Software Inspections We Can Trust

David Lorge Parnas, P.Eng,
Ph.D, Dr.h.c., Dr.h.c., FRSC, FACM, FCAE
SFI Fellow, Professor of Software Engineering
Director of the Software Quality Research Laboratory (SQRL)
Department of Computer Science and Information Systems
Faculty of Informatics and Electronics
University of Limerick

Software is devilishly hard to inspect. Serious errors can escape attention for years. Consequently, many are hesitant to employ software in safety-critical applications and developers and users are finding the correction of software errors 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 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 t**hat 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.

The inspectors produce and review mathematical documents. The mathematics allows them to check for complete coverage; tabular notation allows the work to proceed systematically in small steps.

Responsibilities of (Software) Engineers

Engineers are expected:

- to understand the properties of their products thoroughly.
- to follow established rules of good practice when designing and building products.
- to apply theory that has been demonstrated to lead to better, or safer, products.
- to use their understanding of fundamental science and mathematics to keep up with new ideas.

If you do not apply a technique that has been shown to work, and something goes wrong, the responsibility is yours.

All of this is recognized in traditional Engineering - why do we not expect the same of software developers?

Software Engineering is not just Project Management

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

The engineer must *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 ma**nagers will not be able to successfully manage these projects.

Inspections have to be carefully managed but the detailed work must be executed using the most rigorous techniques available.

Today we can automate much of the management of inspections.

We are not yet able to automate the actual checking, but tools can help.

When is Software a Critical product?

“Critical” does not necessarily mean “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
- Any system where a failure may lead to a lawsuit
- Administrative systems for public organizations

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

These three legs are complementary

- The three legs are *all* needed.
- The tripod will not stand if any leg is forgotten.
- The third leg is the shortest - it limits our ability to extend the others.
- Today we discuss only leg (1).

Twelve Reasons 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 a large meeting; 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.
(11) Unstated assumptions are not questioned.
(12) Inadequate time is allowed.

Active Reviews are Effective Reviews

A dilemma:

- Errors 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 look for and report subtle errors!
- Reviewer's approval is not reviewed.

One more dilemma:

- No individual can review all aspects of a design.
- In a group, people tend to relax knowing that others are also working on 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 single bit (ok/not ok).

Previous Work on Inspections

Best known publication Fagan - 1976.

Many followers - new book by Gilb.

Explicitly focus on the **management** issues.

- 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 it in a natural language.

Paraphrases are informal.

Most observers find these more effective than conventional reviews or walk-throughs, but...

... can we do better?

Parnas/NRL/AECB/AECL/Ontario Hydro

Focus is more technical.

Depends on hierarchical decomposition rather than sequential reading.

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

Produces useful *precise* documentation as a side effect.

- Proceed much more quickly if the documentation has already been produced by the developers.
- Documentation also useful in the testing process.

Insures that cases and variables are not overlooked.

Applies mathematics to check for completeness and consistency of documents.

Can take advantage of mathematics based tools such as theorem provers and computer algebra systems.

Active Review of Design Documents

Base the review process on the nature of the document.

(1) Begin by identifying and listing 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].

Beyond Inspecting Documents - 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 programs that were considered correct and thoroughly tested.

Our Code Inspection Process

(3) Prepare a precise specification of what code should do.

(4) Decompose the program hierarchically into parts.

(5) Produce the descriptions required for the “display approach” [2].

(6) Compare the “top level” display description with the requirement specification.

Observations:

- You can’t inspect without precise requirements.
- Step (2) would 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

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

Definition: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 is 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.

Decades 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.

Standard definition:
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 Dom(R) and is $\{p \mid \exists q\, [R(p,q)]\}$.

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

Below, “relation” means “binary relation”.

New Definition:
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. $C_L \subseteq Dom(R_L)$.

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

Definition:
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 <x,…,y> 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 describing* P

By convention, if C_P is not given, it is $Dom(R_P)$.

With this convention, our approach is upwards compatible with the “cleanroom” approach to describe deterministic programs.

[1] Please note that C_P is not the same as the precondition used in VDM [4]. C_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 in some state; if $(x, y) \in R_P$, y might be that state.
- If P starts in x, and $x \in (Dom(R_P) - C_P)$, the termination of P is non-deterministic; in this case when started in x, P might not terminate or might terminate in any state y such that $(x, y) \in R_P$.
- If P starts in x, and $x \notin Dom(R_P)$, then P will never terminate.

By these conventions LD-relations are able to provide complete before/after descriptions of any program but we can revert to a simpler representation (functions) for those cases that arise most often.

Specifying Programs (1)

Specifications may *allow* behaviour that is never 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.

Definition:
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 interpretation of an LD-relation (specification or description) must be stated explicitly!

Tabular Descriptions and Specifications

Specification for a search program

$(\exists\, i,\ B[i] = x)$	$(\forall\, i,\ ((1 \le i \le N) \Rightarrow B[i] \ne x))$

j' |
present'=

B[j'] = x	*true*
true	false

$\wedge$ NC(x, B)

Description of a search program

$(\exists\, i,\ B[i] = x)$	$(\forall\, i,\ ((1 \le i \le N) \Rightarrow B[i] \ne x))$

j' |
present'=

$(B[j'] = x) \wedge (\forall\, i,\ ((j' < i \le N) \Rightarrow B[i] \ne x))$	*true*
true	false

$\wedge$ NC(x, B)

Readers and Writers

Formal methods discussions emphasise program development.

A successful program will have more readers than writers, because it will be maintained for many years.

The needs of reviewers and maintainers, are as important as the needs of program designers.

Programs should be presented in a way suitable for review and maintenance.

Proper decomposition into modules will reduce the complexity and length of programs.

But, some may still be quite long.

Displays are a documentation method for

- non-trivial,
- long,
- well-structured

programs.

The Limits of Human Comprehension

Human beings cannot easily understand long programs.

Studying a long program, we mentally decompose it.

We provisionally, assign a function to each part.

We try to convince ourselves that, if each part implements its assigned function, the whole program is correct.

We then try to confirm our starting hypothesis, and ....iterate until exhausted!

Reviewer should not have to guess a program's structure.

Program should be presented as a collection of small parts.

The intended function of each part should be precisely stated.

It should be possible to review those small parts separately.

Reviewer's responsibility is limited to checking small fragments.

Documentation Principles

An observation by Ludwig Wittgenstein:

- "Was sich überhaupt sagen läßt, läßt sich klar sagen; und wovon mann nicht reden kann, darüber muß man schweigen"
- Anything that can be said at all, can be said clearly, and what you cannot talk about, you shouldn't discuss at all.

What do we need in software documentation?

(1) Precision: Mechanical Interpretation, no ambiguity.

(2) Accuracy: not "almost right".

(3) Consistency: no contradictions, hence no duplication.

(4) Completeness: All visible behaviour covered, even when there is a choice.

(5) Verifiability: Organised for easy checking, a place for everything.

(6) Changeability, design for retrieval, no searching.

(7) No fuzzy "motherhood" statements!

(8) Scalability: Size of summary documents do not increase with size of program.

The Concept of Displays

Designers should present a program as a set of displays.

A *display* consists of three parts:

(1) A specification describing what the program should do.

(2) the program itself.

(3) specifications of subprograms invoked by this program.

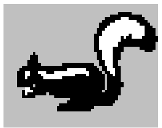
All programs in part 2 can be short because they invoke other programs.

A display presents a program in such a way that its correctness can be examined without looking at any other displays.

A *set of displays is complete* if, for each specification of a non-standard subprogram found in part (3), there is exactly one display in which this specification forms part (1). Completeness can be checked mechanically.

A *display is correct* if the program in part (2) will satisfy the specification in part (1) provided that all invoked programs satisfy the specifications in part (3).

A *set of displays is correct* if it is complete and all displays are correct.



Why use the Functional Approach

The display method is independent of the specification technique.

The display method is independent of programming language.

It works best with the functional approach.

[e.g. H.D. Mills, N.G. de Bruijn, Majster-Cederbaum, ...].

Functional Approaches "scale up" because there are no distinguished primitive programs. The "laws" of programming apply to all programs whether primitive (supplied to the programmers) or constructed by the programmers.

It can be applied even if the names represent huge programs.

Isn’t Stepwise refinement enough?

Stepwise refinement leads to long programs [Wirth example].

Stepwise refinement leads to repetitive programs.

Displays with functional specifications avoid these pitfalls.

NEVER write a long program and if you write one, do not expect someone else to read it!

Hierarchical control structure in programs

"Structured" Programming constructs have three very useful properties:

(1) programs constructed using them can be decomposed into a hierarchy of parts (with lower level parts completely contained in an upper level part) using simple parsers; those parsers need not even distinguish one identifier from another,

(2) the semantics of the total program can be determined from the semantics of its parts[1], using simple operations (cf. e.g. [14, 15]).

(3) semantics can be determined in a simple order: inner parts first.

The above properties make it easier to study a long program.

The Display Method takes advantage of those properties and is intended to be used for programs that have these properties.

---

1 It is an unfortunate property of today's programming languages that the semantics of the components depends on things out side those components.

Use of Data Abstractions

The best structured program will be difficult to explain and understand if it is presented in terms of complex data structures.

Data structures should be hidden by the introduction of *abstract data types*

Precise program documentation is not possible unless the abstract data type interfaces are precisely documented.

The following examples have been selected so that they can be understood without an understanding of module specifications.

A Simple Pseudo Code

Variables that are not declared are assumed to be globally declared.

B → P (guarded program)

- P will is executable only if B is ***true***. Appears only in guarded program lists.

(A | B| ....) (guarded program list)

- A, B, ... must be guarded programs. One of executable alternatives from the list will be executed. If no alternatives are executable the program aborts (fails to terminate).

A;B (execute B after A)

☞ (go): Program used to control iteration.

● (stop): Program used to control iteration.

it P ***ti*** (iterated program)

- Execute P at least once.
- Continue executing P according to execution of "☞" (go) or "●" (stop)
- Final "☞" or "●" determines whether or not the smallest enclosing loop will iterate.

Primitive Programs Used in This Example

$\lfloor x \rfloor$:the largest integer less than x.

card(s): the cardinality of set s.

skip: program that does nothing (No Operation) is used as a place holder.

$\Leftarrow$: assignment operation

Examples

$\lfloor 3.6 \rfloor = 3$

$\lfloor -2.5 \rfloor = -3$

card({2,4,6,8}) = 4

$((a \geq 0 \rightarrow a) \rightarrow a \Leftarrow a+1 \mid (a \leq 0 \rightarrow a) \rightarrow a \Leftarrow a-1)$

$n \Leftarrow 1;$

it (n ≤ N → (n ⇐ n +1;☞ | n > N → ●) ***ti***

Number 1 of Three Programs. Which one is right?

Problem:

$$NC(A) \wedge (\forall\, q, (1 \le q \le N \Rightarrow H'[q] = card(\{l \mid (\forall i, 0 \le i < \lfloor q \div 2 \rfloor \Rightarrow A[l+i] = A[l+q-1-i])\})))$$

```
( 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 );
       n ⇐ n +1; H[n]  ⇐ c; ☞ )
   | n > N  →     ● )
ti ) )
```

Number 2 of Three Programs. Which one is right?

Problem:

$$NC(A) \wedge (\forall\ q, (1 \le q \le N \Rightarrow H'[q] = card(\{l \mid (\forall i, 0 \le i < \lfloor q \div 2 \rfloor \Rightarrow A[l+i]=A[l+q-1-i])\}))$$

```
( 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 ))
```

Number 3 of Three Programs. Which one is right?

Problem:

$$NC(A) \wedge (\forall\ q, (1 \le q \le N \Rightarrow H'[q] = \mathrm{card}(\{ l \mid (\forall i, 0 \le i < \lfloor q \div 2 \rfloor \Rightarrow A[l+i] = A[l+q-1-i])\}))$$

```
( integer array H[1:N];
 (integer c; integer n; n ⇐ 0;
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 ) )
```

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 )
)
```

Display: An Example

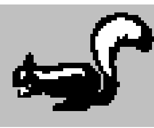
Problem: ctpal ≡

		true	
			H_1
c' =		card[a]({l | pal(A,l,n + l - 1)})		∧NC(n,A)
	H_2		G

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

Solution: ctpal ≡
(**integer** u, l; **boolean** p; $l \Leftarrow 1$; c ⇐ 0;
it (u ⇐ l + n -1;
(u ≤ N → (palul; (¬p → **skip** | p → c ⇐ c+1);
$l \Leftarrow l+1$; ☞)
| u > N → ●))
ti)
palul ≡: NC(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]))



Display: (palul) Palindrome from u to l

Note: p is declared globally as a **boolean**

Problem: palul $\equiv:$ NC(l,u,A) $\wedge$ (p' = pal(A,l,u))

Solution: palul $\equiv$

(**integer** i; i $\Leftarrow$ 0; p $\Leftarrow$ **true**;
it (i < $\lfloor$(u - l +1)÷2$\rfloor$ $\rightarrow$
(A[l+i] = A[u-i] $\rightarrow$ (i $\Leftarrow$ i + 1;☞)
|A[l+i] $\neq$ A[u-i] $\rightarrow$ (p $\Leftarrow$ **false**; ●))
| $\lfloor$(u - l +1)÷2$\rfloor$ $\leq$ i $\rightarrow$ ●)
ti)

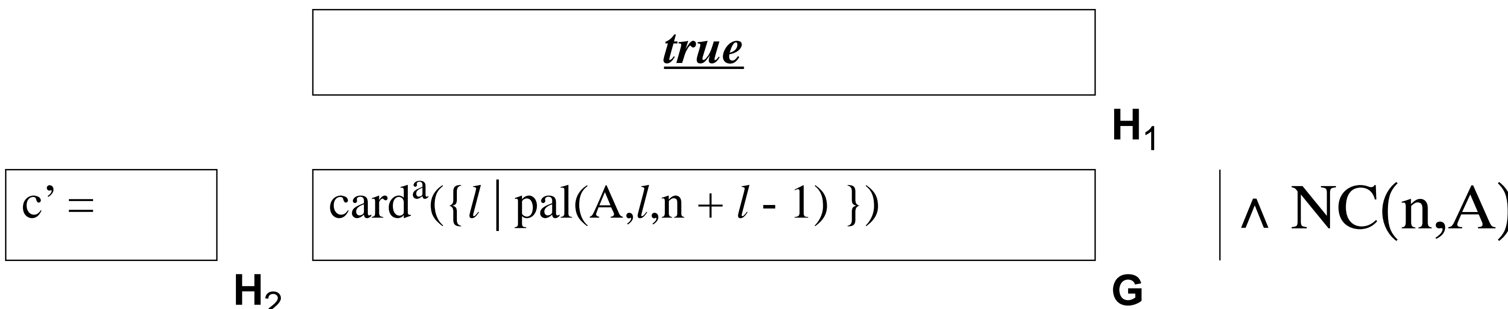
auxilliary function:

pal(A,b,c) = ((1 $\leq$ b $\leq$ c $\leq$ N) $\wedge$
($\forall$ i, 0 $\leq$ i < $\lfloor$(c- b +1)÷2$\rfloor$ $\Rightarrow$ A[b+i]=A[c-i])))

Note that part 3 of this display is empty.

Display: (ctpal) Counting Palindromes in A[1:N] (N > 1)

Problem: ctpal ≡



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

Solution: ctpal ≡

(**integer** u, *l*; **boolean** p; $l \Leftarrow 1$; $c \Leftarrow 0$;
it ($u \Leftarrow l + n - 1$;
($u \leq N \rightarrow$ (palul;($\neg p \rightarrow$ **skip** | $p \rightarrow c \Leftarrow c+1$);
$l \Leftarrow l+1$; ☞)
| $u > N \rightarrow$ ●))
ti)

palul ≡: $NC(l,u,A) \wedge (p' = pal(A,l,u))$

$pal(A,b,c) \equiv ((1 \leq b \leq c \leq N) \wedge (\forall i, 0 \leq i < \lfloor (c-b+1) \div 2 \rfloor \Rightarrow A[b+i]=A[c-i])))$

Producing a Palindrome Histogram

Problem: palhist ≡

$NC(A) \wedge (\forall\ q, (1 \leq q \leq N \Rightarrow H'[q] = card(\{l \mid pal(A, l, n + q - 1)\}))$

Solution: palhist ≡ (**integer** c, n; n ⇐ 1;

it (n ≤ N → (ctpal; H[n] ⇐ c; n ⇐ n +1;☞)

| n > N → ●)

ti)

ctpal ≡

		true	
			H_1
c' =		card({l | pal(A,l,n + l - 1) })		∧ NC(n,A)
	H_2		G

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.

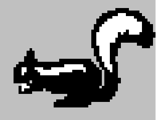
But,

How important is it to you?

- It will cost “up front time”, may save time and cost later.

Tabular Description of Reactor Shutdown Code

	‘|OKTT| = .FALSE.	(‘|OKTT| = .TRUE.) AND NOT !NoSensTrip!	(‘|OKTT| = .TRUE.) AND . !NoSensTrip!)
B(‘|PTB|,||DOW1||’)	B(‘|PTB|,‘||DOW1|| .OR.‘#TMASK(‘|PTB|)#)	**Table 4**	B(‘|PTB,‘||DOW1|| .OR.‘#TMASK(‘|PTB|)#)
B(‘#CN#,||DOW2||’)	B(‘#CN#,‘||DOW2||)	**Table 4**	B(‘#CN#,‘||DOW2||)
B(‘#CND#,||DOW2||’)	B(‘#CND#,‘||DOW2||)	**Table 4**	B(‘#CND#,‘||DOW2||)
||EX||’	‘||EX|| .OR. ‘|MASK|	‘||EX|| .OR. ‘|MASK|	‘||EX|| .OR. ‘|MASK|
|HI1|’	‘|HI1|	‘//HTL(5)// - ‘|HYS|	‘//HTL(5)// - ‘|HYS|
|HI2|’	‘|HI2|	‘//HTL(5)//	‘//THL(5)//
|LO1|’	‘|LO1|	‘//LTL(5)//	‘//LTL(5)//
|LO2|’	‘|LO2|	‘//LTL(5)// = ‘|HYS|	‘//LTL(5)// + ‘|HYS|
||MC||’	‘||MC||	**Table 4**	0
||PC||’	’||PC||	**Table 4**	0
B(j,|STBV|’), j = ‘|STB| + j-1, i in {1...5}	B(j,‘|STBV|)	**Table 3**	**Table 3**
B(j,|STBV|’), NOT (j in {‘|STB| + i-1}, i in {1...5})	B(j,(‘|STBV|) AND. ‘|UM|))	B(j,(‘||STW|| .AND. ‘|UM|))	
B(‘|STB| + i-1,||STW||’), i in {1...5}	B(‘|STB| + i-1, (‘||STW|| .OR. ‘|UM|))	**Table 3**	**Table 3**
B(j,||STW||’), NOT (j in {‘|STB| + i-1},	B(i,(‘||STW|| .OR. ‘|UM|))	B(i,‘||STW||)	B(i,‘||STW||)
B(‘|TIB|,||TIW||’)	B(‘|TIB|,(‘||TIW|| .OR.‘#TMASK(‘|TIB|)#))	B(‘|TIB|,(‘||TIW|| .AND.‘#FMASK(‘|TIB|)#))	B(‘|TIB|,(‘||TIW|| .AND.‘#FMASK(‘|TIB|)#))
||HIF(1...5)||’	‘||HIF(1...5)||	**Table 2**	**Table 2**
|I|’	‘|I|	6	6
||LOF(1...5)||’	‘||LOF(1...5)||	**Table 2**	**Table 2**



University of Limerick

More Tables

Table 2

	!AbvHi-Hys(i)!	!InHiHys(i)!	!InNorm(i)!	!InLoHys(i)!	!BlwLoHys(i)!
||HIF(i)||’	.FALSE.	‘||HIF(i)||	.TRUE.	.TRUE.	.TRUE.
||LOF(i)||’	.TRUE.	.TRUE.	.TRUE.	‘||LOF(i)||	.FALSE.

Table 3

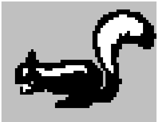
A* = [(‘||MC|| ≥ ‘|DEL|) OR (‘||MC|| < 0) OR (‘||PC|| + 1 ≥ ‘|PCL|) OR ((‘||PC|| +1) < 0)]

	A	NOT *A*
||PC||’	‘|PCL|	‘||PC|| + 1
||MC||’	‘|DEL|	‘||MC||
B(‘|PTB|,||DOW1||’)	B(‘|PTB,(‘||DOW1|| .AND.‘#FMASK(‘|PTB|#))	B(‘|PTB|,‘||DOW1||
B(‘#CN#,||DOW2||’)	B(‘#CN#,(‘||DOW2|| .AND.‘#FMASK(‘#CN#)#))	B(‘#CN#,‘||DOW2||
B(‘#CND#,||DOW2||’)	B(‘#CND#,(‘||DOW2|| .AND.‘#FMASK(‘#CND#)#))	B(‘#CND#,‘||DOW2||

Table 4

A* = [(‘||MC|| ≥ ‘|DEL|) OR (‘||MC|| < 0) OR (‘||PC|| + 1 ≥ ‘|PCL|) OR ((‘||PC|| +1) < 0)]

	A	NOT *A*
||PC||’	‘|PCL|	‘||PC|| + 1
||MC||’	‘|DEL|	‘||MC||
B(‘|PTB|,||DOW1||’)	B(‘|PTB,(‘||DOW1|| .AND.‘#FMASK(‘|PTB|#))	B(‘|PTB|,‘||DOW1||
B(‘#CN#,||DOW2||’)	B(‘#CN#,(‘||DOW2|| .AND.‘#FMASK(‘#CN#)#))	B(‘#CN#,‘||DOW2||
B(‘#CND#,||DOW2||’)	B(‘#CND#,(‘||DOW2|| .AND.‘#FMASK(‘#CND#)#))	B(‘#CND#,‘||DOW2||



Structure and Inspection

Well-structured programs can be decomposed by purely syntactic means.

Well-structured programs are much easier to inspect.

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

Our Initial Experience: Darlington Nuclear Power Generating Station[1]

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 AECB[2] 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".

$$(MAX_{[t-4,t]}\ (WL(t)) + MIN_{[t-4,t]}\ (WL(t))) \div 2 > 100$$

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

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

A Fourth (Unreasonable) Interpretation:

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

$$MIN_{[T-4,T]} [WL(t)] > 100$$

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.

Conventions

If R is a relation, then:

- "R" denotes the set of ordered pairs that constitutes this relation,
- "R(x,y)" denotes the characteristic predicate of the set R.

Let P be described by an LD-relation L = (R, C).

Let and let v_1, ..., v_k be program variables in P which form its data structure, $v = (v_1, ..., v_k)$. Then:

- " $'v_i$" (to be read "v_i *before*") denotes the value of the programming variable v_i before an execution of P,
- " v_i' " (to be read "v_i *after*") denotes the value of the variable v_i after a terminating execution of P.

Each pair in R will be of the form ('v, v').

We often write "R(,)" as an abbreviation of R(('a,'b,'c, ...), (a',b',c', ...)).

$$NC(v_1, \ldots, v_k) \Leftrightarrow (v_1' = {}'v_1) \wedge \ldots \wedge (v_k' = {}'v_k)$$

Parameters and side-effects

The specification of the procedure invocation will be written in terms of *actual* parameters.

In the declaration of this procedure *formal* parameters will be used.

Both, the specifications of subprograms appearing in the declaration, and statements in the declaration body must be written in terms of the formal parameters of the procedure (and its other local or non-local objects).

For sake of simplicity we will forbid any form of aliasing, e.g.:

- If more than one parameter is called by variable, then the actual parameters must be different variables.

If there are side-effects, then a variable external to the procedure body may not be passed as a parameter called by variable.

One More Example: The Problem of the Dutch national flag[1]

There is a data type colour $\stackrel{\mathrm{df}}{=}$ {blue,red,white}

There is an abstract data type "buckets".

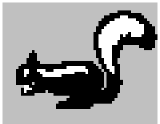
Variables of this type may be used as a vector of N "pebbles" of "colour" type, where $N \geq 0$ is an integer.

The only operations on v are: PUT(i,c), LOOK(i), SWAP(i,j)

Design a procedure to rearrange (if necessary) the pebbles in the order of the Dutch national flag

- Use no Arrays. Amount of space must not depend on number of pebbles.
- Call LOOK(i) no more than once for each value of i.[2]

1 Introduced and (perhaps) solved by E. W. Dijkstra in 1976
2 We will see that this requirement cannot be included in the specification without adding variables.



Defining the Data Structure

$1 \le k < r$:the kth bucket is in zone ER (number of buckets $r-1 \ge 0$)

$r \le k \le w$:the kth bucket is in zone X (number of buckets $w-r+1 \ge 0$)

$w < k \le b$:the kth bucket is in zone EW (number of buckets $b-w \ge 0$)

$b < k \le N$:the kth bucket is in zone EB (number of buckets $N-b \ge 0$)

This can be illustrated by the following figure:

ER	X	EW	EB

^1 ^r ^w ^b ^N

Initially, $r=1$, and $w=b=N$, so that the zones ER, EW, and EB are empty. The program then proceeds by incrementing r, and decrementing w and b, while making the necessary swaps, until the area marked “X” is empty ($r = w+1$).

Display 1

Specification

DutchFlag	**external variable: v**
$\mathbf{R}_0(,) = flag(v') \wedge same_colors('v,v')$	

Program

Procedure declaration:

```
procedure DutchFlag;
var r, w, b : integer;
begin
  r :=1; w := N; b := N;
  Rearrange(r, w, b)
end {DutchFlag}
```

Specifications of Subprograms

Rearrange(r, w, b)	**external variable: v**
$\mathbf{R}_1(,) = (('r = 1) \wedge ('w = N) \wedge ('b = N)) \Rightarrow (partial_flag(v',r',w',b') \wedge (w' = r'-1) \wedge same_colors('v,v'))$	

(on Display 2)

END OF DISPLAY 0

Display 2

Specification

Rearrange(r, w, b)	**external variable: v**
$\mathbf{R}_1(,) = ((\text{`r} = 1) \wedge (\text{`w} = N) \wedge (\text{`b} = N)) \Rightarrow (\textit{partial_flag}(v',r',w',b') \wedge (w' = r'-1) \wedge \textit{same_colors}(\text{`v},v'))$	

(from Display 1)

Program

Procedure declaration:

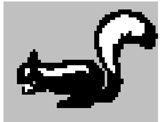
```
procedure Rearrange(var r, w, b : integer);
begin
  while w ≥ r do
    Decrease(r, w, b)
end {Rearrange}
```

Specifications of Subprograms

Decrease(r, w, b)	**external variable: v**
$\mathbf{R}_2(,) = (\textit{partial_flag}(\text{`v},\text{`r},\text{`w},\text{`b}) \wedge (\text{`r} \le \text{`w})) \Rightarrow (\textit{partial_flag}(v',r',w',b') \wedge ((w'-r') < (\text{`w}-\text{`r})) \wedge \textit{same_colors}(\text{`v},v'))$	

(on Display 3)

END OF DISPLAY 0



Display 3

Specification

Decrease(r, w, b)	**external variable: v**
$R_2(,) = (partial_flag(`v,`r,`w,`b) \wedge (`r \le `w)) \Rightarrow (partial_flag(v',r',w',b') \wedge ((w'-r') < (`w-`r)) \wedge same_colors(`v,v'))$	

(from Display 2)

Procedure declaration:

```
procedure Decrease(var r, w, b : integer);
var colr, colw : color;
begin
  IncR;
  if r < w then begin
    DecW;
    UseColw
  end {if};
  UseColr
end  {Decrease}
```

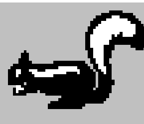
Specifications of Subprograms[1]

DecW	**external variables: v, r, w, b, colr, colw**
$R_3(,) = partial_flag(`v, `r, `w, `b) \wedge (`r < `w) \Rightarrow partial_flag(v', r', w', b') \wedge$	

(on Display 4)

	true
w' |	$(r' < w') \wedge (((r'+1) < w') \Rightarrow (v'_{w'} \ne white))$
colw' =	$v'_{w'}$

$\wedge$ NC(v,r,b,colr)



Display 3 (Continuted)

UseColr	external variables: v, r, w, b, colr, colw

(on Display 6)

$\mathbf{R}_5(,) = partial_flag(`v, `r, `w, `b) \wedge (`colr = `v_{`w}) \wedge$
$(`r \leq `w) \wedge ((`r < `w) \Rightarrow (`colr \neq red))$

$\Rightarrow$

$partial_flag(v', r', w', b') \wedge same_colors(`v,v') \wedge$

	‘colr =		
	red	**white**	**blue**
r' |	r' = ‘r + 1	NC(r)	NC(r)
w' |	NC(w)	w' = ‘w – 1	w' = ‘w – 1
b' |	NC(b)	NC(b)	b' = ‘b – 1

$\wedge$ NC(colr,colw)

Display to be continued

[1] Note*:* v_i is defined in part C of Lexicon.

Display 3 (Continuted)

UseColw	**external variables: v, r, w, b, colr, colw**

(on Display 7)

$\mathbf{R}_6(,) = partial_flag(`v, `r, `w, `b) \wedge (`colr = `v_{`r}) \wedge (`colw = `v_{`w}) \wedge$
$(`r < `w) \wedge (((`r+1) < `w) \Rightarrow (`colw \neq white))$
$\Rightarrow$
$partial_flag(v', r', w', b') \wedge same_colors(`v,v') \wedge (v'_{w'} = colr') \wedge$

	'colw =		
	red	**white**	**blue**
r' |	r' = 'r + 1	NC(r)	NC(r)
w' |	NC(w)	w' = 'w – 1	w' = 'w – 1
b' |	NC(b)	NC(b)	b' = 'b – 1

∧ NC(colr,colw)

END OF DISPLAY 3

Display 4

Specification

DecW	external variables: v, r, w, b, colr, colw

(from Display 3)

$\mathbf{R}_3(,) = partial_flag('v, 'r, 'w, 'b) \wedge ('r < 'w) \Rightarrow$

$partial_flag(v', r', w', b') \wedge$

	true
w' |	$(r' < w') \wedge$ $(((r'+1) < w') \Rightarrow (v'_{w'} \neq \text{white}))$
colw' =	$v'_{w'}$

$\wedge$ NC(v,r,b,colr)

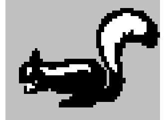
Program

```
{DecW}
colw := LOOK(w);
while (colw = white) and ((r+1) < w) do begin
  w := w–1; colw := LOOK(w)
end
```

Specifications of Subprograms

Empty

END OF DISPLAY 0



Display 5

Specification

IncR	external variables: v, r, w, b, colr, colw

(from Display 3)

$\mathbf{R}_4(,) = \mathit{partial_flag}(`v, `r, `w, `b) \wedge (`r \leq `w) \Rightarrow$

$\mathit{partial_flag}(v', r', w', b') \wedge$

	true
r' |	$(r' \leq w') \wedge$ $((r' < w') \Rightarrow (v'_{r'} \neq red))$
colr' =	$v'_{r'}$

$\wedge$ NC(v,w,b,colw)

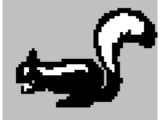
Program

```
{IncR}
colr := LOOK(r);                    { v is an implicit variable used by LOOK }
while (colr = red) and (r < w) do begin
  r := r+1; colr := LOOK(r)
end
```

Specifications of Subprograms

Empty

END OF DISPLAY 0



Display 6

Specification

UseColr	external variables: v, r, w, b, colr, colw

(from Display 3)

$\mathbf{R}_5(,) = partial_flag(`v, `r, `w, `b) \wedge (`colr = `v_{`w}) \wedge$
$(`r \le `w) \wedge ((`r < `w) \Rightarrow (`colr \ne red))$
$\Rightarrow$
$partial_flag(v', r', w', b') \wedge same_colors(`v,v') \wedge$

	‘colr =		
	red	**white**	**blue**
r’ |	r’ = ‘r + 1	NC(r)	NC(r)
w’ |	NC(w)	w’ = ‘w – 1	w’ = ‘w – 1
b’ |	NC(b)	NC(b)	b’ = ‘b – 1

∧ NC(colr,colw)

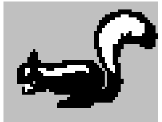
Program

```
{UseColr}
case colr of
   red:      r := r+1;
   white:    w := w–1;
   blue:     begin SWAP(w,b); w := w–1; b := b–1 end
end
```

Specifications of Subprograms

Empty

END OF DISPLAY 0



Display 7

Specification

UseColw	external variables: v, r, w, b, colr, colw

(from Display 3)

$\mathbf{R}_6(,) = partial_flag(‘v, ‘r, ‘w, ‘b) \wedge (‘colr = ‘v_{‘r}) \wedge (‘colw = ‘v_{‘w}) \wedge$
$(‘r < ‘w) \wedge (((‘r+1) < ‘w) \Rightarrow (‘colw \neq white))$

$\Rightarrow$

$partial_flag(v’, r’, w’, b’) \wedge same_colors(‘v,v’) \wedge (v’_{w’} = colr’)$[a] $\wedge$

	‘colw =		
	red	**white**	**blue**
r’ |	r’ = ‘r + 1	NC(r)	NC(r)
w’ |	NC(w)	w’ = ‘w – 1	w’ = ‘w – 1
b’ |	NC(b)	NC(b)	b’ = ‘b – 1

$\wedge$ NC(colr,colw)

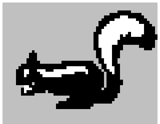
a. The post-condition $v’_{w’}$ = colr’ is redundant and has been added for ease of comprehension.

Program

```
{UseColw}
case colw of
   red:      begin SWAP(r, w); r := r+1 end;
   white:    w := w–1;
   blue:     begin SWAP(w, b); w := w–1; b := b–1; SWAP(r,w) end
end
```

Specifications of Subprograms: Empty

END OF DISPLAY 0



Lexicon

A. Auxiliary functions

card: set → integer

$card(s) \stackrel{df}{=} | s |$ (i.e. number of elements in the set s)

flag: buckets → boolean

$flag(v) \stackrel{df}{=} \exists r,b\ [partial_flag(v,r,r-1,b)]$

partial_flag: buckets × integer × integer × integer → boolean

$$partial_flag(v,r,w,b) \stackrel{df}{=} (1 \le r) \wedge (r-1 \le w) \wedge (w \le b) \wedge (b \le N) \wedge$$
$$\forall i\ (1 \le i \le N)\ [\ ((i < r) \Rightarrow (v_i = \text{red})) \wedge$$
$$((w < i \le b) \Rightarrow (v_i = \text{white})) \wedge$$
$$((b < i) \Rightarrow (v_i = \text{blue}))\ ]$$

Note: v_i is defined in part C of this Lexicon.

same_colors: buckets × buckets → boolean

$same_colors(v1,v2) \stackrel{df}{=}$

$$(card(\{i \mid (1 \le i \le N) \wedge (v1_i = \text{red})\}) = card(\{i \mid (1 \le i \le N) \wedge (v2_i = \text{red})\})) \wedge$$
$$(card(\{i \mid (1 \le i \le N) \wedge (v1_i = \text{white})\}) = card(\{i \mid (1 \le i \le N) \wedge (v2_i = \text{white})\})) \wedge$$
$$(card(\{i \mid (1 \le i \le N) \wedge (v1_i = \text{blue})\}) = card(\{i \mid (1 \le i \le N) \wedge (v2_i = \text{blue})\}))$$

Lexicon

B. Pascal external definitions and declarations

```
const N = {literal non-negative integer}
type color = (red, white, blue);
type buckets =    {vector(N, color) - cf. part C of this Lexicon}
var v : buckets;
procedure LOOK(i : integer);
  {cf. part C of this Lexicon}
procedure SWAP(i, j : integer);
  {cf. part C of this Lexicon}
```

C. vector(n,elem) Module Interface Specification

(0) CHARACTERISTICS

- type specified: vector(n,elem)
- features: single-object, generic
- foreign types: elem, <integer>, <positive_integer>
- foreign types: n: <positive_integer>, elem

Module Specification of Buckets

(1) SYNTAX

ACCESS-PROGRAMS

Program Name	Arg#1	Arg#2	Value Type
LOOK	<integer>: V		elem
PUT	<integer>: V	elem:V	
SWAP	<integer>: V	<integer>: V	

(2) CANONICAL TRACES

canonical(T) $\Leftrightarrow$ T = $[PUT(i, e_i)]_{i=1}^{n}$

_ $\equiv$ $[PUT(i, _)]_{i=1}^{n}$

EQUIVALENT NOTATION FOR TRACES

Trace	Equivalent notation
v.LOOK(i)➢	v_i

Equivalence Assertions

(3) EQUIVALENCES

T.LOOK(i) ≡> T

T.PUT(i, e) ≡>

Condition	Equivalence
$\neg(1 \le i \le n)$	%wrong_index%
$1 \le i \le n$	T1.PUT(i,e).T2 where T=T1.PUT(i,x).T2

T.SWAP(i, j) ≡>

<table>
<tr><th colspan="2">Condition</th><th>Equivalence</th></tr>
<tr><td colspan="2">$\neg((1 \le i \le n) \wedge (1 \le j \le n))$</td><td>%wrong_index%</td></tr>
<tr><td rowspan="3">$(1 \le i \le n) \wedge (1 \le j \le n) \wedge$</td><td>$(i < j)$</td><td>T1.PUT(i,x).T2.PUT(j,y).T3
where
T = T1.PUT(i,y).T2.PUT(j,x).T3</td></tr>
<tr><td>$(i = j)$</td><td>T</td></tr>
<tr><td>$(i > j)$</td><td>T1.PUT(j,x).T2.PUT(i,y).T3
where
T = T1.PUT(j,y).T2.PUT(i,x).T3</td></tr>
</table>

(4) RETURN VALUES

Program Name	Argument No	Value
LOOK	Value	e where #0 = T1.PUT(#1,e).T2

Conclusions

Programs must be understood in small chunks

Programs should be presented in small chunks

NEVER read (or write) a long program.

Precise specifications/descriptions are essential

Size of specification not based on program size.

Without precise descriptions of program structure, even great programmers will err.

Correctness can be checked “by head”

Completeness, consistency, can be checked by machine.

Tools advantageous in daily use.

Review: What must you do

(1) Begin with a specification of what you want the critical program to do.

(2) Decompose the program:

- Introduce modules/data abstractions/objects wherever possible and provide abstract specifications for them
- Use hierarchical decomposition as demonstrated earlier.

(3) Produce a set of displays based on the decomposition.

(4) Make sure that the displays are complete and consistent

- Every specification at the bottom of a page must appear at the top of another.
- There can be only one implementation display for each program.

(5) Verify/Inspect each display. Use tabular structure to decompose the inspection process.

(6) When errors in specifications are found, mark all displays that include those specifications as requiring a repeat inspection.

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.