Requirements Documentation: A Systematic Approach

David Lorge Parnas,P.Eng,
Ph.D, Dr.h.c., Dr.h.c., FRSC, FACM, FCAE

SFI Fellow, Professor of Software Engineering
Director of the Software Quality Research Laboratory (SQRL)
Department of Computer Science and Information Systems
Faculty of Informatics and Electronics
University of Limerick

Abstract

Unless you have a complete and precise description of a product’s requirements, it is very unlikely that those requirements will be satisfied. An incomplete or inconsistent requirements document can mislead 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 an informal requirements statement into a mathematical language, and show that the result is complete and unambiguous; the original may still be faulty.

This talk describes a sound procedure for documenting requirements - one that lets you know when your document is complete and consistent. 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.

The Responsibilities of an Engineer

An Engineer's responsibility includes making sure that products are "*fit for use*".

This requires precise, detailed, communication with the application experts.

Communication must be on the basis of a *written* statement of the requirements:

- that can be read and analysed *by both users and implementors*
- that can be used to *evaluate* the product after completion
- that can be used to *settle disputes after completion*.
- that can be used to communicate the eventual changes that are required.

No engineer can fulfil his/her responsibilities without such a document.

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 acceleration, 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 were never documented or reviewed by users.

Those with a resting heart rate between the possible set points will have to chose between *too low* and *too high*.

Some of the parameters not meaningful to doctors they use default values. No instructions are given to doctors.

I hope that this would not have happened if the planned behaviour had been communicated to doctors and they had reviewed them carefully.

Why Programmers Should Not Make User-visible Decisions

Context:

- U.S. attack aircraft with two altimeters; pilot can choose the preferred one
- Software checks functionality, switches to other device if readings are unreasonable.

Question:

What should we do if both altimeters are broken?

- Programmer asks pilot: What is average altitude of flight?
- Program displays the average altitude when no altimeter is functional.

The answer to the real question: Flash Pull-Up cue.

The result: Pilots have been trained to watch for “average” altitude appearing for more than a few seconds and told to “pull up” if that happens.

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 constitute a precise requirements document.

Building on Rock vs. Building on Sand

My basic assumptions:

- Software that is "almost right" is wrong.
- Imprecisely defined notation leads to software that is almost right.
- Notations used in tools are precisely defined, but that definition may not be clear and simple. (e.g. Statecharts)
- Mathematical definitions of software notations are essential, but must be simple and based on standard mathematics.
- Mathematics can be simple and not fundamentally new.
- Undefined Modelling Languages (UMLs) are quicksand. 💣
- Documentation can be useful for design, review, implementation, testing and maintenance.
- More practical difficulties with documentation result from issues about the required content of documents than from issues about format.

<u>What Could we do with Mathematical Descriptions of software?</u>

(1) Describe software products that we already have - so that people can answer questions about them without reading the code.

(2) Write specifications for software products we do not yet have - so that the programmers and clients can reach agreement on the requirements.

(3) Verify that a product meets its requirements (testing and/or inspection).

(4) Build tools to check specifications

(5) Build tools to simulate systems and check systems.

<u>What are the Acceptance Criteria for the Descriptions?</u>

(1) <u>Descriptions must be easier to understand than the code</u>.

(2) Documentation must state the requirements in a way that does not restrict the solutions unnecessarily.

(3) Testing and proof could 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.

The Purpose of a Requirements Document

We are talking about a technical document written for developers.

- Requirements documentation should not be a sales pitch!
- Requirements documentation should not be an introduction.
- Requirements documentation should serve as a reference document during development.
- Requirements documentation should prevent programmers from taking unilateral decisions that should be made with others.
- Requirements documentation should be used for design reviews.
- Requirements documentation should be used for test case generation and test result evaluation.
- Requirements documentation should be used, and kept up to date, during maintenance.
- Requirements documentation can be modified to specify revisions.

<u>The Goals of the Requirements Phase</u>

A. Decide what to build before starting to build it:

- Make “what decisions” explicitly before design, not implicitly during design.
- Make sure you build what is needed.
- Allow future users, or representatives, to comment <u>before</u> the product is built.

B. Provide an organised reference document for the software developers:

- Provide accurate, consistent information.
- Answer constraint questions only once.
- Relieve them of any need to decide what is best for the user.
- Compensate for their ignorance of the application environment.
- Give them the information needed to make good design decisions.
- Allow her to make accurate estimates of time and resources needed.

C. Allow for personnel turnover.

- Record what has been learned for future replacements.

The Goals Of The Requirements Phase

D. Provide a reference document for a Quality Assurance Group

- Test design should not depend on program.
- Authority required:---Q.A. and programmer may disagree.

E. 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.

F. Specify constraints for future revisions.

Note: A requirements document is not a “write, use (at most) once and discard” document. A good document, properly maintained, will be useful until the product is discarded and possibly for the next product.

Writing Down Requirements

The most costly errors are those made early in the process

Early errors are the hardest to change.

Misunderstandings about requirements lead to early mistakes.

Programmers need to be told what is needed now; they must also be told what is subject to change.

Requirements must be subject to review.

Safety reviews of software should be based on the statement of requirements, not the code.

Maintenance actions must be based on requirements.

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

That *written* statement must be precise and complete.

That *written* statement must be organized for reference so that information appears in a specified place and will not be missing or (almost) duplicated.

The Two-variable Model

The “traditional model” of a hardware/software system 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.

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

Review by application experts is hard to get.

The Four Variable Model

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

Peripheral devices sense monitored variables and determine computer inputs.

Peripheral devices read computer outputs and control 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

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

Subject-matter experts review requirements in terms of monitored/ controlled variables.

Software requirements are described using input and output variables.

How to Document System Requirements

The first step is 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.

For many projects you cannot even find a complete list of these variables and there is no agreement on what they are.

Software documents may ignore them.

Defining the Meaning of Monitored and Controlled Variables

This is not a Computer Science problem.

This is a conventional engineering problem.

- It may require drawing diagrams.
- There must be a coordinate system explicitly defined.
- The units must be explicitly defined.
- 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.

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 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,

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

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!

The Documentation Procedure - Roughly

(1) Produce a complete list of “controlled variables”.

(2) Produce a complete list of monitored variables.

- For each monitored variable specify the set of possible values
- If appropriate specify the set of possible time functions (histories) of those values.
  - for continuous variables, derivatives must be limited
  - for discrete variables possible sequences
- This defines the domain of NAT

(3) For each of the controlled variables, describe its value at time *t* as a function of the values of monitored variables at time *t* or earlier. The domain of the function must include the domain of NAT.

If the list of controlled variables is complete, and each function defined for all possible values of the monitored variables, the document is complete.

Because each controlled variable is mentioned only in one function, if it is a function, the document is consistent.

“Divide and Conquer” makes it work.

The Documentation Procedure - Some Refinements

Describe functions using tabular expressions

- Divide and Conquer applied again - by cases.

Describing groups of controlled variables together can simplify the job.

- Important when values change at the same events
- Important when values are strongly related
- Important when values depend on same subset of monitored variables.

New controlled variables will be discovered during the process.

- Users will realise that they overlooked important functions.

New monitored variables will be added.

- System may monitor itself.
- Some outputs may depend on monitored variables that were overlooked.

Shared function definitions simplify descriptions.

- Many functions have same special cases (e.g. account overdue)
- Modes of operation affect many functions.

A Valuable Special Case: Systems Characterised by Modes and Current Values.

For many systems, only some history is relevant.

This can often be summarised by identifying “modes of operation”.

There will often by a small finite number of mode classes each with a small finite number of mode states.

The current mode in each class can be defined by transition tables.

The controlled values are then a function of the current mode and the current inputs.

For this class of systems, we can build monitoring test systems.

We can use other summaries of the past history (such as average values) in the formulae in the tables.

Modes And Their Use

Modes are classes[1] of states:

- There are too many states to deal with directly.
- The actual states are implementation dependent.
- Modes characterise the history of the system.
- The purpose of modes is to simplify the function descriptions. Choose them accordingly.

There can be several classes of modes.

- Each mode class partitions the set of states
- A system is always in one and only one mode from each class.
- Designing in these terms can help you to avoid the complications encountered by Statemate users.

There can be interactions (excluded combinations). These should be minimised!

Mode transitions are caused by events.

[1] The distinction between states and classes of states is essential for a simple underlying model. Calling everything “state” is a mistake.

Modes can be defined by transition tables

Mode Transition Table “007”

	Airplane	**ATV**	**Submarine**
Airplane		@T(weight on wheels)	@T(wet)
ATV	@F(weight on wheels) ***when*** ¬ wet		@T(wet)
Submarine	@F(wet) ***when*** ¬ (weight on wheels)	@F(wet) ***when*** (weight on wheels)	

Assumption:

weight on wheels, and wet are detectable conditions.

“wet” is not rain, it means “in a body of liquid”.

Two Views of Modes

Modes are classes of event histories.

- Each Mode Class corresponds to a partitioning of the set of event histories.
- Each Mode in a mode class is one of the partitions of that partitioning.

Modes are classes of system states

- Each Mode Class corresponds to a partitioning of the set of system states.
- Each Mode in a mode class is one of the partitions of that partitioning.

In a correct system these are equivalent views.

The first is a black box view and the second a clear box view.

The Basic Structure of a System Requirements Specification

A. Computing Resources:

- All resources available. System must demand no more.
- This section is often omitted when a standard support system is used.

B. Monitored Environmental Variables:

- Conventional description of the physical quantity (physical interpretation).
- Define a data type for each.
- Describe all possible values.

C. Controlled Environmental Variables:

- Conventional description of the physical quantity (physical interpretation)
- Define a data type for each.
- Describe all possible values.

D. Feedback:

- Describe relations between Controlled and Monitored Variables. (NAT)
- This section can often be omitted

The basic structure of a System Requirements Specification?

E. Environmental Conditions (NAT)

- Usually a predicate on values of M

F. Behavioural Requirements (REQ)

- Mode definitions belong here
- Usually one relation for each element of C.
- Occasionally one relation for all of C.
    - Domain is set of values for M.
    - Range is set of values for C.
    - Usually a function, but
- Tolerance supplied separately (see below).
- May be a set of functions.

G. Dictionaries: for items used more than once.

- functions,
- types,
- constants,

The basic structure of a System Requirements Specification?

H. Exceptions, Undesired Events.

- A useful redundant check.
- force designers to think about these things in advance.
- Improper response to UEs one of the worst aspects of today's software

I. Timing and Accuracy Constraints (tolerance tables):

- Accuracy for approximations to reals.
- Timing for discrete values.

J. Assumptions & Expected Changes:

- Use 2 complementary lists: likely changes, fundamental properties.
- Requires real stretching of imagination.
- Spartan systems are still better than nothing.

K. Sources:

- People
- Documents
- Other sources (e.g. experiments)

How can we document system design?

If you contract for software separately:

i^t denotes the vector valued time function

$$(i^t_1, i^t_2, \bullet\bullet\bullet, i^t_r)$$

one element for each of the input registers

o^t denotes the vector valued time function

$$(o^t_1, o^t_2, \bullet\bullet\bullet, o^t_q)$$

one element for each of the output registers

Document the following relations:

Relation IN:

domain contains values of $\underline{m}^t$

range contains values of $\underline{i}^t$

$(\underline{m}^t, \underline{i}^t)$ is in IN if and only if input device permits that behaviour

It must be the case that

(1) domain(IN) $\supseteq$ domain(NAT)

Relation OUT:

domain contains the possible values of $\underline{o}^t$

range contains the possible values of $\underline{c}^t$

$(\underline{o}^t, \underline{c}^t)$ is in OUT if and only if output device permits that behaviour

Additional Chapters in System Design

1. Input Devices:
   •Define relationship between representation and an abstract data type.
   •Describe IN(M,I) (one table per element of I).
   •Do <u>not</u> specify purpose.

2. Output Devices:
   •Define relation between representation and an abstract data type.
   •Describe OUT(O,C).
   •Do <u>not</u> specify the required values.

How can we Document Software Requirements?

Good News:

Software requirements determined by system design and system requirements

(1) $REQ(\underline{m}^t, \underline{c}^t)$

(2) $IN(\underline{m}^t, \underline{i}^t)$

(3) $OUT(\underline{o}^t, \underline{c}^t)$ and

(4) $NAT(\underline{m}^t, \underline{c}^t)$

The actual software can be described by

(5) $SOF(\underline{i}^t, \underline{o}^t)$

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

How Can We Describe Event Classes?

An event class is defined by a relation on the state set: {(old state, new state)}.
Note: State here is the environmental state.

Example:

('altitude > 0 ^ 'wheels = \$retracted\$) ∧
(altitude' = 0^ wheels' = \$retracted\$)

Convenient Notation.

@T(predicate) ***when*** (predicate)

Example

@T(altitude = 0 ∧ wheels = \$retracted\$)
or
@T(altitude = 0) ***when*** (wheels = \$retracted\$)

OR - the union of two relations.

AND - the intersection of two relations.

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.

Why are Functional Approaches Superior?

Functions summarise behaviour, they do not detail intermediate steps or internal structures.

Functions can often be described in closed form expressions.

Implementation of monitors/oracles can be much more efficient.

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>

This Example is too Simple

There is no description of the timing aspects of NAT

If you can have simultaneous changes in two types of stock, two changes closer together than 50ms, or add/remove more than one donut at a time, this system will not work.

It is better to find that out at this stage than after coding.

All of the decisions would have eventually been made - no extra work.

“Up front investment in details, saves time in the end.

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.

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 testing our programs and making sure that programs and documentation are consistent.

- Test case generation
- Test completeness evaluation
- Test Oracle Generation
- Monitor program 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.

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.

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)
-
-
-

Take Time to Save Time