University of Limerick

A Family of Mathematical Documents for Professional Software Documentation

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)
University of Limerick,[1] Ireland

Software documentation is one of the most important, and least studied, problems in Software Engineering. Much time is spent trying to recover, digest, and understand information that is either missing, hard to find, inaccurate, inconsistent, or incomplete.

We propose a family of documents characterised by two major decisions:

- Document content (not format) is specified using a relational model.
- Relations are represented using mathematical expressions in tabular form.

This talk will motivate these decisions, illustrate the relational definitions of document content, illustrate the concept of tabular expressions, and discuss the potential applications of such documents in software development.

[1] On leave from McMaster University in Hamilton Ontario Canada

Outline

(1) Software engineering research

(2) Software engineering vs. project management.

(3) The role of documentation in traditional engineering

(4) Why models are not documentation

(5) What documents are needed in software development

(6) Content before form: relational models

(7) Relational approach to requirements

(8) Module interface documentation

(9) Terminating program documentation

(10) Making Documents both readable and precise (tabular expressions)

(11) Disciplined software design and assessment through documentation

Software Engineering Research

- Study problems that are relevant to software development
- Find the real problem, don't attack the symptoms
- Don't fill pages with the implications of arbitrary decisions
- Know and apply work of the past (new buzzwords notwithstanding)
- Use traditional science and mathematics to get results that are demonstrably true, not just anecdotal, intuition, speculation, or guesses.
- Confirm that mathematically sound methods can actually (not just theoretically) be used by software development (empirical trials).
- Confirm that things that *seem* to work actually have a sound basis.

Some Personal Bias

"It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and in hardware design, is unsound and clumsy because the people who do it have not any clear understanding of the fundamental design principles of their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing."
--------(Christopher Strachey). -

The **finite-**state machine (Mealy, Moore, Turing) is the model that we need to solve practical digital computer problems. Many of the problems that plague the industry today are a direct result of **finite** limitations on storage and computing capacity. Work that deals with "more powerful" models will ignore the real problems. (Corollary: algorithmic complexity is important, computability is not!)

Conventional (applied) mathematics is quite capable of handling the problems that arise in documenting programs. Our job is to learn how to apply old concepts, inventing new ones only when really necessary.

Specification languages are unimplemented programming languages!

Software Engineering is not Project Management

Management is getting things built *without* knowing exactly what they are.

Engineering demands a *complete understanding of* a product's properties.

Software projects are hard to manage because they are badly designed and inadequately documented.

- Imagine trying to manage car development if we could not specify the size and pitch of bolts and nuts (or did not bother to do it).

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

Managers are important, but **first** we need Engineers.

With good Engineers, management is much easier.

The Role of Documentation in Engineering

Engineers do not begin bridge development by placing rocks in the water.

Engineering design proceeds through the development, review, revision, and then refinement of a series of documents.

Each document conforms to the previous (approved) document but adds further design information for further review.

Construction begins when complete design documentation is available.

This is the least expensive way to get the necessary review and approval.

“Refactoring” of a bridge or other structure is too expensive and difficult.

Software is not different!

Software development *should* not be different, but it is!

The Role of Documentation in Software Development Today

In software we often begin by the equivalent of throwing rocks in the water.

- The first, and usually the only, precise documents are code.
- Much documentation is written afterwards (by uninformed people).
- There is no opportunity to review decisions until code has been written.
- Design work doesn't conform to earlier documents, "They are history".
- Documents are often inconsistent.
- A big part of a programmer's job is searching for information (looking to see where the rocks landed).
- When they find information, the next step is to verify that it is accurate.
- Only then can the "real work", writing or modifying code, begin.

On the Importance of Documentation

The vast majority of software problems that I see are either caused, or exacerbated by, the lack of precise, complete, and correct design documentation.

- There are misunderstandings during design.
- Mistakes during implementation
- Counterproductive changes during “maintenance”.
- Wasted time trying to find out what is intended.

This was true 35 years ago.

It is no less true today.

Documentation is a problem that is both real and old.

We won’t be professional software developers until we know how to produce really useful documentation.

There is no more important research problem.

Models are not (necessarily) Documentation

Definitions:

- A *description* of a product provides accurate information about that product.
- A *specification* states the requirements for a product.
- A *document* may be either a description or a specification.
- A *model* is something with some of the properties of a product but may have properties that differ from those of the product.

Models can be physical products that are more easily produced, viewed, and studied than the actual product or mathematical models.

Models are useful", but, if they are not accurate "documents", be wary/

Computer Scientists without engineering educations often confuse these.

Descriptions, specifications, and models cannot be distinguished on the basis of their form. There must be a statement of intent accompanying them.

Descriptions,

A *description* of a product provides information about that product.

- A description need not be complete. We can use a collection of descriptions.
- What the description states must be true of the product, else it is wrong.
- A description is useful if (and only if) it is easier to derive information from the description than to derive it from the product.
- If we use more than one description, we should minimize the overlap. Overlap leads to inconsistency.
- If a product satisfies a specification, that specification is also a description.

Specifications

A *specification* states the **requirements** for a product.

- If the specification is not an accurate description, the product may be wrong.
- If the specification is true of a product, but the product is not satisfactory, the specification is wrong. It may be inaccurate or just incomplete.
- If the specification is not true of a product, but the product is satisfactory, the specification is wrong (either overspecification or false requirements).
- An implementor that accepts the task of supplying a product that meets a specification is obligated to make the specification a true description. A professional will check that a specification can be implemented before starting.
- The specification will describe externally visible behaviour (i.e. information that can be detected by the user), this is the binding information.
- The specification should not contain (invisible) implementation suggestions; if these are useful, they should be supplied separately.

Descriptions, Specifications, and Models (again)

Specifications and Descriptions are both essential in Engineering.

Models are useful, **but also dangerous**.

It is important to know which one you have.

In CS there is a long history of careless use of these terms. Many "specifications" are actually models or partial (often inaccurate) descriptions.

Most "specification" languages were really modelling languages. Often they turned out to be implementation languages (e.g. Algol 60).

MBD (Model Based Design) is not the right approach.

DBD (Documentation Based Design) is much sounder.

Complete Family of Consistent, Complementary Documents

Industry plagued by documents that have inconsistent notations, inconsistent contents and gaps in their coverage.

Our proposal:

- A relational definition of the content of each document is provided.
- Definitions are intended to minimize overlap
- Consistency checks are possible
- Same (tabular) notation used in all documents.

What Documents Are Now in the Family?

(1) "*System Requirements Document*": Treats computer system as "black-box".

(2) "*System Design Document*": Describes computers and communication.

(3) "*Software Requirements Document*" [(1) + (2) or (1) - (2)].

(4) "*Software Function Specification*": Describes actual software behaviour.

(5) "*Software Module Guide*": How to find your module. (informal document).

(6) "*Module Interface Specifications*": Treat each module as black-box.

(7) "*Uses Relation Document*": Range and domain comprise module programs.

(8) "*Module Internal Design Documents*": Describe data structure and program functions.

• • • •

Content Before Form: Relational Definitions

Each document is defined as a description of some mathematical relations.

We define the contents by defining the relations.

There are many ways to describe a relation.

- f(x) = 2*x+ 2 and f(x) = 2*(x+1) describe the same relation (function).

Defining the content does not determine format or language.

Software examples:

- Program effect described by a relation from data-state to data-state. (x,y) in that function if program started in x can terminate in y.
- Relation can be described by precondition/postcondition pair (extra variables often needed).
- Relation can be described by Dijkstra's "wp" and '"wlp.
- Function can be described by predicates (Hehner).
- Function can be described by concurrent-assignment statement (Mills).

The Advantages of Relational Models

I do not understand why “formal methods” have separated from “Relational Models”.

There is a large community (e.g Schmidt, Brink, Mili, Desjarnais, Khedri, and many more) who have studied relational methods in CS.

By abstracting from representational issues, the basic properties become crystal clear.

Schmidt argues that you can reduce the “order” of the logic needed.

Issues such as non-determinism, which seem to be new in parts of the FM community, are old in the RM community.

Leading FM researchers tell me that they find RM useless. I find them very useful.

They allow me to separate form (notation) from content.

More on Relational Approach to Documentation

The following paper, defines the document content precisely:

Parnas, D.L., Madey, J., "Functional Documentation for Computer Systems Engineering", Science of Computer Programming (Elsevier) vol. 25, number 1, October 1995, pp 41-61

This (very abstract) paper tells you what should be in each document. It does not say how to format the information or what notation to use.

It does not give you section headings. This is different from most documentation standards.

- No relation described in two documents.
- Lists of variables must be consistent
- Each document can be checked against other documents

Illustration: The Two-Variable Approach to Requirements:

The “traditional approach” used in control systems is based on two assumptions.

- The system has inputs and outputs.
- The outputs are a mathematical function of the inputs.

In this model, the inputs are the actual physical inputs to the hardware/ software system (bits).

The mathematical functions are often quite complex and hard to describe.

Review by application experts is hard to get.

Illustration: The Four Variable Approach to Requirements

Outside the system there are physical variables, some monitored, some controlled, some both.

Peripheral devices sense monitored variables and determine computer inputs, read computer outputs and influence the value of the controlled variables.

Some variables can be both monitored and controlled.

Otherwise, the sets are disjoint.

M-variables-->I/O device-->inputs-->software-->outputs-->I/O device-->C-variables [1]

The System Requirements specify the desired relation between the monitored (M) and the controlled (C) variables.

Subject-matter experts review requirements in terms of monitored/controlled variables not the inputs and outputs.

Software requirements are described using input and output variables.

[1] Green denotes variables, cyan denotes hardware, red denotes software.

How to Document System Requirements

The first steps are to:

- Identify monitored variables (m_1, m_2, •••, m_n).
- Identify controlled variables (c_1, c_2, •••, c_p).

The primary *monitored* variables are things outside the system whose values should influence the output of the system. Examples:

- customer meter reading
- steam temperature
- time of day that product to arrives at a manufacturing station

The primary *controlled* variables are things outside the system whose values should be determined by the system. Examples:

- what the operator sees on a computer screen
- what appears on a bill or other statement
- positions of a tool being controlled by the computer.

In practice, you rarely find a complete list of these variables.

Defining the Meaning of Monitored and Controlled Variables

This is not a Computer Science problem; it is conventional engineering.

- It may require drawing diagrams.
- There must be a coordinate system explicitly defined.
- The units must be explicitly defined.
- Precision is needed. For example, the time that a product arrives at a station is different from the time that the station starts to work on it, etc.

It is essential that everyone have a common set of definitions.

These things can and should be defined before programmers get involved.

These things have nothing to do with programming languages, support software, etc.

Choosing the “right” coordinate systems, etc. can make a big difference.

What Information Must Be in a Systems Requirements Document?

Answer: Definitions of the following relations[1]:

Relation NAT

- domain contains values of $\underline{m}^t$,
- range contains values of $\underline{c}^t$,
- $(\underline{m}^t, \underline{c}^t)$ is in NAT if and only if nature permits that behaviour.

Relation REQ

- domain contains values of $\underline{m}^t$,
- range contains values of $\underline{c}^t$,
- $(\underline{m}^t, \underline{c}^t)$ is in REQ if and only if the system should permit that behaviour.

[1] $\underline{m}^t$ denotes a mathematical function that describes the value of m as a function of (a real variable) time.

Can We Check System Requirements For Consistency?

It should be true that,

$$\text{domain}(\text{REQ}) \supseteq \text{domain}(\text{NAT}).$$

Otherwise the document is incomplete.

The relation REQ can be considered *feasible with respect to NAT* if (1) holds and,

$$\text{domain}\,(\text{REQ} \cap \text{NAT}) = (\text{domain}(\text{REQ}) \cap \text{domain}(\text{NAT})).$$

Otherwise you are asking the system to break the laws of nature.

Checking these properties shows completeness and feasibility, not correctness!

Published examples that purport to show the inadequacy of this model violate these rules, i.e. make contradictory assumptions.

Other Documents

Software Requirements = SOF(I,O)

Input Device Description = IN(M,I)

Output Device Description = OUT(I,O)

Cross Document Checks

Input variables in SOF must be those in IN

Output variables in OUT must be those in SOF

Monitored variables in IN must be those in REQ

Controlled variables in OUT must be those in REQ

Variables in REQ and NAT must be the same.

For the software to be acceptable, SOF must satisfy:

(6) $\forall \underline{m}^t \; \forall \underline{i}^t \; \forall \underline{o}^t \; \forall \underline{c}^t \; [IN(\underline{m}^t, \underline{i}^t) \wedge SOF(\underline{i}^t, \underline{o}^t) \wedge OUT(\underline{o}^t, \underline{c}^t) \wedge NAT(\underline{m}^t, \underline{c}^t) \rightarrow REQ(\underline{m}^t, \underline{c}^t)]$

Using functional notation:

(6a) $\forall m^t \; [m^t \in \text{domain}(NAT) \rightarrow (REQ(m^t) = OUT(SOF(IN(m^t))))]$

The Trace Function Approach to Module Interface Documentation

A trace is a sequence of event descriptors.

Each descriptor describes the before and after value of all variables accessible by the module.

Global variables whose values are changed are "*output variables*".

Those whose values change module behaviour are "*output variables*".

If a module program is to be invoked, we consider its name to be the value of an input variable.

For every output variable, we describe its value as a function of the trace, (or the required relation between its value and the trace.)

Module Internal Design and Documentation

If we already have the module specification, we need to document:

1. **The complete data structure.**
   - Often data elements are introduced piecemeal - we need know it all.
   - Brooks, “Show me your algorithms and I will ask to see your data structure; show me your data structure and I may not need to see your algorithm.”
2. **The interpretation of that data structure (abstraction function).**
   - Every possible data state corresponds to a trace, but which one.
   - **The abstraction function maps from concrete states to the abstract representation.**
3. **The effect of each program on the data structure.**
   - This can be done by H.D. Mills’ “program functions” mapping from concrete states to concrete states.
   - We actually use a relational model to allow non-determinism.
   - This model has been known (in one form) for 40 years or more.

LD-Relations

Definition 11:

A *binary relation* R on 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. $R = \{(p,q): U \times U \mid R(p,q) = \underline{\text{true}}\}$. 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". θ

Definition 12:

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. $C_L \subseteq Dom(R_L)$. θ

Alternative Behavioural Descriptions

A standard (decades older) alternative uses a single relation with a special symbol (e.g. "⊥") representing non-termination.

- This is mathematically (theoretically) equivalent. You can derive the LD-Relation from it and vice versa.
- ⊥ is not a state and exceptions must be made for it when giving the algebra of relations. Certain statements of algebraic laws become more complex.
- The characteristic predicate of the relation cannot be written in terms of variable values alone. One must add something. ***This is a practical disadvantage***.

LD-Relations can be characterised by predicates. This gives us "predicative programming" (Hehner) but for the completely general behavioural description you should use two predicates.

We define defaults to handle the common (deterministic termination) case.

Other Alternative Behavioural Descriptions

(1) Before/after descriptions do not describe invariants. A single predicate suffices, but users of a program do not need to know the invariant of its loops. This is part of internal documentation.

(2) VDM does not describe the behaviour of a program if the precondition doesn't hold. The VDM model does not allow one to distinguish certain programs that have distinct before/after behaviour.

(3) Pre/Post conditions are a fiction. We are not really interested in two separate conditions and are often forced to add (otherwise unnecessary) variables so we can describe a relation. Using relations helps practitioners avoid certain common errors that arise from forgetting this.

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

Definition 17:

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 $\langle x,\ldots,y\rangle \in Exec(P,S)$, and $C_P = S_P$.[1] L_p is called the *LD-relation of P* θ

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

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

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

The following follow from Definition 17:

- If P starts in x and $x \in C_P$, P will always terminate; if $(x, y) \in R_P$, P may terminate in y.
- If P starts in x, and $x \in (Dom(R_P) - C_P)$, the termination of P is non-deterministic; in this case, if $(x, y) \in R_P$, when P is started in x, the execution may terminate in y or may not terminate.
- If P is started in x, and $x \notin Dom(R_P)$, then the execution will not terminate.
- If P is a deterministic program, the relational component, R_P, is Mills' program function and C_P (which will be exactly $Dom(R_P)$) need not be written. Hence, our approach is "upward compatible" with Mills' [2,3].

These conventions allow complete before/after descriptions of any program with convenience for cases that arise most often.

Using LD-Relations as Before/After Descriptions

LD-relations have advantages over other, more popular, before/after descriptions.

- They provide complete before/after descriptions of non-deterministic programs.
- They can be described by giving the characteristic predicates of the Relation and Competence Set; those predicates can be expressed in terms of values of the actual, program variables.

The last property is of great practical value.

It allows us to use completely conventional (classical) mathematics yet still provide descriptions in terms of things that programmers know about.

Specifying Programs

Specifications may allow behaviour not actually exhibited by a satisfactory program. We can use LD-relations as before/after specifications. We must describe the relation between the description and a specification.

Definition 18:

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 the LD-relation* L_S, **iff** $C_S \subseteq C_P$
and $R_P \subseteq R_S$, and
(2) P *satisfies the specification* S, **iff** L_p satisfies at least one element of S. θ

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

Specifying Programs

The following follow from Definition 18.

- 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 interpretation of LD-relations *must* be stated explicitly!

How can Documents be Both Precise and Readable?

This is precise:

$$(((\exists\, i,\ B[i] = \text{`x}) \wedge (B[\text{j'}] = x) \wedge (\text{present'} = \textbf{true})) \vee ((\forall\, i,\ ((1 \le i \le N) \Rightarrow B[i] \ne x)) \wedge (\text{present'} = \textbf{false}))) \wedge (\text{`x} = \text{x'} \wedge \text{`B} = \text{B'})$$

But,

- Few people want to read even simple examples like this.
- You have to parse it all and understand a lot before you can find what you want.

Lets try anyway - just once.

$$(((\exists\, i,\ B[i] = \text{`x}) \wedge (B[\text{j'}] = x) \wedge (\text{present'} = \textbf{true})) \vee ((\forall\, i,\ ((1 \le i \le N) \Rightarrow B[i] \ne x)) \wedge (\text{present'} = \textbf{false}))) \bigwedge (\text{`x} = \text{x'} \wedge \text{`B} = \text{B'})$$

$$(((\exists\, i,\ B[i] = \text{`x}) \wedge (B[\text{j'}] = x) \wedge (\text{present'} = \textbf{true})) \bigvee ((\forall\, i,\ ((1 \le i \le N) \Rightarrow B[i] \ne x)) \wedge (\text{present'} = \textbf{false})))$$

$$(((\exists\, i,\ B[i] = \text{`x}) \bigwedge (B[\text{j'}] = x) \wedge (\text{present'} = \textbf{true}))$$

$$((\forall\, i,\ ((1 \le i \le N) \Rightarrow B[i] \ne x)) \bigwedge (\text{present'} = \textbf{false})))$$

$$(\forall\, i,\ ((1 \le i \le N) \Longrightarrow B[i] \ne x))$$

It is not more formal or difficult than a programming language but parsing is difficult.

How can Documents be Both Precise and Readable?

The following is readable

"Set i to indicate the place in the array B where x can be found and set present to be **true**. Otherwise set present to be **false"**

but vague and unclear:

- What do you do if the array is of zero length?
- What do you do if x is present more than once?
- Are you allowed to change B or x?
- What does the "otherwise" mean: Does it mean, if you don't do what you should, or if there is no place in the array where x can be found, or if there are many places where x can be found?

We have all seen worse examples of such sentences.

How can Documents be both Precise and Readable?

This is readable and more precise than the text above:

Specification for a search program

	x can be found in B	x can not be found in B
j' must be present' must be	a place where x can be found in B	any number at all
	true	**false**

How can Documents be both Precise and Readable?

This is precise and readable (by trained people).

Specification for a search program

	$(\exists\, i, B[i] = x)$ [a]	$(\forall\, i, \neg(B[i] = x))$
j' |	B[j'] = x	*true*
present'=	true	false

$\wedge$ NC(x, B)

a. These tables depend on using a logic in which evaluating a partial function outside of its domain yields "***false***" for all built-in predicates.

1 The first can be input to mathematics based tools but is hard for people.
2 The second seems clear but does not answer key questions.
3 The third is clearer but does not answer one key question and cannot be input to reliable tools.
4 The fourth is complete and could be processed by tools. It is, in theory, equivalent to the first, but in practice much better.

The table parses the expression for the reader, but is still mathematics.

Tabular Notation Is Useful For All Software Design Documents

We first "discovered it" in the requirements area, but it can be used for

- **system design documents**
- **module interface specifications**
- **module internal design documents**

These documents can be tested for completeness and consistency.

Producing these documents is part of "the discipline".

Checking them is the remainder of the discipline.

They make the programming *much faster* and *more reliable*.

But, most important:

They can be read by human beings.

- They were read and corrected by pilots, engineers, telephone operators, and even managers.

They help us to reduce the number of errors.

University of Limerick

Requirements for a Keyboard Testing System

N(T)=

	¬(T = _) ∧		
T = _	**1 < N(pr(T))< L**	**N(pr(T))= L**	**N(pr(T))=1**
nt(T)= N(pr(T))		N(pr(1,T))+ 1		2
¬(nt(T) = N(pr(T))) ∧ ¬(nt(T) = esc)	1	N(pr(1,T)) - 1	N(pr(1,T)) -1	1
¬(nt(T)= N(pr(T))) ∧ (nt(T) = esc) ∧ ¬(nt(pr(T)) = esc)		N(pr(1,T))	N(pr(1,T))	1
¬ (nt(T)= N(pr(T))) ∧ (nt(T) = esc) ∧ (nt(pr(T)) = esc)				

Status(T)=

	¬(T = _) ∧		
T = _	**1 < N(pr(T))< L**	**N(pr(1,T))= L**	**N(pr(1,T))=1**
nt(T)= N(pr(T))		“incomplete”	“pass”	incomplete
¬(nt(T) = N(pr(T))) ∧ ¬(nt(T) = esc)	“incomplete”	“incomplete”	“incomplete”	“incomplete”
¬(nt(T)= N(pr(T))) ∧ (nt(T) = esc) ∧ ¬(nt(pr(T)) = esc)		“incomplete”	“incomplete”	“incomplete”
¬ (nt(T)= N(pr(T))) ∧ (nt(T) = esc) ∧ (nt(pr(T)) = esc)		“fail”	“fail”	“fail”

T is the input history. No internal structures are mentioned.

Queue: Internal Design (part I)

Data Structure Description and abbreviations

CONSTANTS

Constant Name	Definition
QSIZE	12

TYPES

Type Name	Definition
<qds>	array[0..QSIZE-1] of integer

VARIABLES

Type Definition/Name	Variables	Initial Values
<qds>	DATA	“Don’t Care”
0..QSIZE-1	F, R	“Don’t Care”
<boolean>	FULL	“Don’t Care”
<boolean>	old	false

Abbreviations:

edge = $(R = F + 1) \vee (F = QSIZE\text{-}1) \wedge (R = 0)$ =

<qs> = $qds \times 0..QSIZE\text{-}1 \times 0..QSIZE\text{-}1 \times boolean$

The Abstraction Function[1]

af: <qs> → <queue12>

af(DATA,F,R,FULL,old) =

(¬ *edge* ∨ FULL) ∧ (F ≥ R) ∧ old	Q12INIT.(DATA[F]).(DATA[F−1]). … (DATA[R])
(¬ *edge* ∨ FULL) ∧ (F < R) ∧ old	Q12INIT.(DATA[F]). … .(DATA[0]).(DATA[QSIZE-1]). … .(DATA[R])
edge ∧ ¬ FULL ∧ old	Q12INIT
¬ old	

[1] The above table explains how the data are intended to be interpreted. It is redundant and used for checking.

University of Limerick

Queue: Program Functions

pf_Q12INIT =

F' =	0
R' =	1
FULL' =	false
DATA' |	true
old =	true

gpf_ADD(a) = NC(F) ∧ ∀j (j ≠ R') [NC(DATA[j])] ∧ NC(a) ∧

	(‘R = 0) ∧ old ∧			(‘R ≠ 0) ∧ old ∧			
	‘*edge* ∧		¬ ‘*edge*	‘*edge* ∧		¬ ‘*edge*	¬ old
	‘FULL	¬ ‘FULL		‘FULL	¬ ‘FULL		
DATA'[R'] =	‘DATA[‘R]	a	a	‘DATA[‘R]	a	a	‘DATA[‘R]
R' =	‘R	QSIZE-1	QSIZE-1	‘R	‘R – 1	‘R – 1	‘R
FULL' =	‘FULL	false	‘F = QSIZE-2	‘FULL	false	*edge*’	‘FULL

pf_REMOVE = NC(DATA,R) ∧

	(¬ ‘*edge* ∨ ‘FULL) ∧ old ∧		(‘*edge* ∧ ¬‘FULL) ∨¬ old
	(‘F = 0)	(‘F > 0)	
F' =	QSIZE-1	‘F – 1	‘F
FULL' =	false	false	‘FULL

pf_FRONT = NC(R,FULL, DATA, F) ∧

	¬ ‘*edge* ∨ ‘FULL old ∧	(‘*edge* ∧ ¬‘FULL) ∨ ¬ old
return value =	‘DATA[‘F]	

Tabular Descriptions and Specifications

Specification for a search program

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

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

$\wedge$ NC(x, B)

Description of a search program

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

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

$\wedge$ NC(x, B)

Disciplined Design through Documentation

There is no magic easy way to high quality. It requires discipline!

Documentation can make discipline easier.

Documentation can make discipline easier to manage.

We cannot tell people how to think; we can make them document the results of their thoughts in a disciplined, checkable way.

The road to better software stands on a foundation of documents that are **actually used:**

- to record design decisions
- to review design decisions
- to guide future design decisions and revisions

The documents must be (1) written to meet standards, (2) reviewed against standards, and (3) be fully taken into account in future development. *Research is required on how to do this*.

How Documents Support Disciplined Design

Most of us are (all too) familiar with tax forms.

Computing your income tax requires discipline to be sure that you take all factors into account and perform all steps properly.

The return form, helps you to do that.

You are not restricted in the order in which you enter information, but, you must eventually enter it all and use it all.

Software documentation standardised forms can play the same role.

- they tell you what information must be included
- they tell you how to organize it
- they tell you how to check for completeness and consistency.

However, for software, you have to make up your forms as you design. We create blank tables, then fill them in.

Incomplete and Imperfect Documents

When engineers work with physical products they must use imperfect implementations of abstract specifications.

With software, imperfection is not theoretically necessary but it may be convenient and acceptable.

The imperfections must be “bounded” and explicitly limited in their applicability.

For example, we may ignore the limits on representations of numbers because we only work with a limited range of numbers.

It is important to include this in the specification.

No new mathematics, no special notation, is needed for this.

The use of mathematics in engineering does not imply a belief in perfection.

- This is a red herring thrown out by those who do not want to change their ways.

What are the uses of these Documents?

Program function tables are detailed designs for the programs.

They are more easily checked than code.

Table writers make fewer errors than direct coders.

Writing the code once you have the table is usually quick and reliable.
No time is lost.

The tables make the language independent design decisions.

The tables are easy reference material.

The tables also show how code can be simplified.

Test oracles and run-time monitors can be generated.

Other testing functions (e.g. coverage measures) are supported.

These tables support a systematic inspection process.

They take effort but that effort is not wasted because they are used repeatedly in subsequent work.

Checking A Module Design

A module design may be wrong, i.e even if the programs that implement it are satisfy their specifications, the module won't work.

It is good to be able to check this before you invest in coding.

The module design can be checked against the module specification.

We can check before implementation that the design can be made to work.

There is an equation that should be verified. (Commuting Diagram)

We will always make mistakes along the way but we can do better.

Designing and Documenting the "Program Uses Program" Relation

Note that there are many programs in a module.

Definition of uses:

- Given program A with specification S and program B, we say that A uses B if A cannot satisfy S unless B is present and functioning correctly.
- Example:hardware for division use power supply but calls divide by 0 routine

Virtual-machine analogy

This determines what subsets are executable.

Extensibility depends on this (and interface design) as well.

This is often a programmer's casual decision; it should be designed and reviewed before programming.

Not all programs in a module need be on the same level.

This is not a "module uses module" hierarchy.

Systematic Quality Assessment (Inspection, Testing)

Even the best software seems to have bugs and quirks.

Systematic testing (not “try it you’ll like it) and disciplined inspection can find the bugs *before* they are released.

Software is tested against the design documents.

Design documents also used (or produced!) during inspection.

Experience (Darlington Nuclear plant):

- 218 discrepancies found after 6 years of better than average automated testing. (200 were not considered errors.)
- 15 years of active use and change with only one error (analysis) found

The secret of this success is disciplined analysis based on highly structured documentation.

Code Inspection Discipline

The secret of inspection is “divide and conquer”.

Each part must be small enough to be completely inspected.

Each inspection must consider every variable and every case.

There must be a method that insures that each separately inspected part fits with the others and that no part has been overlooked.

This requires discipline and strict methods.

If you don’t do this, you cannot trust the results of your inspection.

If you design and document using displays, inspection is quicker and more effective.

SQRL Code Inspection Process

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

(2) Decompose the program hierarchically.

(3) Produce the descriptions 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 be 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 systematic sequence of small steps.

How can Precise Design Documents Help Testers?

Having documentation ameliorates the following problems.

- Component testing requires precise definition of component boundaries.
- Statistical data is needed for reliability estimation of components.
- Some components may be ready for testing before others.
- Generating test cases is prone to oversight and very time consuming.
- Evaluating test results is very time consuming and prone to oversight.
- Difficult to generate test cases until program is complete.
- Difficult to evaluate thoroughness of test coverage.

Test oracles and monitors can be generated from the documents.

Test cases can be generated (and evaluated) using the documents.

Prototypes or scaffolding can be produced from the documents.

Research Problems

Other Useful Documents?

More efficient and effective evaluation, especially repeated computations.

Improved quantification notation and evaluation.

Systematic use of inverse functions to eliminate quantifiers.

Theorem Provers that can verify that tables are “proper”.

Theorem provers that can check consistency.

Standards, models, and procedures that industry can use

Empirical confirmation or refutation of usability (more)

Improved training and tutorial material.

Better document driven testing.

- (Testing remains the hardest “easy” problem of the developer
- Inspection or Verification does not replace testing.

Summary

Documentation based design can improve the quality of software.

- Documents must be precise representations of relations. (Math is essential)
- Tabular notation leads to more discipline
- Tabular notation leads to more readable design documents.
- Tabular representations of relations are essential in inspection
- Tabular representations of relations are useful for testing.
- The relational theory is sound
- The documents have proven very useful in practice.