

Software Inspections We Can Trust

David Lorge Parnas, P.Eng.

NSERC/Bell Industrial Research Chair in Software Engineering
 Director of the Software Engineering Programme
 Department Of Computing And Software (CAS)
 Faculty of Engineering, McMaster University
 Hamilton ON Canada L8S 4K1

Software is devilishly hard to inspect. Serious errors can hide for years. Consequently, many are hesitant to employ software in safety-critical applications and all companies are finding correcting and improving software to be an increasingly burdensome cost.

This talk describes a procedure for inspecting software that consistently finds subtle errors in software that is believed to be correct. The procedure is based on four key principles:

- All software reviewers actively use the code.
- Reviewers exploit the hierarchical structure of the code rather than proceeding sequentially through the code.
- Reviewers focus on small sections of code, producing precise summaries that are used when inspecting other sections. The summaries provide the “links” between the sections.
- Reviewers proceed systematically so that no case, and no section of the program, gets overlooked.

During the procedure, the inspectors produce and review mathematical documentation. The mathematics allows them to check for complete coverage; the notation allows the work to proceed in small systematic steps.

Department of Computing And Software
 Software Engineering Research Group
 “connecting theory with practice”

Why is Software so often a Problem?

Developers *consistently* underestimate the difficulty of building software for long-term use.

They *write* software rather than *design* it.

They do not:

- systematically, identify and record requirements,
- hold reviews of the requirements document,
- explicitly design, document and review software structure,
- carefully inspect all designs and programs.

These steps are standard practice for all engineering products other than software.

The steps are not taken for software because,

- “Software is easy!”
- “The code is self-documenting!”
- “Software is *just* a set of instructions.”
- “Anyone who knows the language can program.”

Famous last words!

Department of Computing And Software
 Software Engineering Research Group
 “connecting theory with practice”

Responsibilities of (Software) Engineers

- To understand the properties of their products thoroughly.
- To follow established rules of good practice when designing and building products.
- To apply theory where it has been demonstrated to lead to better, or safer, products.

Engineering is Not Management

The art of system management is the ability to get things built without knowing exactly what they are.

The engineer is expected to thoroughly understand the properties of the product.

Software projects are hard to manage - especially if they are badly designed, but...

Unless we have good Engineers, the best managers will not be able to successfully manage these projects.

Department of Computing And Software
 Software Engineering Research Group
 “connecting theory with practice”

Why Don't People Apply Engineering Discipline to Software?

- (1) Some don't have an engineering education.
- (2) Some don't think it's necessary.
- (3) Some don't know how to do it.

Why don't we demand that software people have appropriate qualifications?

- Experience shows that it is necessary.
- We license hairdressers don't we.

Why aren't software designers required to be Engineers?

- They should understand more than the code.
- They must be sure their product is fit for use.

Why do we continue to think of them as scientists and to educate them accordingly?

- Inertia?

Department of Computing And Software
 Software Engineering Research Group
 “connecting theory with practice”

Why Don't Designers Apply Mathematics, and "Theory" to Software Products?

The last 30 years have seen great advances in our understanding of software science.

Programs written by most engineers have not taken advantage of this theory.

Programs written by most other programmers do not reflect this theory.

- Many don't know the theory.
- Those who know it don't know how to apply it
- Much of it is difficult to apply, perhaps even not applicable.
 - Deals with impractical languages
 - Deals with unbounded memory size
 - Uses unnecessarily difficult notation
 - Designed for the wrong purpose

There is a need to connect theory to practice.

Let's start with software inspections.

When is Software Critical?

Critical is not necessarily "safety critical"

Other types of critical programs:

- Mass distributed programs in warranty situations
- Critical kernels in many systems
- Financial Systems
- Security (Privacy, Data Protection) programs

The common property of all of these examples is that the cost of a failure is high.

If you value your reputation, your work may be critical.

The Critical-Software Tripod

- (1) Precise, well organised, mathematical documentation with systematic review
- (2) Extensive Testing
 - Systematic Testing-quick discovery of gross errors
 - Random Testing -discovery of shared oversights and reliability assessment
- (3) Qualified People and Approved Processes

The Three Legs are complementary

The three legs are all needed.

The stool falls over if any leg is forgotten.

The third leg is the shortest.

It's the shortest leg that we should worry about.

Today we discuss only leg (1).

Why Conventional Reviews are Ineffective

- (1) The reviewers are swamped with information.
- (2) Most reviewers are not familiar with the product design goals.
- (3) There are no clear individual responsibilities.
- (4) Reviewers can avoid potential embarrassment by saying nothing.
- (5) The review is conducted as a large meeting where detailed discussions are difficult.
- (6) Presence of managers silences criticism.
- (7) Presence of uninformed reviewers may turn the review into a tutorial.
- (8) Specialists are asked general questions.
- (9) Generalists are expected to know specifics.
- (10) The review procedure reviews code without respect to structure. (n lines per hour)
- (11) Unstated assumptions are not questioned.

Effective Reviews are Active Reviews

A dilemma:

- Errors in programs and design documents should be found *before* the documents/systems are used.
- Errors in programs and documents are usually found *when* the documents are used.

Another dilemma:

- Everyone's work requires review!
- It's easier to say "OK" than to find subtle errors!
- Reviewer's approval is not reviewed.

One more dilemma:

- No individual can review all aspects of a design.
- When working in a group, people tend to relax in the knowledge that others are also working the problem.

Solutions:

- Make the reviewers use the documents.
- Make the reviewers document their analysis.
- Have specialised reviews. Ask the reviewer about things that they know.
- Make the reviewers provide specifics - not just a bit.

Previous Work on Inspections

Best known approach Fagan - 1976.

Many followers - new book by Gilb.

Explicitly focus on the **management** aspects.

- Who should be there?
- What are the roles of the participants?
- How long is a meeting?
- How fast do you work?
- Forms for reporting errors?

Read the code in sequence and paraphrase.

Paraphrases are informal.

Most observers find these more effective than conventional reviews or walkthroughs, but...

... can we do better?

Parnas/NRL/AECB/AECL/Ontario Hydro

Focus on the **engineering** side.

Depend on hierarchical decomposition rather than sequential reading.

Use mathematical notations to provide precise descriptions rather than informal paraphrases.

Produce useful *precise* documentation as a side effect.

Proceed much more quickly if the documentation was produced by the developers.

Insures that cases and variables are not overlooked.

Applies simple mathematics to check for completeness aspects.

Active Review of Design Documents

Base the review process on the nature of the document.

(1) Begin by identifying desired properties.

(2) Prepare questionnaires for the reviewers. Ask them questions that:

- make them use the document.
- make them demonstrate that the desired properties are present.
- ask for sources of information to support the answers to other questions.

For example:

- Ask reviewers to identify the domain of the program
- Ask reviewers to identify "error" cases.
- Ask reviewers to explain why no other error cases are possible.
- Ask reviewers to explain why the behaviour required for each case is the desired behaviour.

For more information read [1].

Inspecting Programs

It is the code that “hits the road”.

Getting the requirements right, the structure right, the interfaces right, the documentation right, etc. are all important but *we have to check the code*.

The same review principles apply, viz:

- Make the reviewers use the material they review.
- Make the reviewers answer questions.
- Ask the reviewer about things that they know.
- Make the reviewers provide specifics.

We compare completed programs with previously reviewed specifications.

We ask some reviewers to produce precise descriptions.

We ask other reviewers to show that the descriptions match the specifications.

It is hard work but it produces results.

- We get good documentation for future use.
- We find errors in the best industrial code - programs that were considered correct.

Our Code Inspection Process

- (1) Prepare a precise specification of what the code should do - a program function table.
- (2) Decompose the program into small parts appropriate for the “display approach” [2].
- (3) Produce the specifications required for the “display approach”.
- (4) Compare the “top level” display description with the requirement specification.

Observations:

- You can’t inspect without precise requirements.
- Step (2) would already have been done if you use the display method for documentation.
- Step (3) is truly an active design review
- All reviewer work is itself reviewable.
- If you did not already have it, the by-product is thorough documentation.
- It’s a bunch of small steps and very systematic.

Descriptions vs. Specifications

An actual description is a statement of some actual attributes of a product, or set of products. □

A specification is a statement of all properties required of a product, or a set of products. □

In the sequel, “description”, without modifier, means “actual description”.

The following are implications of these definitions:

- A description may include attributes that are not required.
- A specification may include attributes that a (faulty) product does not possess.
- The statement that a product satisfies a given specification may constitute a description.

The third fact results in much confusion. A useful distinction has been lost.

Descriptions vs. Specifications

Any list of attributes may be interpreted as either a description or a specification.

Example:

“A volume of more than 1 cubic meter”

This could be either an observation about a specific box or, a statement of the requirements for a box that is about to be purchased.

A specification may offer a choice of attributes; a description describes the actual attributes, but need not describe the product completely.

Sometimes one may use one’s knowledge of the world to guess whether a statement is a description or a specification.

Example:

“Milk, badly spoiled”

Guessing is not reliable. We need to explicitly label specifications and descriptions so that the intended use is clear.

Do We Need *New* Semantics Theories For Programming?

Not for the practical software engineering problems that I see.

I can find 30 year old theory that works for the problems that I will describe today.

Semantic theory has failed to describe real languages, but (in my opinion) the fault lies with the languages.

We do need improvements in:

- the notation used to describe actual programs
- the ability to describe behaviour in terms of the values of observable variables - nothing else.
- convenient ways to deal with all aspects of termination including non-deterministic non-termination.

What follows is mathematically equivalent to some **very old** ideas, but has some practical advantages.

A Mathematical Interlude - LD-relations.

A *binary relation* R on a given set U is a set of ordered pairs with both elements from U , i.e. $R \subseteq U \times U$.

The set U is called the *Universe of R* .

The set of pairs R can be described by its *characteristic predicate*, $R(p,q)$, i.e. $R = \{(p,q): U \times U \mid R(p,q)\}$.

The *domain* of R is denoted $\text{Dom}(R)$ and is $\{p \mid \exists q [R(p,q)]\}$.

The *range* of R is denoted $\text{Range}(R)$ and is $\{q \mid \exists p [R(p,q)]\}$.

Below, "relation" means "binary relation".

A *limited-domain relation* (LD-relation) on a set, U , is a pair, $L = (R_L, C_L)$ where:

R_L , the *relational component* of L , is a relation on U , i.e. $R_L \subseteq U \times U$, and

C_L , the *competence set* of L , is a subset of the domain of R_L , i.e. $C_L \subseteq \text{Dom}(R_L)$.

Using LD-Relations as Before/After Behavioural Descriptions (1)

Let P be a program, let S be a set of states, and let $L_P = (R_P, C_P)$ be an LD-relation on S such that $(x,y) \in R_P$ if and only if $\langle x, \dots, y \rangle$ is a possible terminating execution of P , and $x \in C_P$ if and only if P is guaranteed to terminate if it is started in state s .¹

L_P is called the *LD-relation of P*

By convention, if C_P is not given, it is, (by default), $\text{Dom}(R_P)$.

With this convention, our approach is upwards compatible with the "cleanroom" approach for dealing with deterministic programs.

¹ Please note that C_P is not the same as the precondition used in VDM [4]. S_P is the set of states in which the termination of P is certain.

Using LD-Relations as Before/After Behavioural Descriptions (2)

The following follow from the definitions:

- If P starts in x and $x \in C_P$, P always terminates; if $(x, y) \in R_P$, P may terminate in y .
- If P starts in x , and $x \in (\text{Dom}(R_P) - C_P)$, the termination of P is non-deterministic; in this case, if $(x, y) \in R_P$ when P is started in x , it may terminate in y or may not terminate.
- If P starts in x , and $x \notin \text{Dom}(R_P)$, then P will never terminate.

By these conventions we are able to provide complete before/after descriptions of any program but retain a simpler representation to use for those cases that arise most often.

Specifying Programs (1)

Specifications may *allow* behaviour not actually exhibited by a satisfactory program.

We can also use LD-relations as before/after specifications. To understand the meaning of a specification, you must understand what “satisfies” means.

Let $L_p = (R_p, C_p)$ be the description of program P.
 Let S, called a *specification*, be a set of LD-relations on the same universe and $L_S = (R_S, C_S)$ be an element of S.
 We say that

- (1) P satisfies an LD-relation L_S , if and only if $C_S \subseteq C_P$ and $R_P \subseteq R_S$, and
- (2) P satisfies a specification, S, if and only if L_P satisfies at least one element of S.

Often, S has only one element. If $S = \{L_S\}$ is a specification, then we can also call L_S a specification.

Specifying Programs (2)

The following follow from the definitions:

- A program will satisfy it’s own description as well as infinitely many other LD-relations.
- An acceptable program must NOT terminate when started in states outside $Dom(R_S)$.
- An acceptable program must terminate when started in states in C_S ($C_S \subseteq Dom(R_p)$).
- An acceptable program may only terminate in states that are in $Range(R_S)$.
- A deterministic program can satisfy a specification that would also be satisfied by a non-deterministic program.

Note the following differences between the description and the specification of a program.

- There is only one LD-relation describing a program, but that program will satisfy many distinct specifications described by different LD-relations.
- An acceptable program need not exhibit all of the behaviours allowed by R_S ($R_p \subseteq R_S$).
- An acceptable program may be certain to terminate in states outside C_S . ($C_S \subseteq C_p$).

The intended use of each LD-relation (specification or description) must be stated explicitly!

Tabular Descriptions and Specifications

Specification for a search program

$$(\exists i, B[i] = x) \quad (\forall i, ((1 \leq i \leq N) \Rightarrow B[i] \neq x))$$

j'	B[j'] = x	true	\wedge NC(x, B)
present' =	true	false	

Description of a search program

$$(\exists i, B[i] = x) \quad (\forall i, ((1 \leq i \leq N) \Rightarrow B[i] \neq x))$$

j'	$(B[j'] = x) \wedge (\forall i, ((j' < i \leq N) \Rightarrow B[i] \neq x))$	true	\wedge NC(x, B)
present' =	true	false	

The above is one of many kinds of tables!
 Simple tables like this understate the advantage.
 These have proven “practitioner appeal”

A Simple Example

```
(integer array H[1:N];
(integer c; integer n; n <= 1;
it (n <= N ->
(
(integer u; integer l; boolean p; l <= 1; c <= 0;
it (u <= l + n - 1;
(u <= N -> (
(integer i; i <= 0; p <= true;
it (i <= (u - l + 1) + 2 ->
(A[l+i] = A[u-i] -> (i <= i + 1; *)
| A[l+i] <= A[u-i] -> (p <= false; ●))
| (u - l + 1) + 2 -> i -> ●)
it )
;
(-p -> skip | p -> c <= c + 1; l <= l + 1; *)
| u > N -> ●))
it )
;
H[n] <= c; n <= n + 1; *)
| n > N -> ●)
it )
)
```

Decomposition

(integer array H[1:N];

(integer c; integer n; n <= 1;

it (n <= N →

(

(integer u; integer l; boolean p; l <= 1; c <= 0;

it (u <= l + n - 1;

(u <= N → (

(

(integer i; i <= 0; p <= true;

it (i <= ⌊(u - l + 1) ÷ 2⌋ →

(A[l+i] = A[u-i] → (i <= i + 1; ⚡))

| A[l+i] ≠ A[u-i] → (p <= false; ●))

| ⌊(u - l + 1) ÷ 2⌋ ≤ i → ●)

ti)

;

(¬p → skip | p → c <= c + 1; l <= l + 1; ⚡)

| u > N → ●))

ti)

;

H[n] <= c; n <= n + 1; ⚡)

| n > N → ●)

ti)

)

Department of Computing And Software
Software Engineering Research Group
"connecting theory with practice"

Display: An Example

Problem: ctpal ≡

true

H₁

c' =

card^a({l | pal(A,l,n + l - 1)})

G

H₂

∧ NC(n,A)

a. card(x), where x is a set, is the number of elements in x.

Solution: ctpal ≡

(integer u, l; boolean p; l <= 1; c <= 0;

it (u <= l + n - 1;

(u <= N → (palu; (¬p → skip | p → c <= c + 1);

l <= l + 1; ⚡))

| u > N → ●))

ti)

palu ≡ _lNC(l,u,A) ∧ (p' = pal(A,l,u))

where

pal(A,b,c) ≡ ((1 ≤ b ≤ c ≤ N) ∧

(∀ i, 0 ≤ i < ⌊(c - b + 1) ÷ 2⌋ ⇒ A[b+i] = A[c-i]))

Department of Computing And Software
Software Engineering Research Group
"connecting theory with practice"

Displays: An Explanation

The top part of each display is the specification for the program in the middle.

The program in the middle is kept small by removing sections, creating a display for them, and including their specification in the bottom part.

The bottom part contains a specification of these invoked programs.

To check a display determine the description of the program in the middle, and see if it satisfies the specification at the top. In doing this, use the specifications of the invoked programs, not their text.

To check a set of displays, make sure that every specification at the bottom of one display is at the top of another. The exceptions:

- standard programs
- primitive programs

Completeness can be checked mechanically.

Department of Computing And Software
Software Engineering Research Group
"connecting theory with practice"

Can we Document Real Programs This Way?

Yes,

- Ontario Hydro/AECL/AECB did it.
- Key components of our tool system were documented in this way?
- We have done some parts of commercial systems.
- Small components are done in my industrial courses.

But,

How important is it to you?

It will cost "up front time", may save time and cost later.

Department of Computing And Software
Software Engineering Research Group
"connecting theory with practice"

Structure and Inspection

Well-structured programs are easier to decompose. They can be decomposed by purely syntactic means.

Well-structured programs are much easier to inspect.

Inspection encourages good structuring.

Inspection suggests structural improvements.

Inspected programs are easier to maintain.

Modified programs need not be completely re-inspected. The parts that must be inspected again can be easily identified.

The cost of future maintenance is greatly reduced.

The definition of “well-structured” should not be based on the absence or presence of certain control structures. It has to do with the ease of decomposition. [2]

Our Initial Experience: Darlington Nuclear Power Generating Station¹

Three control systems in Canadian reactors:

- one normal control system
- two independent shutdown systems

Safety analysis *assumes* control system will fail. Only shutdown systems are considered safety-critical.

Previous shutdown systems were analogue and relay systems.

At Darlington they are software controlled.

Each Software System has a simple task.

Their designs are “diverse”.

The systems are more complex than their predecessors with the result that AECB² could not be confident of their trustworthiness.

How can we increase that level of confidence?

¹ Discussed in more detail in [4] and [3].

² Atomic Energy Control Board of Canada

Why We Could Not Use English

The following type of sentence was found in the requirements document.

“Shut off the pumps if the water level is above 100 meters for 4 seconds”

What does this simple sentence mean?

Three Reasonable Interpretations:

“Shut off the pumps if the mean water level over the past 4 seconds was above 100 meters”.

$$\left[\left(\int_{T-4}^T WL(t) dt \right) \div 4 > 100 \right]$$

“Shut off the pumps if the median water level over the past 4 seconds was above 100 meters”.

$$\left(\text{MAX}_{[t-4,t]} (WL(t)) + \text{MIN}_{[t-4,t]} (WL(t)) \right) \div 2 > 100$$

“Shut off the pumps if the “rms” water level over the past 4 seconds was above 100 meters”.

$$\sqrt{\left(\int_{T-4}^T WL^2(t) dt \right) \div 4} > 100$$

A Fourth (Unreasonable) Interpretation:

“Shut off pumps if the minimum water level over the past 4 seconds was above 100 meters”.

$$\text{MIN}_{[T-4, T]} [\text{WL}(t)] > 100$$

This is the most literal interpretation!

It is a disaster waiting to happen!

If you use natural languages, there are thousands of such phrases waiting to “bug” you.

The Inspection Process at Darlington

Four teams:

- (1) Application Experts
- (2) Programming Experts
- (3) Verifiers
- (4) Auditors

Roles of the teams:

- (1) Produces requirements tables.
- (2) Produce Program Function Tables (Displays).
- (3) Show (1) = (2) and that (2) are correct.
- (4) Audit the “proofs”.

Subsequent Experience

In classes on this method, we have applied this to numerous small industrial programs that were believed to be correct.

In most cases, we found unexpected errors.

In some cases, the participants could not state the requirements.

In other cases, the program could not be decomposed (machine code w/o documentation).

I believe that one program was correct.

In all cases, we could improve the program.

We have found errors in textbook programs, library programs, and well-used and tested programs.

No process is perfect, but this one engenders confidence. It produces code that people trust.

What Makes Things Hard?

Variables with no names.

Variables with long names or characterising expressions.

Quantification over indices rather than elements.

Programs that are not understood.

Programs that are badly modularised.

Self-referencing data structures

These can all be fixed!

Essential Point: Divide and Conquer

The initial decomposition is essential. Attempts to simply scrutinise the program fail.

Trying to read the program the way a computer would is much less effective. Logically connected parts may be far apart.

The use of tables is essential. It breaks things down into simple cases so that

- We can be sure that all cases are covered
- Each case is straightforward

We consider all variables, but one at a time.

We consider all cases, one at a time.

We can take “breaks”, go home and sleep, even take holidays, without losing our place.

Using displays and tabular summaries is far more work than Fagan’s English paraphrasing, but it imposes a discipline that helps.

The Other Essential Point: Precise, Abstract Descriptions

Having lots of little parts is not enough.

We have to be sure that the parts fit together.

We have to be able to do that without page-flipping.

Each part’s behaviour must be precisely summarised without giving intermediate states.

We must be sure that the description at the bottom of one display will be identical with that at the top of another display.

These global checks can, and have been, mechanised.

Precise descriptions are painstaking work, but if quality is important, they are essential.

It’s not always easy!

The most critical step, besides decomposition, is finding a good representation for the state space.

It is not always worthwhile.

There are informal variations.

It is a capability that your organisation should have.

Some Suggested Reading

- (1) Parnas, D. L., Weiss, D. M., “Active Design Reviews: Principles and Practices”, *Proceedings of the 8th International Conference on Software Engineering*, London, August 1985.
Also in *Journal of Systems and Software*, December 1987.
- (2) Parnas, D. L., Madey, J., Iglewski, M., “Precise Documentation of Well-Structured Programs”, *IEEE Transactions on Software Engineering*, Vol. 20, No. 12, December 1994, pp. 948 - 976.
- (3) Parnas, D. L. “Inspection of Safety Critical Software using Function Tables”, *Proceedings of IFIP World Congress 1994, Volume III*, August 1994, pp. 270 - 277.
- (4) Parnas, D. L., Asmis, G.J.K., Madey, J., “Assessment of Safety-Critical Software in Nuclear Power Plants”, *Nuclear Safety*, vol. 32, no. 2, April-June 1991, pp. 189-198.