Although "formal methods" such as Z, VDM and B have been widely studied and advocated by researchers, their successful practical adoption is relatively rare. Often, they are tried once or twice and then abandoned. Alternative modeling approaches such as UML are widely used but their lack of precise definition has limited their effectiveness. This talk describes a method that, while not as well known in academic circles as Z and VDM, has been repeatedly applied with success in industry. Unlike the "modeling" approaches, this method retains the precision of mathematics and the meaning of the documents is fully defined. There are three keys to the success of this method:

- There is no new mathematics. The approach is based on applying very well known classical mathematical concepts such as set, relation, and predicate.
- We use new notation, tabular expressions, to achieve readability and ease of use that is not possible with mathematical expressions written as a single string of characters.
- Careful attention to defining the meaning and contents of each document achieves “separation of concerns” to reduce duplication and inconsistency.

The approach was motivated by a sequence of practical projects such as avionics software, telephone software and safety-critical software over a 30 year period and refined by academic research. Using the methods described we have been able to achieve practical goals such as precise requirements documentation, automated test case generation, automated test oracle generation, and very effective inspections.
Practical Mathematical Methods

What’s Wrong With Software Development- Old Problems 3
The Role of Documentation in Engineering 10
Why Use Mathematics in Software Development 21
Software Documents: What Views Are Needed? 29
Functional/Relational Model of Documentation 38
Representation of Relations with Tabular Expressions 49
Applications of Functional Documentation 60
Preparing For Change 61
Advantages Over Other “Formal” Approaches 63
Educational Implications 65
What’s Wrong With Software Development - Old Problems

- Programming progress unpredictable, schedules don’t work.
- Developers unclear about module responsibilities (gaps, duplication)
- “Mythical Man Month” effect
- Developer’s work does not integrate easily
- Inspections of code are not effective - many errors not found.
- Testing does not find faults - important test cases overlooked,
- Many bugs in delivered products
- System doesn’t meet the “real” requirements.
- “Ripple effect”
- Poor design decisions not found until code is tested, in use, or changed.
- Maintaining software (correcting errors/updating) very expensive.

Though very old, these problems are still real and costly.

We have to **first** understand the cause, **then** seek a cure.

3/65
What’s Wrong: A Widespread Belief in Magic

- We can satisfy requirements that we can’t write down.
- Undocumented interfaces will be compatible.
- We can prepare for change by ignoring what may change.
- We can review a design that we can’t document.
- We can maintain software that nobody can understand.
- Saying that we do it is enough. We do not have to really do it. [ISO, CMM(I)].

Dilbert “tells it as it is”.

4/65
Pretend Process Improvement (PPI)

Real improvement seems to be too hard so we pretend.

5/65
What’s Wrong: Simple Solutions to Complex, Difficult, Problems

- Learn “team work”.
- Collect anecdotes (scenarios, use cases)
- Write a wish-list. (e.g., “System must be user-friendly.”)
- Just code! Don’t waste time on anything else!
- Draw diagrams with undefined meaning (buzz diagrams)
- Have stand-up meetings.
- Use structured English.
- Use some favorite programming language
- Make simple models that ignore important facts.

All of these things are easy to advocate and easy to do, but they do not solve the real problems.
What’s Wrong: ignoring the Science

There is a simple science of software:

- Computers and variables are finite state machines.
- Software determines the initial state of the user’s machine.
- The effect of programs can be described by relations/functions.
- The laws of program composition are the well-known laws of function composition.
- Relations can be described either by enumeration or by mathematical expressions or by a combination of the two.

Much of the software literature:

- fails to take advantage of this body of knowledge
- treats software but ignores the special characteristics of software
- treats the symptoms (people problems) not the causes.

We have all learned this science but not how to use it.
What’s Wrong: ignoring the People

Many mathematical models of software are very complex.
Languages have built in assumptions about software (special cases) and become research subjects themselves.
Models are more complex than the system.
Expressions are complex, hard to parse.
Completeness and consistency hard to check.
Incidental details as prominent as the essential facts.
Impose additional tasks that do not further the main task
Logicians like axiomatic methods, but many others do not.
Great for thesis production but not software production.
What’s Wrong: Assuming that People Cannot be Educated

- “Nobody does it that way.” (1972)
- “Nobody will do that!” (2007 reviewer, editor)
- “I have no idea how to do that.” (very good developer)
- “Does Microsoft do that?” (a self-styled innovator)

Can we expect change without changing people?

Parallels to EE history (e.g. transmission lines).

We must learn to distinguish technicians from Engineers.

Engineers become Engineers through education and guided experience.

Software Developers “learn to swim by jumping in the water and either sinking or surviving”. We give them the equivalent of a bathing suit.
The Role of Documentation in Engineering

Engineers design through documentation.

They begin, not by building, but by writing or drawing.

Each step of a design is marked by the completion and approval of a document.

Designers are expected to be cognizant of, and to respect the statements in, a document.

Documents are used to settle disputes between developers and designers.

Inspectors check subsequent developers against what is said in approved documents.

No Professional Engineers work without documents.
What is a Document?

A binding record of decisions, one that restricts future decisions until it is officially revised and the revisions are accepted by all parties.

An implicit description of excluded alternatives

• If no alternative has been excluded, no decisions have been made. (check your “decisions”).

To be as useful as possible these documents must be:

• Accurate
• Consistent
• Complete (all restrictions described).
• In a form suitable for use as a reference by implementers during future development.
Documents as Predicates

A predicate is a function that maps a set of objects onto \textit{true} and \textit{false}.

Comparing a document with the object that it purportedly describes, you can tell if the document is \textit{true} or \textit{false}.

In this way, a document is a predicate on a set of objects. It partitions that set of objects into three sets:

- Those for which it is undefined because it mentions attributes not present (not in the predicate’s domain)
- Those for which it is \textit{true}, i.e. everything that the document says is an accurate description of the object.
- Those for which the document says something that is \textit{false}
Document Roles: Descriptions vs. Specification

Engineering usage:

• A description states facts about products; it may include both incidental and required properties.

• A specification is a description that states only required properties of products.

• A full specification is a specification that states all required properties.

The same notation may be used for all 3.

• This has confused many researchers.

• These classifications are a matter of intent not notation.

• There is no such thing as a “specification language”.

You must label each document indicating its intended use.
What I Do NOT Mean by “Document”

Introductions, “big picture” descriptions
FAQ (Frequently Asked Question) answers
Anecdotes
Sketches
Discussion papers
Lies
  • An abstraction is something that represents many objects equally well.
  • Lies represent no objects (e.g. infinite capacity stack)
Why do Engineers Need Documents

Professional Responsibility to make product “fit for use”

- Need for review by those who know what is needed
- Need for approval by those with authority
- Need to guide/constrain those who continue the work
- Need to help inspectors during discussion
- Need to inform those who maintain the product.

None of these things can be done without documents.

Which of these is **not** true for software?
What do Engineering Documents Look Like

Diagrams with precisely defined meanings (restricted form)
A minimum of 3 “views” for physical objects
More “views” for more complex objects.
Detailed descriptions or specifications of components
Equations defining behavioral characteristics
Numerical values for equation parameters

Note:

- Engineering documents are designed for use.
- Engineering documents are taken seriously.
- Ignore one and you are responsible for problems.
How Do Software Developers See Documentation

An annoyance forced on them by management.
An unreliable information source.
Something to be written by non-programmers
Something written after code is working.
Design documents not useful during maintenance.
An opportunity for creative writing.
Software Developers Do Not Treat Documents With Respect

CS Profs do not teach respect for documentation.
CS Profs do not teach how to document.
Older practitioners do not treat documents with respect.
There are few good examples around.
Management does not require good documentation.
Time pressures force programmers to neglect documents.
  • Documents seem to take time.
  • Timesaving benefits of good documents not experienced
Most do not believe that good documentation is possible.
Documents vs. Models

Two Questions:

- Are models documents? (not usually)
- Are documents models? (yes, good ones!)

What is a model?

- A simplified version of “the real thing” that is easier to study.
- Models have some properties of the “real thing”
- Not all properties of the models are properties of the “real thing”.

Models are not usually descriptions.

- Everything you can learn from a description must be true of reality.

What you can learn from a model of a Boeing 747?

- That it has a bump, two big wings, two small wings ...
- That it is plastic, has painted “windows”, can be held in your hand, …

Documents are models that you can trust.
 Views in Documents

A random collection of true statements is nearly useless.

- The information must be organized.
- The organization should be a near partition (minimal overlap, avoid inconsistency).
- There should be partitioning rules, simple rules to tell writers where to put information, and readers where to find it.
- Time of discovery or writing should not be the criteria used; you should not have to know when a decision was made in order to know where to find it.
- Revisions to keep the documents factual but must not change the document’s view.
- For mechanical objects, the standard views are top, front, and side. There are additional views needed for complex objects.

We need to find standard views for software systems.
Why Use Mathematics in Software Development

Mathematics has two fundamental advantages:

- Good mathematics has fully defined semantics - no ambiguity
- Mechanical transformation (reasoning, deduction) rules.

No other medium (e.g. (undefined) pictures, language) provides these advantages.

Why should we forgo such a tool?

- We have to learn how to use the tools available to us.
- We may not need to (re) invent tools.
When Should Developers Use of Mathematics During Design

Recording design decisions precisely
Reviewing design decisions
Checking completeness and consistency of designs
Giving reliable, precise, guidance to programmers
  • speed up programming
  • better quality programming

Programs are mathematical objects and it is only through mathematics that we can do these things properly.

When you try to describe a mathematical theorem or other mathematics in words, it gets fuzzy and unclear.
Use of Mathematics During Quality Assurance

Quality Assurance is a review activity that should be carried out both during and after development.

It involves:

- Inspection (systematic “divide and conquer” examination)
- Testing (use in carefully chosen cases)
- Verification (mathematical checks)

Each of these activities can be made more effective by using mathematics.
Mathematical Assistance in Inspection

The secret of inspection is “Divide and Conquer”

- Divide the system into parts that can be studied independently in such a way that if every part is correct, the whole system is correct.
- To conquer, check each of those parts.

With the aid of mathematical documentation, we can achieve highly effective inspections, find many easily overlooked errors, and engender confidence in a design.

The mathematics is used to document interfaces and to partition the state space.

This has been applied on safety critical software for nuclear power plants. **No errors discovered for > 15 years of use.**
Mathematical Assistance in Testing

There are two difficult problems in testing:

- Selecting the cases to check
- Determining if the computed results are correct

Mathematics can help in both of these:

- Statistics can be used to determine expected reliability.
- Algebra and logic used to find “difficult” test cases.
- Logic used to assure full case coverage.
- Mathematical documents can be converted to test oracles or real-time monitors.
Mathematics is Essential for Verification

During inspections, critical cases (or all cases if you wish) can be proven correct. The main problems are:

- Many programs seem to work but are not correct. (proof impossible)
- It can be difficult to formulate the theorem that must be proven.
- The correctness of some programs depends on subtle theorems.
- Proof requires mechanical and boring work.

Mathematical documentation helps by:

- stating the theorem that must be proven,
- providing lemmas in the form of documentation about other programs,
- If the expressions are well-structured (tabular expressions), the job can be divided into smaller easier proofs.
Use of Mathematics During Maintenance

Maintainers have to revise programs already in use to:

- correct errors
- improve performance
- add new features
- accommodate changes in the environment or equipment.

Maintainers require precise documentation to:

- describe the “design” so changes can be consistent with it
- provide detailed descriptions of programs to avoid inserting “bugs”.
- quickly find the information that they need
- assist them in testing and inspecting the revised programs

Only mathematical documents can provide the information that maintainers need.
Use of Mathematics in Documents

The key to gaining the advantages of mathematics when developing software is to use mathematics in documents.

The remainder of this talk is devoted to showing how that can be done for software. The key problems are:

- Determining the contents of documents
- Organizing the documents as reference documents so that the information is easy to find.
- Structuring the mathematical expressions so that they are easy to use.
- Assuring consistency by avoiding overlap
- Checking for completeness
Software Documents: What Views Are Needed?

- System viewed as a Black Box
- Internal system structure
- Software black box
- Software component/module: Black Box
- Uses Relation
- Software module: internal design
- Terminating program: Black Box
- Terminating program: Hierarchical set of displays
System Viewed as a Black Box

The system senses values of monitored variables, and determines values of controlled variables.

View describes all monitored and controlled variables.

View gives the value of each controlled variable at a moment in time as a function of the history of the values of monitored and controlled variables.

Nothing else is in this view.
System Structure View

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

For each device

• Describe all monitored, input, output, and controlled variables
• View gives the value of each output variable at a moment in time as a function of the history of the values of input and output variables.

For the software

• Describe all input variables and all output variables.
• View gives the value of each output variable at a moment in time as a function of the history of the values of input and output variables.

Describe the connection via the variables.

Monitored variables and controlled variables must match the system black box view. Use shared list.
Alternative System Structure View

Describe virtual I/O devices and remaining software

Virtual devices are a combination of hardware and software that convert the values of the monitored variables to input variable values that are (approximately) the same. This approach has the following advantages:

- All interfaces and specifications are simpler.
- Hardware characteristics are better hidden.
- Requirements for software resemble those for complete hardware/software system.

David Parnas
2 April 2009 16:45:23
IPA SEC slides.
A component or module is a set of programs:

- *Module* is a set of programs that should be designed together.
- *Component* is a set of programs that will be distributed together (as a unit)

For each such software unit:

- Describe all input and output variables.
- View gives the value of each output variable at a moment in time as a function of the history of the values of input and output variables.

No other information in this view.
Uses Relation Between Programs

For each program in the black box view of a module/component, state which other programs it uses.

A program, A, *uses* another program, B, if A cannot satisfy its specification unless B is present and functioning properly.

This can be represented by a graph or table.

The relation should define a hierarchy if we want functional subsets.

This is a relation between programs, not modules.

Programs from one module may be on different levels of the uses hierarchy.
Software Module: internal View

The most important aspects of a module’s internal design are:

• The complete data structure
• The function performed by each of the externally visible programs
• The way the data structure is to be interpreted.

H., D. Mills, N. Wirth and C.A.R. Hoare have said this.
Terminating Program: Black Box

A terminating program starting in a state $s_1$ will terminate in a state $s_2$. In the black box view of such programs we need to describe the set of starting states, the set of stopping states, and the mapping between them.

Nothing else is shown in this view. We mention no intermediate states. We do not describe the number of steps in going from the starting state to the stopping state.
Terminating Program: Hierarchical Set of Displays

Long programs must be viewed as a set of displays

Every display has 3 parts.

- A specification of a program
- The program text (short)
- Specifications for all invoked programs

Invoked programs may be either

- primitive: we don’t look inside it
- presented as a set of displays

This view allows us to focus on small amounts of code and fully understand each program but cover them all.

“Divide and Conquer”: the key to understanding programs.
Functional/Relational Model of Documentation

Where is the mathematics?

- We discussed the nature of Engineering Documentation.
- We discussed the concept of a view.
- We discussed standard views for software documents.
- We discussed the role of documents.
- So far, very little mathematics.

We use mathematics to define the contents of each view.
We use mathematics to describe and specify documents.
Next we look at how we define contents (not labels).
Basic Mathematical Concepts

Set

- \{2,r,b\}

Relation

- Set of ordered pairs, \(x>y\), \(x<y\)

Function

- Set of ordered pairs \((x, y)\) in which each \(x\) appears at most once.

Expressions vs. Function and Relation

- \(f(y) = y + y\) and \(f(y) = 2 \times y\) Two expressions same function

Predicate:

- function with values \textit{true} and \textit{false}

Characteristic Predicate of a Set

- Predicate that is \textit{true} for all members of the set but \textit{false} otherwise.
Content definitions for documents (general concept)

Endless arguments about where information should be.
Information is hard to find; time is wasted.
The solution is precise definitions of content (not form).
Every document presents one view of a component.
The contents of the document will be the characteristic predicate of one or more relations.
The next few slides will describe those relations for each of the standard views of software.
System Black Box View

Represent the following relations:

Relation NAT

- Domain contains values of $m_t$, (values as a function of time)
- Range contains values of $c_t$,
- $(m_t, c_t)$ is in NAT if and only if nature permits that behaviour.

Relation REQ

- domain contains values of $m_t$,
- range contains values of $c_t$,
- $(m_t, c_t)$ is in REQ if and only if system should permit that behaviour.

These must describe physical reality and they must be consistent.
System Structure View

Describe the connection via the variables.

In the following variables represent time functions

Describe the following relations

- **IN = \{m, i\}** such that the input devices will produce i if m describes the history of monitored variable values.

- **OUT = \{o, c\}** such that the output devices will produce c if o describes the history of output variable values.

- **SOF = \{i, o\}** such that the software will produce o if i describes the history of output variable values.
Alternative System Structure View

Describe **virtual** I/O devices and remaining software

\[
\text{M-variables} \rightarrow \text{I/O device} \rightarrow \text{inputs} \rightarrow \text{software} \rightarrow \text{outputs} \rightarrow \text{I/O device} \rightarrow \text{C-variables}
\]

Describe the connection via the variables.

Describe the following relations

- \( \text{IN} = \{m, i\} \) such that the virtual input devices will produce \( i \) if \( m \) describes the history of monitored variable values.

- \( \text{OUT} = \{o, c\} \) such that the output devices will produce \( c \) if \( o \) describes the history of output variable values.

- \( \text{SOF} = \{i, o\} \) such that the software will produce \( o \) if \( i \) describes the history of output variable values.
Software Component/module: Black Box

For each such software unit

- Describe all input and output variables.

Describe the following relations

- \( \text{SOF} = \{i, o\} \) such that the software will produce \( o \) if \( i \) describes the history of output variable values.

No other information in this view.
Uses Relation Between Programs

For each program in the black box view of a module/component, state which other programs it uses.

This is a binary relation.

• With a small number of programs we can list the ordered pairs.
• We can also draw a graph with a node for each program and an arrow between two nodes if a uses relation applies
• We can give the characteristic predicate of the relation.
• We can use a table with a row and column for each program and a “1” in a cell to indicate if the row program uses the column program.
Software Module: internal View

The complete data structure can be described using the data declaration features of the programming language.

I will discuss program functions on the next slide.

The interpretation of the data structure is defined by an abstraction relation.

- This relation maps between states of the data structure and the input/output histories that will lead to those states.
- A pair (state, history) is in the relation if that state can be reached after that history.
Terminating Program: Black Box

A pair \( (s_1, s_2) \) is in the relation describing a program if when the program is started in \( s_1 \) it will terminate in \( s_2 \).

If there is no pair with \( s \) on the left, the program will not terminate if started in state \( s \).

If there is more than one pair with \( s \) on the left, the program is called non-deterministic.

If the program is non-deterministic, we need an additional set: the set of states in which termination is guaranteed.

These relations can be represented by their characteristic expression, by enumeration, or by a graph.

These are called program-function or program-relation.
**Terminating Program: Hierarchical Set of Displays**

The specifications in part 1 (specification of the program in the display and part 3 (specification of programs invoked in the display) are the program relation for those programs.

Part 2 is program language text. These should be simple short programs that are kept short by invoking other programs.
Representation of Relations with Tabular Expressions

Mathematics is used every day by traditional Engineers.

Software Engineers don’t use mathematics the same way.

Conventional expressions become very complex when describing functions with discontinuities.

Mathematical Expressions can be written in a tabular format that makes them easier to write and read.

The following slides contain examples that illustrate this way of writing mathematical expressions.
A Semi-formal Mathematical Expression

\[ f(x, y) = \begin{cases} 
  x^2 - y^2 & \text{if } (((y < 0) \land (x < 0)) \lor ((y < 0) \land (x > 0)) \lor ((x = 0) \land (y = 0))) \\
  x + y & \text{if } (((y = 0) \land (x < 0)) \lor ((y = 0) \land (x > 0)) \lor ((y > 0) \land (x = 0))) \\
  x^2 + y^2 & \text{if } (((y < 0) \land (x = 0)) \lor ((x < 0) \land (y > 0)) \lor ((x > 0) \land (y > 0)))
\]
Inverted Tables

Grids T[0] and T[1] contain predicates
The value expressions are in T[2].

\[\begin{array}{|c|c|c|}
\hline
x + y & x - y & x \times y \\
\hline
\end{array}\]

\(y < 0\)
\(y = 0\)
\(y > 0\)

\(T[1]\)

\(T[2]\)

\(T[0]\)
# Generalized Decision Table

<table>
<thead>
<tr>
<th></th>
<th>x + 1</th>
<th>y + 2</th>
<th>x + y</th>
<th>x - y</th>
<th>y - x</th>
</tr>
</thead>
<tbody>
<tr>
<td>T[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>V₁&gt;1</th>
<th>true</th>
<th>true</th>
<th>¬ (V₁&gt;1)</th>
<th>¬ (V₁&gt;1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T[0]</td>
<td>¬ (V₂&gt;5)</td>
<td>V₂&gt;5</td>
<td>V₂&gt;5</td>
<td>¬ (V₂&gt;5)</td>
<td>¬ (V₂&gt;5)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>true</th>
<th>V₃+3&gt;5</th>
<th>¬ (V₃+3&gt;5)</th>
<th>V₃+3&gt;5</th>
<th>¬ (V₃+3&gt;5)</th>
</tr>
</thead>
</table>

Find a column in T[0] with all entries **true**

Evaluate the T[1] entry from that column

Decision tables have been used for years, these are better!

52/65
The four functions defined by Table 5 are:

- $S(2,1)$ maps from (height in inches, weight in lb.) to a weight class.
- $S(2,3)$ maps from (height in inches, weight in kg.) to a weight class.
- $S(4,1)$ maps from (height in meters, weight in lb.) to a weight class.
- $S(4,3)$ maps from (height in meters, weight in kg.) to a weight class.

We could also define 4 functions that map to the colour code and omit $T[5]$ and $T[6]$.

Below we define the meaning of this type of tabular expression.
Case Study: Dell Keyboard Checking Tool

Program in daily use in Limerick for many years.
Believed to be completely correct.

Two documents totaling 21 pages (English).

- ambiguities
- missing cases
- errors

Posed as a challenge by skeptical manager

All information could be expressed in one page

- revealed errors in program and documents
- much more precise and easily used.
- served as input to testing and inspection.
This Mathematical Expression Documents the Behaviour

\[
\begin{align*}
(N(T)=2 \land keyOK \land (\neg(T=_\_ \land N(p(T))=1)) \lor (N(T)=1 \land (T=_\_ \lor (\neg(T=_\_ \land N(p(T))=1))) \land \\
(\neg(keyOK \land \neg(prevkeyOK \land \neg(prevkeyesc))) \lor ((\neg(T=_\_) \land N(p(T))=1)) \land \\
((\neg(keyOK \land keyesc \land \neg(prevkeyesc)) \lor (\neg(keyOK \land keyesc \land prevkeyesc \land \\
prevexpkeyesc)) \lor ((N(T)=N(p(T))+1) \land (\neg(T=_\_) \land (1<N(p(T)<L)) \land (keyOK)) \lor \\
((N(T)=N(p(T))-1)) \land (\neg(keyOK \land \neg(keyesc \land (\neg(prevkeyOK \land prevkeyesc \land \\
prevexpkeyOK) \lor prevkeyOK) \land ((\neg(T=_\_) \land (1<N(p(T)<L)) \lor (\neg(T=_\_) \land N(p(T)=L))) \\
\lor \\
((N(T)=N(p(T))) \land (\neg(T=_\_) \land (1<N(p(T)<L)) \land (\neg(keyOK \land \neg(keyesc \land (\neg(prevkeyOK \land prevkeyesc \land prevkeyesc \land \\
prevexpkeyesc)) \lor (\neg(keyOK \land keyesc \land prevkeyesc \land prevkeyesc \land \\
prevexpkeyesc)) \lor (\neg(keyOK \land keyesc \land prevkeyesc \land prevkeyesc \land \\
\neg(prevexpkeyesc)) \land (1<N(p(T)<L)) \lor (N(P(T)=Fail) \land (\neg(keyOK \land keyesc \land prevkeyesc \land \\
\neg(prevexpkeyesc)) \land (1<N(p(T)<L)) \lor (N(P(T)=Pass) \land (\neg(T=_\_) \land N(p(T))=L) \land \\
(keyOK))
\end{align*}
\]

Theoretically right. Practically Useless!
# Keyboard Checker: Tabular Expression

\[ N(T) = \]

| \( T = \_ \) | \( \neg (T = \_ \) \land | \n | \( N(p(T)) = 1 \) | \( 1 < N(p(T)) < L \) | \( N(p(T)) = L \) |
|---|---|---|---|---|
| \| 2 | \( N(p(T)) + 1 \) | Pass |
| \( \neg \text{keyesc} \land \neg \text{keyOK} \land \) | \( N(p(T)) - 1 \) | \( N(p(T)) - 1 \) |
| \( \neg \text{prevkeyOK} \land \neg \text{prevkeyesc} \land \) | \( N(p(T)) \) | \( N(p(T)) \) |
| \( \neg \text{prevkeyOK} \land \neg \text{prevkeyesc} \land \neg \text{prevkeyOK} \land \) | 1 | 1 | \( N(p(T)) \) | \( N(p(T)) \) |
| \( \neg \text{prevkeyesc} \land \neg \text{prevkeyesc} \land \) | 1 | \( N(p(T)) \) | \( N(p(T)) \) |
| \( \neg \text{prevkeyesc} \land \neg \text{prevkeyesc} \land \neg \text{prevkeyesc} \land \) | Fail | Fail | Fail |
| \( \text{prevkeyesc} \land \neg \text{prevkeyesc} \land \) | 1 | \( N(p(T)) \) | \( N(p(T)) \) |
| \( \text{prevkeyesc} \land \text{prevkeyesc} \land \text{prevkeyesc} \land \) | Fail | Fail | Fail |
| \( \text{prevkeyesc} \land \text{prevkeyesc} \land \text{prevkeyesc} \land \) | 1 | \( N(p(T)) \) | \( N(p(T)) \) |
TABLE X
name Auxilary Function

<table>
<thead>
<tr>
<th>name</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>rrcConnRequest</td>
<td>CapacityReq</td>
</tr>
<tr>
<td>capacityCfm</td>
<td></td>
</tr>
<tr>
<td>registerServingUeCtxtReq</td>
<td>admissionCfm</td>
</tr>
<tr>
<td>cellParamsRC2Cfm</td>
<td>admissionCfm</td>
</tr>
<tr>
<td>cellParamsReq</td>
<td>cellParamsCfm</td>
</tr>
<tr>
<td>adjacentIntraFreqCellsRsp</td>
<td>allocateDlChCodeNmCfm</td>
</tr>
<tr>
<td>initialResourceCfm</td>
<td>reserveAal2CepCfm</td>
</tr>
<tr>
<td>nibapRISetupRespInd</td>
<td>ConnCfm</td>
</tr>
<tr>
<td>spConfigCfm</td>
<td>rrcMsgUlInd</td>
</tr>
<tr>
<td>rrcMsgDlCfm</td>
<td>fddIfhoSupp</td>
</tr>
<tr>
<td>rrcMsgDlCfm</td>
<td>failSpRelease</td>
</tr>
<tr>
<td>failRadioLinkRelease</td>
<td>celloDisconnect</td>
</tr>
<tr>
<td>releaseSpResources</td>
<td>DlChCodeRelease</td>
</tr>
<tr>
<td>DecreaseLoad</td>
<td>ueUnRegistration</td>
</tr>
<tr>
<td>failReject</td>
<td></td>
</tr>
</tbody>
</table>

TABLE XI
name Auxilary Function

<table>
<thead>
<tr>
<th>name</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Auxiliary functions - to make functions more readable.
Not beautiful, but this is the information maintainers needed.
No Theoretical Advantage

Just practical advantages:

- fewer errors
- checkability
- fewer oversights and reduced contradiction
- ease of reference

This is the way to “capture” design information.
Applications of Functional Documentation

• Communication with users - “What’s really needed?”
• Design medium for architects.
• Reviewable design documents.
• Guidelines for implementors.
• Information for testers (V, W models)
  ★ test oracle and monitor generation
  ★ test case generation.
• Information for code inspection. (displays)
• Information for maintainers.
Preparing For Change

Don’t confuse agile software with agile processes!

• Agile software is easy to change when you need a change.
• Agile processes allow programmers to make quick decisions without review, change decisions, revise code.
• It is agile software that we need;
• We won’t get that from an agile process!

We cannot design software where everything is equally easy to change.

We must design software for specific types of change.

This requires analysis of what will change and interfaces that hide what will change.
Don’t use Change as an Excuse

We are often told not to document designs because it is all going to change anyway.

- This is a self-fulfilling prophecy.
- Code built without careful (reviewed) design will change.

The solution is

- Identify what is “common” and what is “variable”
- Document the common factors; base interfaces on them.
- Hide the variabilities behind precise interfaces.

This is “Design for Change/Agility”

This is “separation of concerns” (Dijkstra)

This is abstraction, information hiding, even O-O.
Advantages Over Other “Formal” Approaches

Information that you need, when you need it, where you put it, and where you will find it.

No inconsistency with basic checks (variable matching)

Simple basic (old) mathematics.

Meaning fully defined.

Provides documents, not just models.

Notation designed for complex expressions that will be used as a reference document for practitioners.

Expressions for evaluation, not axioms for derivation.
Why it Works in Practice

Purpose driven - motivated by practical applications.

- Every idea came out of a project, was then formalized
- Designed as a response to perceived needs and real cause

Content definitions

- These are not what you notice first (that's tables)
- These are what makes it really work.

Precise semantics

- Tables defined by equivalent classical expressions
- Connection to real world (programs) always clear.

Practical notation

- You don’t have to read it all to find what you want.
Educational Implications

This is engineering education not science education
Students must learn how to use science and mathematics when designing, not how to extend our knowledge.
Basic Engineering is relevant for Software Engineers too
Mathematics, as in classical Engineering, plays central role.
Discrete education needed, but must be applied. Expressions not axioms.
Projects are essential and it is essential that the role of documentation be stressed. Design projects so that success requires documentation.
Documents from day 1, not after bad habits are learned.