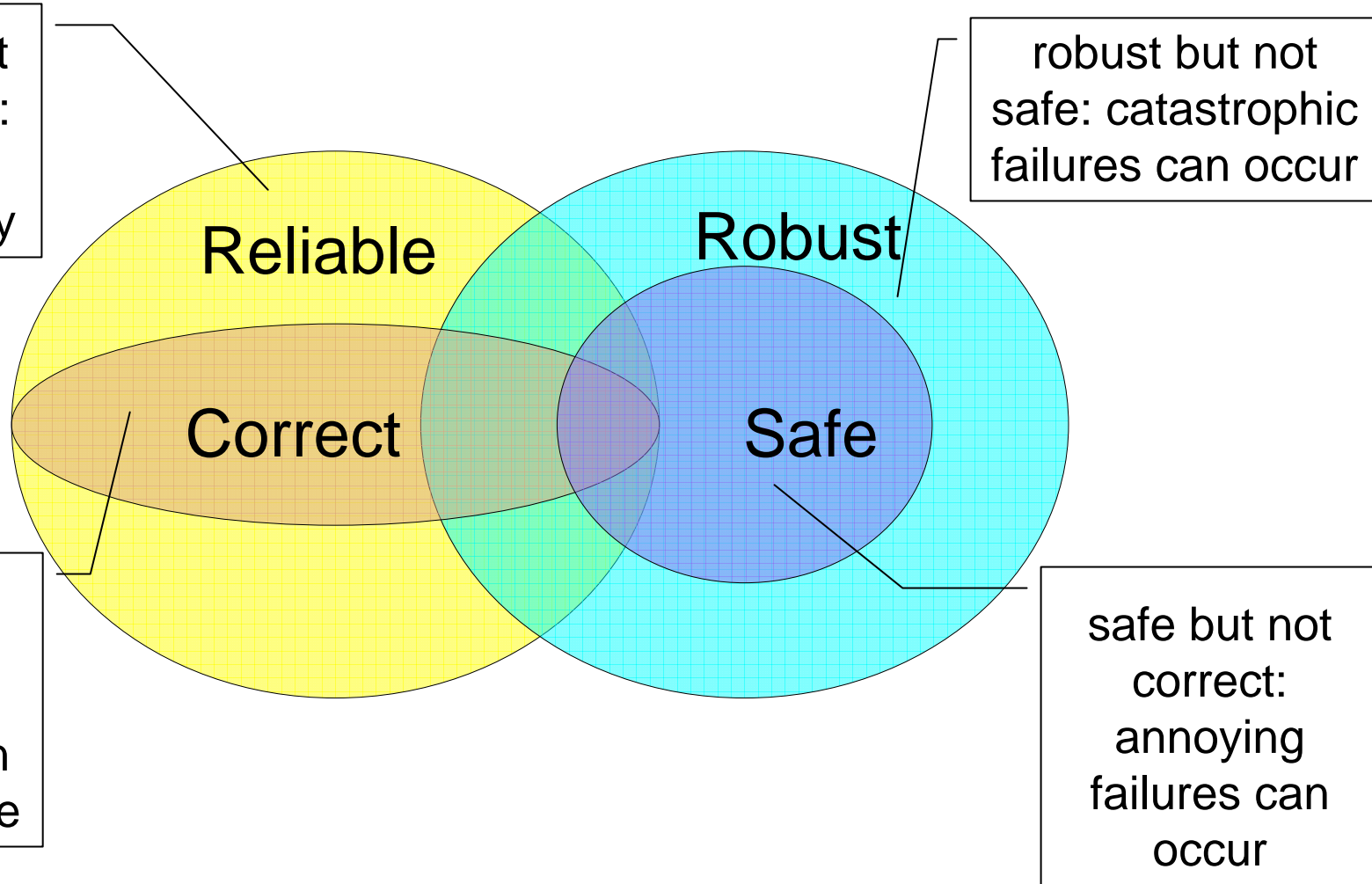


- **Correctness, reliability:** let traffic pass according to correct pattern and central scheduling

- **Robustness, safety:** Provide degraded function when possible; never signal conflicting greens.

- Blinking red / blinking yellow is better than no lights; no lights is better than conflicting greens

Relation among Dependability Qualites



Limitations and Synergism

Static Analysis

Limited in properties to which it applies

Readily automatable

Testing

Can only establish the absence of faults for specific initial states and assumptions.

Can be automated but seldom is

Model Checking

Limited by state space explosion

Can be fully automated

Runtime Monitoring

Can only be applied to a limited set of properties

Adds runtime overhead

Can be automated but seldom is

Formal Proofs

Limited in application to “well-structured” systems and requires great expertise to use.

Just about impossible to automate at the current time.

Synergism

Property Specification

“Most” of the properties verifiable by any method can be expressed in an extended version of temporal logic which incorporates types and values of variables.

Assumption – “Most” correctness, reliability and performance properties can be expressed in an extended temporal logic.

Synergism

Static analysis – Abstraction and model development basic to all other methods are largely based on static analysis.

Testing – Can be made systematic and complete for some subset of properties. Effective testing is a prerequisite for model checking and runtime monitoring.

Model Checking – Complete for the properties and systems to which it applies. Properties established via model checking need not be tested or monitored while assumptions hold.

Runtime Monitoring – Can be made complete for a subset of properties. Overlaps in coverage with model checking.

Formal Proofs – Complete when it can be applied.

Synergism

Many state space reduction algorithms are based on static analysis.

Automation of testing is largely based on static analysis.

Automatic generation of runtime monitoring code is based on static analysis.

State space reduction for model checking is largely based on static analysis

Theorem proving and model checking

Model checking and testing

Taxonomy and Classification of Verification Methods

Attributes

Applied to: representation (static) or execution (dynamic)

Complexity Management: abstraction (folding) or sampling
of state space

Accuracy: pessimistic inaccuracy or optimistic inaccuracy

Pessimistic – find errors which are not real

Optimistic – miss errors which are present

Different emphasis to the same properties

Dependability requirements

- They differ radically between
 - Safety-critical applications
 - flight control systems have strict safety requirements
 - telecommunication systems have strict robustness requirements
 - Mass-market products
 - dependability is less important than time to market
- can vary within the same class of products:
 - reliability and robustness are key issues for multi-user operating systems (e.g., UNIX) less important for single users operating systems (e.g., Windows or MacOS)

Different type of software may require different properties

- **Timing properties**
 - **deadline satisfaction is a key issue for real time systems, but can be irrelevant for other systems**
 - **performance is important for many applications, but not the main issue for hard-real-time systems**
- **Synchronization properties**
 - **absence of deadlock is important for concurrent or distributed systems, not an issue for other systems**
- **External properties**
 - **user friendliness is an issue for GUI, irrelevant for embedded controllers**

Different properties require different V&V techniques

- **Performance can be analyzed using statistical techniques, but deadline satisfaction requires exact computation of execution times**
- **Reliability can be checked with statistical based testing techniques, correctness can be checked with test selection criteria based on structural coverage (to reveal failures) or weakest precondition computation (to prove the absence of faults)**

Different V&V for checking the same properties for different software

- **Test selection criteria based on structural coverage are different for:**
 - **procedural software (statement, branch, path,...) – object oriented software (coverage of combination of polymorphic calls and dynamic bindings,...)**
 - **concurrent software (coverage of concurrent execution sequences,...)**
 - **mobile software (?)**
- **Absence of deadlock can be statically checked on some systems, requires the construction of the reachability space for other systems**

Principles

Principles underlying effective software testing and analysis techniques include:

- **Sensitivity: better to fail every time than sometimes**
- **Redundancy: making intentions explicit**
- **Partitioning: divide and conquer**
- **Restriction: making the problem easier**
- **Feedback: tuning the development process**

Sensitivity:

Better to fail every time than sometimes

- **Consistency helps:**
 - **a test selection criterion works better if every selected test provides the same result, i.e., if the program fails with one of the selected tests, it fails with all of them (reliable criteria)**
 - **run time deadlock analysis works better if it is machine independent, i.e., if the program deadlocks when analyzed on one machine, it deadlocks on every machine**

Redundancy:

Making intentions explicit

- **Redundant checks can increase the capabilities of catching specific faults early or more efficiently.**
 - **Static type checking is redundant with respect to dynamic type checking, but it can reveal many type mismatches earlier and more efficiently.**
 - **Validation of requirements is redundant with respect to validation of final software, but can reveal errors earlier and more efficiently.**
 - **Testing and proof of properties are redundant, but are often used together to increase confidence**

Partitioning:

Divide and conquer

- **Hard testing and verification problems can be handled by suitably partitioning the input space:**
 - **both structural and functional test selection criteria identify suitable partitions of code or specifications (partitions drive the sampling of the input space)**
 - **verification techniques fold the input space according to specific characteristics, thus grouping homogeneous data together and determining partitions**

Restriction.

Making the problem easier

- **Suitable restrictions can reduce hard (unsolvable) problems to simpler (solvable) problems**
 - **A weaker spec may be easier to check: it is impossible (in general) to show that pointers are used correctly, but the simple Java requirement that pointers are initialized before use is simple to enforce.**
 - **A stronger spec may be easier to check: it is impossible (in general) to show that type errors do not occur at run-time in a dynamically typed language, but statically typed languages impose stronger restrictions that are easily checkable.**

Role of Design and Structure in Verification and Validation

The program or system must be amenable to verification.

Assumption: Component-oriented development is required for systems of non-trivial size.

1. Components provide a semantic basis for definition of properties and assumptions.
2. Properties can be established on components under assumptions which model compositions and execution environments.
3. The components can then be replaced in verifications of compositions by an adequate set of established properties.
4. Exhaustive analysis and/or testing is sometimes possible on a component by component basis.

5. Components provide a basis for larger semantic units for monitoring and definition of redundancy.
6. Patterns of components enable definition of properties to be defined and assumptions for verification of properties.