

**Requirements Documentation:
Why a Formal Basis is Essential**

David Lorge Parnas, P.Eng.

NSERC/Bell Industrial Research Chair in Software Engineering
Director of the Software Engineering Programme
Faculty of Engineering
McMaster University, Hamilton ON Canada L8S 4K1

Abstract

Unless you have a complete and precise description of your product's requirements, it is very unlikely that you will satisfy those requirements. A requirements document that is incomplete or inconsistent can do a great deal of damage by misleading the developers.

A collection of statements in English or some other natural language cannot be checked for completeness and will not be precise. Even if you translate a requirements statement into a mathematical language, you can only show that the result of the translation is complete and unambiguous; the original may still be faulty.

This talk will describe a simple and sound procedure for documenting requirements - one that lets you know when your document is complete and consistent (but not necessarily correct). Documents produced by following this procedure can be reviewed by potential users and specialists and can serve as the input to tools that generate prototypes and monitors.

**Confession:
I am not a "Requirements Engineer"**

I have spent many hours working on requirements documents and developing methods.

"Engineer" is a qualification. It denotes people who are licensed to practice the particular engineering profession. There are specified educational and experience requirements.

Nobody can declare them self to be an "<x>-Engineer; if they could the term would be meaningless.

I am an Engineer in Ontario.

A few of the licensing authorities have started to put meaning into the term "Software Engineer" but the term "Requirements Engineer" is meaningless.

The encouragement of ever more narrow branches of Engineering is not a positive contribution.

"Engineering" - Nearly Meaningless as a Verb

"Engineering" (verb)

- Doing something that Engineers do, but Engineers do lots of things

Requirements Engineering (verb phrase)

- Elicitation (getting the information)
- Documentation (writing it in an organised manner)
- Analysis (checking for properties like completeness and consistency)
- Verification (making sure the document is right)
- Testing - seeing if you said what you meant?
-
-
-

Being more specific will help you to do a better job.

The Responsibilities of an Engineer

An Engineer is a professional whose responsibility includes making sure that products are "*fit for use*".

This requires careful communication with those who know the application environment.

The careful communication must be on the basis of a *written* statement of the requirements

- that can be read and analysed *by both sides*
- that can be used to *evaluate* the product after completion
- that can be used to *settle disputes*.

A Small Horror Story that is Close to my Heart.

Product: A motion sensitive pacemaker
 Situation: Rarely needed, only if heart rate drops to low for activity level.
 Software: Measures motion, estimates activity level, computes expected heart rate, intervenes (causing some discomfort) if rate is lower than expected.

- Parameters include resting heart rate and slope of curve.
- Resting heart rate can only be set in multiples of 10.

This does not meet patient’s requirements.
 Software requirements never documented or reviewed by users.
 Those with a heart rate between the possible set points will have to chose between too low and too high.
 Doctor has no idea what the parameters mean and does not set them anyway. No instructions are given.

A Small Success Story

PROSYS: A small Software Company in Germany
 Developer and customer sit down and complete

- A list of quantities of interest
- A set of tables

They check for completeness and consistency.
 They initial the tables.
 The tables are used to develop the software.
 60% of the code can be generated by simple tools.
 The software is almost right.
 There is no doubt about who pays for each “fix”.

The Secret: The tables constituted a precise requirements document.

Building on Rock vs. Building on Sand

My basic assumptions:

- Software builders need tools.
- Software tools must be based on practical experience - reflecting what developers need to do.
- Software that is “almost right” is wrong.
- Notation that is not precisely defined leads to software that is (at best) almost right.
- Notations that are used in tools are precisely defined, but that definition must be clear and simple for both tool users and tool implementors.
- Mathematical definitions of software notations are essential, but must be simple and based on standard mathematics.
- Little new mathematics is needed. No new fundamental mathematics is needed.
- Documentation is a serious *unsolved* problem for software developers. It is essential for design, review, implementation, testing and maintenance.

What do we want to do with mathematical descriptions of software?

- (1) Describing software products that we already have - so that people can use them without reading the code.
- (2) Writing specifications for software products we do not yet have - so that the programmers and clients can agree on the requirements.
- (3) Be able to verify that a product meets its requirements (testing or proving).

What are the acceptance criteria?

- (1) Descriptions must be easier to understand than the code.
- (2) We must state the requirements in a way that does not restrict the solutions unnecessarily.
- (3) Testing and proof can eventually be automated.

Are Programs Different From Other Engineering Products?

Before we had computers, engineers used classical mathematics to describe and analyse their products.

In Computer Science, most researchers have turned to newly invented “languages”.

We are using software to replace conventional products.

Why can't we simply go on using the mathematics we used to use?

- Wrong Answer: Conventional products are inanimate objects.
- Wrong Answer: We need to describe the procedure followed by the program.
- Right Answer: The functions have many more points of discontinuity. We will return to this point later.

Describing Engineering Products

Engineers use mathematics, not just words, to describe their products.

- They use a variety of descriptions rather than attempt one “complete” description.
- There is never a complete description of a product.
- Each product description is intended for a different purpose and each is an accurate description of some aspect of the product.
- Even taken together, these descriptions never constitute a complete description. There are always some facts that are not stated in the descriptions.

Even for simple physical objects, Engineers produce several drawings (“views”) and require additional numerical specification sheets

Current specification “languages” make no provision for this. They use the same approach for all “views”!

Types of Software Products

- (1) Systems interacting with the real-world,
- (2) Conventional terminating programs,
- (3) Programs that shouldn't terminate,
- (4) Modules (hiding data structures, etc.)
- (5) Objects (created by modules)

For systems, can we follow control theory and deal with relations between time functions.

For terminating programs, we can describe the program's effect on the exposed data structure

For modules, we must not mention the data structure. Consequently, we must describe classes of traces (sequences of visible events)

For objects, we extend the module method to allow multi-object operations.

For non-terminating programs, we can replace time-functions, with event sequences.

The Audience of a Requirements Document

We are talking about a technical document written for a group of developers.

It is not a sales pitch.

It is not an introduction to the user or the application area.

It should serve as a reference document during the development.

It should take decisions away from the programmers that should be made by others.

The document will also be used for design reviews.

It should be used for test case generation and test result evaluation.

It should be used, and kept up to date, during maintenance.

The Goals of the Requirements Phase

- A. Decide what to build before starting to build it:
- Make the “what decisions” explicitly before design, not implicitly during design.
 - Make sure you build what is needed.
 - Allow the users to comment before the product is built.
- B. Provide an organised reference document for the software engineer:
- Provide accurate, consistent information.
 - Answer constraint questions only once.
 - Relieve her of any need to decide what is best for the user.
 - Compensate for ignorance of the application environment.
 - Give him the information needed to make his design decisions.
 - Allow her to make accurate estimates of time and resources needed.

Allow for personnel turnover.

The Goals Of The Requirements Phase

- C. Provide a reference document for a Quality Assurance Group
- Test design should not depend on program.
 - Authority required:---Q.A. and programmer may disagree.
- D. Specify all constraints on the implementation
- Know what you're up against.
 - Have some “protection” against customer changes.
 - Be able to judge feasibility and cost.

Specify constraints for future performance.

Writing Down Requirements

The most costly errors are those made early in the process - they are the hardest to change.

Misunderstandings about requirements lead to early mistakes.

Programmers need to be told what is needed.

They must also be told what is subject to change.

Requirements must be subject to review.

Safety reviews of software must be based on a previously agreed statement of requirements.

Maintenance actions must be based on requirements.

None of these things is possible unless we have a *written* statement to work with.

That *written* statement must be precise and complete.

What's Wrong with Requirements Methods?

Many researchers and developers think of requirements as a set of elements, each element being one requirement.

Consider three requirements.

- The output must be an integer.
- The output must be positive.
- The output must not be zero.

Consider an alternative formulation:

- The output must be a natural number

These are equivalent - one requirement or three?

We cannot count requirements or list them

If we try, we have no hope of checking for completeness, consistency, correctness.

There is a better way, based on the basic model used in control system analysis.

The Two-variable Model

This is the “traditional model” of a hardware/software system.

- 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.

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

Review by Subject Matter Experts is hard to get.

The Four Variable Model

Outside the system we have physical variables, some monitored, some controlled, some both.

Some devices sense monitored variables and determine computer inputs.

Others devices read computer outputs and control the controlled variables.

Some variables can be both monitored and controlled.

Otherwise, the sets are disjoint.

The System Requirements specify the desired relation between the monitored and the controlled variables.

Subject-matter experts can understand and review requirements in those terms.

The ideal system is a state machine whose outputs are piece-wise continuous functions of the history of its environment.

- All digital systems are approximations of that ideal.
- We specify the ideal, then specify the tolerances.

How to document system requirements?

The first step is to:

Identify monitored variables (m_1, m_2, \dots, m_n).

Identify controlled variables (c_1, c_2, \dots, 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

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

- what the operator sees
- what appears on a bill
- control positions

This is only the beginning, but for many projects you cannot even find a complete list of these variables and there is no agreement on what they are.

How can we document system requirements?

Answer: Describe the following relations¹:

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 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?

It must be true that,

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

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

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

Checking these properties shows completeness and feasibility, not correctness!

Must Mathematics be Unreadable

Complex formulae are unreadable. Engineers avoid them.

Axiomatic definitions are often too subtle. The implications are often not understood.

Equations can be hard to solve and indirect

But, ...

- Using two dimensional (tabular) formulae one can parse a complex formula into simple ones.
- Using "function" concepts one can provide definitions that can be evaluated to yield values.
- We don't have to specify things indirectly.
- Tabular expressions can be "closed form expressions" so that you find the value of interest just by carrying out operations to evaluate an expression.

Incomplete Knowledge and Mathematics

"I don't know" can be said mathematically. We describe partial functions and relations.

"I don't care" can also be said mathematically. We allow the range (co-domain) of the relation to be the universe (set of physically possible values)

In practice, it is important to distinguish:

- I don't know, but must find out
- I don't know, and the case will not arise
- I don't care.

In the first case, we leave entries blank or use "?" in those portions of the tables corresponding to information yet to be retrieved.

In the second case, there may be no space in the table for the missing information.

In the third case "*true*" is used to denote the fact that any value will satisfy our requirements.

Design can begin with incomplete knowledge, if we know what it is that we don't know.

A SIMPLE EXAMPLE

REQUIREMENTS FOR DONUT DISPLAY

Notation

|x| means x is an input variable.

||x|| means x is an output variable.

&x& means x is a monitored environmental variable.

#x# means x is a controlled environmental variable.

[i] means i is a subscript for an array.

<i> means i is a subscript for a bitstring.

'x means the value of x before some event.

x' means the value of x after some event.

@F(x) means the event of x becoming false.

Bitstrings are used as booleans, 1 = true.

Note: binary is a function for converting integers to bitstrings by interpreting the bitstrings as binary numbers.

SIMPLE EXAMPLE (cont.)

Monitored Variables:

integer array &STOCK& [0:31].

- Each integer represents the number of donuts in the display case.
- They are initially 0. Range of values [0 ... 127].

Controlled Variables:

integer array #DISPLAY# [0:31].

- These are values displayed in the bakery, one for each donut type. Range of values [0 ... 127].

RELATION NAT:

&STOCK&[i], #DISPLAY#[i] between 0 and 31

RELATION REQ:

#DISPLAY# = &STOCK&

Note: These are the ideal requirements. We must still specify tolerances in terms of delays, etc.

SIMPLE EXAMPLE (cont.)

Input Variables:

bitstring |IN| <1:8>

Output Variables:

bitstring array ||OUT||[0:31]<1:8>

Relation IN:

Let t represent real time.

Let t0 be the most recent time at which
@F(&STOCK&' = '&STOCK&)

Let i be such that at t0
(&STOCK&'[i] ≠ '&STOCK&[i])

|IN|<1> = (t-t0) < 50ms.

|IN|<2:6> = (|IN|<1>->binary(i) else 00000)

|IN|<7>=

(|IN|<1>=>'&STOCK&[i] < &STOCK&[i]' else 0)

|IN|<8> = 0

Relation OUT:

For all integer i,
(0 ≤ i ≤ 31)=

binary(#DISPLAY#[i]) = ||OUT||[i] <1:7>

Tabular Notation:

For requirements that are not so simple

Specification for a search program

	Normal Line	High Priority Line	non-existent
Normal Available	allocate line	allocate line	error message
Normal Full Reserve Available	return busy signal	allocate line	error message
No Reserve	return busy signal	return busy signal	error message

Break down situations into cases

Make sure you have covered all the cases

Fill in one square at a time.

You may have to divide columns and rows as you realise that you need to distinguish more cases.

**Divide and Conquer
or
Separation of Concerns**

This is essential in any systematic process

- Identify variables without considering functions.
- Consider one controlled variable at a time
- Use only monitored variables, adding as needed.
- Consider one case at a time.

Do this *before* programming. Don't let the programmer make those decisions!

The case identification will happen anyway. It is best to have it happen "up front".

After Requirements: Design

Module structure can be based on documents.
 Each function can be a separate module, but
 - find shared services to avoid duplication
 - make sure sharing is not a coincidence
 - introduce shared-service modules

After Requirements: Implementation

Programmers really use these requirements documents. The documents pass the “coffee stain” test.

Programmers do not use their own guesses and do not have to hunt down experts. Time is saved.

After Requirements: Review

Reviewers can also use these documents. They know what the code is supposed to do.

After Implementation: Testing

Testers can generate tests and evaluate test sets.

After Delivery: Change

Tables provide basis for change request (contract).

Tables During Design and Implementation.

The advantages of tabular notation do not stop at the requirements stage.

Tables can be used for systematic module interface documentation.

Tables can be used for program function documentation.

Specification for a search program

x in array	x not in array
------------	----------------

j'	B[j'] = x	<i>any value</i>
present' =	true	false

\wedge
 NC(x, B)

These tables are especially valuable for more complex functions.

Coding becomes easy.

More Formal Tables:

Specification for a search program

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

j'	B[j'] = x	<i>true</i>
present' =	true	false

\wedge
 NC(x, B)

You may not like it but it is better than

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

The math used is entirely conventional and well understood.

Every McMaster Software Engineering graduate masters the notation by the end of second year.

Using formal tables allows us to take advantage of tools.

What Can Tools Do For Us?

If they are math based, requirements tools can assist us with the process from elicitation to evaluation.

- Input tools
- Printing/Format tools
- Indexing Tools

Assist us in checking our documentation

- Syntax checking tools
- Completeness/consistency checking tools

Assist us in inspecting our programs

- Tools for managing systematic inspections

Assist us in testing our programs and making sure that programs and documentation are consistent.

- Test case generation
- Test completeness evaluation
- Test Oracle Generation
- Statistical Reliability Estimation

Transform tables into better formats

Check for table equivalence.

Manage an inspection process.

Why is Tabular Notation Superior?

- It applies the *divide and conquer* principle reducing a complex expression to a structured presentation of a set of simple expressions
- You don't have to read the whole expression to use it.
- It has already been "parsed" for you.
- It can be checked for completeness and consistency

Why Tables Retain the Advantages of Mathematics

- Tables are still mathematical expressions and have a simple and intuitive interpretation.
- Tables have precise meaning
- Tables can be interpreted/evaluated by tools.

Tables provide a precise notation that is readable.

Tables can be used in many contexts.

Tools Based on Tabular Notation

A table input tool that allows you input tabular expressions without worrying about appearance.

- Motivated by experience in industry.
- Several different styles of input tool accommodating different taste.
- All tools compatible.

A "table holder" that stores tabular expressions in format independent notation.

Two table printing tools.

- allow table formatting without table alteration
- based on experience in industry
- allow tables to be included in other documents
- newest tool includes indexing feature,

Table Checking Tool

Table Simplification Tool

Table Expression Evaluation Tool

Tabular Function Composition Tool

Alias evaluation tool

Two tools converting between table types.

Tools

Oracle Generator Tool

- generates a test oracle based on the documentation.
- can be used to keep program and documentation consistent.

Monitor Generation

- an Oracle for real-time systems

Test Set Evaluation

- will evaluate test coverage of a set of tests.
- produces a table showing the number of test cases that fell into each specification case.

Reliability Estimation Tool

- generates statistically meaningful test sets.
- uses the test oracle to evaluate test results.
- estimates the reliability of the program.

Other Possible Tools

System Simulator (SCR tools)

- Can be used to make sure you have specified what you want.
- Can be used in testing
- Checks for completeness and consistency

Statistical test generator Generator

- Uses a description of the operational profile
- Produces trace test cases
- Can be used to estimate reliability

Reliability Estimation Tool

- Combines the two tools above
- Gives the user a choice of statistical methods.

Tools for Requirements Documents

All of the Table Tools

- They can be used in all types of documents
- Completeness and Consistency Checkers
- Expression evaluation is needed.

NRL/SCR tools are quite professional.

Monitor Generator Tool

Why Aren't These Tools Part of a Compiler?

Many of the documents are to be used by people who should not need to look at the code.

The documents can be language independent and apply to more than one implementation.

The documents summarise behaviour without showing implementation.

Why Aren't These Tools Hot on the Market?

They are not on the market.

A few are not yet completed.

They are "proof of concept" tools designed to stimulate others to produce commercial products.

The only "professional" parts are:

- table holder - permits tool interchange
- table evaluator - used in other tools.

Review of the Procedure

The method is output-driven or controlled variable driven.

If you get the outputs right, nobody will care whether you read the inputs or how you used them.

Thus, the system is:

- Find all the controlled variables (tough)
- Find all the monitored variables
- Describe the value of each controlled variable as a function of time and the monitored variables.
- Make sure the domain includes all NAT?
- We use tables to divide up the domain into cases

We can check for completeness of a table if we have a separate description of NAT.

Where Has This Been Applied.

Aircraft Industry: A-7 and others
 Nuclear Industry: Darlington and continuing
 Bell Labs - Columbus - Service Evaluation System
 Others:
 • Engine controllers
 • Aircraft CAD design programs.
 • Water Level Monitoring System (van Schouwen)
 • Fuel Level Monitoring System (SPC - Core)
 •
 •
 •

Establish Requirements as an Engineer Would

Since "time immemorial" (before my Undergraduate time) Engineers have been told that they must have a precise statement of requirements before they begin on a project or they must determine that themselves.
 Engineers do use mathematics.
 They don't try to play "English teacher".
 Most software developers are not Engineers and do not have the educational qualifications. They learned in *ad hoc* ways on the job. Their knowledge is haphazard.
 Many so-called "Software Engineering" programmes do not teach people to be Engineers; they teach generalities about project management.

 Every programmer can learn a formal system (the programming language).

They can learn this too! It is easier.

Reports and Papers

CRL Report 291 "Transformations of Normal and Inverted Function Tables", Zucker, J.I.
 CRL Report 292 "Specifying and Simulating The Externally Observable Behaviour of Modules", Wang, Y.
 CRL Report 297 "Display Management System, A tool to support the Display Method", Wang, Y.
 CRL Report 302 "Generating a Test Oracle from Program Documentation", Peters, D.K.
 CRL Report 315 "Implementation of Table Inversion Algorithms", Shen, H.
 CRL Report 328 "Table Transformation Tools: Why and How", Shen, S., Zucker J.I., Parnas, D.L.,
 CRL Report 330 "Table Construction Tool", Li, W.
 CRL Report 337 "Software Reliability Estimation Tool", Li, ChunMing
 CRL Reports 339 & 340 "Table Tool System Developer's Guide", McMaster University Software Engineering Research Group
 CRL Report 346 "Evaluating Generalized Tabular Expressions in Software Documentation", Abraham, R.F.
 CRL Report 352 "A Grand Table Interface Specification/ Developer's/User's Guide", Vulcanovic, I., von Mohrenschildt, M.
 CRL Report 360 "Specialization: An Approach to Simplifying Tables in Software Documentation", Rastogi, P.
 CRL Report 363 "Table Transformations: Theory and Tools", Zucker J.I., Shen, H.

CRL Report 364 "Function Composition Tool", Tyson, A.H.,
 SERG Report 375 "Generating Indexed Formal Software Documents", Wang, L.
 SERG Report 378 "Semantic Equality of Tables", Nadarajah, K.
 Janicki, R., Parnas, D.L., Zucker, J., "Tabular Representations in Relational Documents", in "Relational Methods in Computer Science", Chapter 12, Ed. C. Brink and G. Schmidt. Springer Verlag, pp. 184 - 196, 1997, ISBN 3-211-82971-7.
 Peters, D., Parnas, D.L., "Using Test Oracles Generated from Program Documentation", *IEEE Transactions on Software Engineering*, Vol. 24, No.3, March 1998, pp. 161 - 173
 Parnas, D.L., Madey, J., "Functional Documentation for Computer Systems Engineering" published in *Science of Computer Programming* (Elsevier) vol. 25, number 1, October 1995, pp 41-61
 van Schouwen, A.J., Parnas, D.L., Madey, J., "Documentation of Requirements for Computer Systems", *Proceedings of '93 IEEE International Symposium on Requirements Engineering*, San Diego, CA, 4 - 6 January, 1993, pp. 198 - 207
 Parnas, D.L. "Inspection of Safety Critical Software using Function Tables", *Proceedings of IFIP World Congress 1994, Volume III* August 1994, pp. 270 - 277.
 Parnas, D.L., "Using Mathematical Models in the Inspection of Critical Software", in *Applications of Formal Methods*, Hinchey M.G., Bowen J.P. (eds.), Prentice Hall International Series in Computer Science, 1995, pp. 17-31.
 Shen H., Zucker J.I., Parnas, D.L., "Table Transformation Tools: Why and How", *Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS '96)*, published by IEEE and NIST, Gaithersburg, MD., June 1996, pp. 3-11.