# Keeping Things "As Simple As Possible, but Not Simpler[1]"

David Lorge Parnas

Abstract

Computer Systems are complex and will always be complex. Accepting this, it is important that we follow Einstein's advice and "Keep things as simple as possible, but not simpler[1].

Because software is complex, both the theoretical tools and the software tools that we use for describing, constructing and evaluating software must be subject to Einstein's maxim. If we use complex models and complex notations to analyze complex objects, the complexity will overwhelm us and force us to either ignore relevant facts or give up the effort.

The number of basic concepts and constructs must be minimized, the notation used must be consistent and general rather than a collection of special cases. This talk proposes

- simple semantic basis for understanding software,
- simple (and general) notation for describing software behaviour,
- simple procedures for constructing certifiable software,
- and simple procedures for examining that software.

[1] Usually attributed to Albert Einstein, „Mache die Dinge So einfach wie möglich - aber nicht einfacher."











## My Purpose Here

Get us to face facts (however unpleasant)

Dispel myths

Avoid Obfuscatory Language

Look for the foundations that limit what can be done.

Point out unreasonable limitations on what 3rd Party certifies can do.

Point out where the real needs for progress in this field can be found.





## When can we trust something (computers, people)?

Simple Answer: Never!

- Everything fails sometime!
- Failure has consequences that are difficult to predict

Better Answer: When everything possible has been done and the benefits outweigh the risks and estimated costs.

- We have to trust sometimes.
- We take calculated risks.

Best Answer: When the product meets agreed (testable) standards.

- No *ad hoc* judgements.
- No "Fingerspitzengefühl"!
- Standards based on risks and benefits.

## What can be (honestly) certified about a product? - I

### Syntactically correct?

- Necessary but not anywhere near sufficient.
- Precise and mechanically testable
- We can only certify that the product is syntactically correct (limited value, but…)

### Tested enough?

- There are many criteria on path coverage.
- There are statistical criteria for reliability prediction.
- None are sufficient, but they are the best we have!
- We can only certify that the criteria have been met.

### Valid (specified completely and appropriately for intended use)?

- Requires description of the environment
- Requires complete specification of complete product (system specification)
- Tools are analysis, simulation, inspection
- We can only certify what has been done.





## What can be (honestly) certified about a product? - II

### Suitably designed and developed for certification?

- The same algorithm can be presented in both scrutable and inscrutable ways.
- Software can be clearly structured or monolithic rats nests.
- If software is presented in a baffling way, it can be rejected for that reason alone.
- Clear architectural and documentation standards are needed!
- We can only certify that design and documentation standards have been met

### Probably reliable?

- Using the same mathematics as used for election polls, it is possible to estimate the likelihood that a products reliability will meet specified standards.
- Test case distribution must match actual usage! (Very difficult)
- Predictions based on statistical estimates can be wrong. Everyone must know this.

### Correct? (Validated Requirements and Verified Implementation)

- Only a fool would do it! Only fools would believe it!
- We can certify that the review process was correct but not that all errors were found.





## Can we Separate: Certification from Design and Implementation?

### NO!

- Good design is easier to inspect and test
- Bad design can make inspection impossible and testing impossibly expensive.
- The best design can be badly implemented.
- The implementation that has to be certified.

In other words, saying, “We won’t tell you how to design or how to code, we will simply examine the product”, is doomed to failure.

If the law doesn’t recognize this, the law must be changed.

Certification begins with a well-documented, good design.

Certify a design before allowing coding to begin. (unenforceable)

The certification process does not begin at the end of the development process but certifiers must not be part of the design team.





## Ways to engender confidence in a product - I

### Mechanical Checks

- systematic
- reliable even for tedious tasks
- repeatable
- often rough, miss “semantic” problems.

### Testing

- Rarely complete
- Systems with memory even more problematic - delayed problems may not be found
- Timing problems hard to discover

### “Divide and Conquer” Inspection

- Properly done - allows higher attention level, better ability to focus
- Hard to do properly - requires complete, precise, interface documentation

## Ways to engender confidence in a product - II

### Models: Simplified version of real thing

- Makes the job simpler.
- Problems in real thing may not be visible in model.
- Problems in model may not be real
- Model verification verifies properties of model, but only the model

### Proof: (Program Correctness) Einstein's opinions on the next slide.





## Einstein's Observations on the Meaning of Mathematical Theorems

- *Insofern sich die Sätze der Mathematik auf die Wirklichkeit beziehen, sind sie nicht sicher, und insofern sie sicher sind, beziehen sie sich nicht auf die Wirklichkeit.*
- To the extent that mathematical theorems refer to reality, they are not reliable and to the extent that they are reliable, they do not refer to reality.
- *Mathematische Theorien über die Wirklichkeit sind immer ungesichert - wenn sie gesichert sind, handelt es sich nicht um die Wirklichkeit.*
- *Mathematic*al results about reality are always uncertain; if the results are certain, they are not about reality.
- *Die Mathematik handelt ausschließlich von den Beziehungen der Begriffe zueinander ohne Rücksicht auf deren Bezug zur Erfahrung.*
- Mathematics deals exclusively with the relation between concepts without concern for the relation of those concepts to what we experience.

The last point explains the first two.

- The power of mathematics is that mathematical reasoning is self-contained.
- Theorems are based on "assumptions" and can be mathematically correct even if the assumptions are not accurate. Program proofs are based on assumptions of many kinds.





## Simplicity: The Essential Tool

Complexity is our enemy in building and certifying software.

Some of it is unavoidable - inherent in the problem and the environment.

Einstein had something to say about overcoming complexity.

- *Mache die Dinge so einfach wie möglich - aber nicht einfacher.*
- Keep things as simple as possible, but not simpler.
- Do things in the simplest possible way, but not in a simpler way.

How do we do this?

- The simplest possible approach to semantics
- Decomposition (Divide and Conquer)
- Interface Design and Documentation
- Regularity - often enforced through arbitrary conventions and standards.





## On Semantics for Software Development

### What is Software Semantics?

- A way of describing what a program or component does to its external environment
- Avoid the quagmire of "meaning" (endless discussions)

### Why bother with semantics?

- Precise, concise summaries of component properties essential for "divide and conquer"
- Interface description essential for multi-person projects.

Its not a philosophical question - its a practical tool

Semantics must give developers the information that they need but no more!

### How can we keep semantics simple?

- Minimum number of primitive concepts (concepts not defined in terms of others)
- Uniformity/Consistency - no special cases (built-in)
- Distinguish notation (representation syntax) from concepts

## Semantics for Software Development

### Primitive concepts

- Machine
- State (condition of something at some point in time, fully restricts future behaviour)
- Set (naive set theory)

### Derived concepts

- relations/functions/predicates etc
- composite machine/component machine (=programming variable)

### Notational Concepts (do not confuse concept with its representation)

- expressions built from mathematical variables, relation names,
- predicate expressions (vanilla logic with provision for partial functions)
- tabular expressions

No special treatment of time - just another variable, treat as output





## Reducing complexity in the software itself

### Three Methods

- Separation with abstract interfaces
- Consistent use of patterns (templates)
- Syntactically decomposable programs (structured programs).

Third-party certifiers should make the use of these methods a requirement. Without them certification is much more difficult.

Laws, regulations, conventions may require change.





## Documentation Can Ease Certification

Understanding other people's programme is a super challenge.

Feeling confident that you have understood is an even bigger challenge.

Without supporting documentation, understanding software is an iterative process

- 'Guess' what each part is supposed to do
- Confirm that if each part did what it "should", the whole program is right.
- Confirm that each part did what you thought it should.
- If any confirmations fail, iterate until….

Process terminates by exhaustion (time, budget, people).

Certifier should not have to guess what is intended. Developer should know intent and write it down.





## Three Types of Documentation

Requirements (using 4-variable approach)

Component Interface using Trace Function Method (discrete form of 4-variable approach)

Program Function Tables.

All based on relational model with integrated treatment of time.

Examples Follow:

# Dell Keyboard Checker

N(T) =

| | | | T=_ | ¬ (T= _) ∧ | | |
|---|---|---|---|---|---|---|
| | | | | N(p(T))=1 | 1<N(p(T))< L | N(p(T))= L |
| keyOK | | | | 2 | N(p(T))+ 1 | Pass |
| ¬keyOK ∧ | ¬keyesc ∧ | (¬prevkeyOK ∧ prevkeyesc ∧ preprevkeyOK) ∨ prevkeyOK | | | N(p(T)) - 1 | N(p(T)) - 1 |
| | | ¬prevkeyOK ∧ prevkeyesc ∧ ¬preprevkeyOK | | | N(p(T)) | N(p(T)) |
| | | ¬prevkeyOK ∧ ¬ prevkeyesc | 1 | 1 | N(p(T)) | N(p(T)) |
| | keyesc ∧ | ¬ prevkeyesc | | 1 | N(p(T)) | N(p(T)) |
| | | prevkeyesc ∧ ¬prevexpkeyesc | | Fail | Fail | Fail |
| | | prevkeyesc ∧ prevexpkeyesc | | 1 | N(p(T)) | N(p(T)) |

**21 Pages Replaced by 2**

**This table defines the function N (the next key to press). The predicates are defined in an additional page as functions of T (the event history)**





# Definitions for Keyboard Checker

| Name | Meaning | Definition |
|---|---|---|
| keyOK | most recent key is the expected one | r(T) = N(p(T)) |
| keyesc | most recent key is the escape key | r(T) = esc |
| prevkeyOK | key before the most recent key was expected one | r(p(T)) = N(p(p(T))) |
| prevkeyesc | key before the most recent key was escape key | r(p(T)) = esc |
| preprevkeyOK | key 2 keys before most recent key was expected key | r(p(p(T))) = N(p(p(p(T)))) |
| prevexpkeyesc | key expected before most recent key was escape key | N(p(p(T)))= esc |

**T represents the history of events.**





# Sample Program Function Tables

## Specification for a search program

| | (∃ *i*, B[*i*] =x) | (∀ *i*, ((1 ≤ i ≤ N) ⇒ B[*i*] ≠ x)) |
|---|---|---|
| j’ \| | B[j’] = x | *true* |
| present’= | true | false |

∧ NC(x, B)

## Description of a search program

| | (∃ *i*, B[*i*] =x) | (∀ *i*, ((1 ≤ i ≤ N) ⇒ B[*i*] ≠ x)) |
|---|---|---|
| j’ \| | (B[j’] = x) ∧ (∀ *i*, ((j’ < i ≤ N) ⇒ B[*i*] ≠ x)) | *true* |
| present’= | true | false |

∧ NC(x, B)





# Requirements Table for Radio System

TABLE XIII

$O(T)$ OUTPUT FUNCTION

| T | | | | O(T) |
|---|---|---|---|---|
| $last(T) \in CONFIRM \wedge$ | $last(T) = conf(1) \wedge$ | $rest(T) = \_$ | | $out(last(T))$ |
| | | $rest(T) \neq \_$ | | $Error$ |
| | $last(T) \neq conf(1) \wedge$ | $last(rest(T)) = conf(num(last(T)) - 1)$ | | $out(last(T))$ |
| | | $last(rest(T)) \neq conf(num(last(T)) - 1)$ | | $Error$ |
| $last(T) \in REJECT \wedge$ | $rnhBarred \wedge$ | $last(T) = rej(2) \wedge last(rest(T)) = rej(4))$ | | $out(last(T)$ |
| | | $\neg last(T) = rej(2) \wedge last(rest(T)) = rej(4))$ | | $Error$ |
| | $\neg rnhBarred \wedge$ | $last(T) = rej(8) \wedge$ | $last(rest(T)) \in prev\_conf(rej(8))$ | $out(last(T))$ |
| | | | $\neg last(rest(T)) \in prev\_conf(rej(8))$ | $Error$ |
| | | $\neg last(T) = rej(8) \wedge$ | $last(rest(T)) = rej(num(last(T)) + 1) \vee last(rest(T)) \in prev\_conf(rej(last(T)))$ | $out(last(T))$ |
| | | | $\neg last(rest(T)) = rej(num(last(T)) + 1) \vee last(rest(T)) \in prev\_conf(rej(last(T)))$ | $out(last(T))$ |
| $\neg(last(T) \in CONFIRM \vee last(T) \in REJECT)$ | | | | $Error$ |

## Summary: If I were asked to certify a software product,…

Set standards for design and documentation

Reject submissions if they do not meet these standards

Start Early: Certify Precisely documented requirements - Do not write them.

Keep Involved: Review Module Structure Documentation - Do not write them.

Upon delivery: Check documentation first (reject if unsatisfactory)

Document driven inspection - Using their documentation.

Document-driven reliability-based testing - Using Certified Requirements.

From the start: Insist that documents and code carry “seals” from qualified (Certified) designers and reviewers.

Do not accept “spaghetti code” “tossed over the wall” - even if correct!