Identifying and Measuring Quality in a Software Requirements Specification

Alan Davis, Scott Overmyer, Kathleen Jordan, Joseph Caruso,
Fatma Dandashi, Anhtuan Dinh, Gary Kincaid, Glen Ledeboer,
Patricia Reynolds, Pradip Sitaram, Anh Ta, and Mary Theofanos

Abstract

Numerous treatises exist that define appropriate qualities that should be exhibited by a well written software requirements specification (SRS). In most cases these are vaguely defined. This paper explores thoroughly the concept of quality in an SRS and defines attributes that contribute to that quality. Techniques for measuring these attributes are suggested.

I. Introduction

Software metrics may be used to measure attributes of software process or intermediate or final products of software development. One early intermediate product of software development is the software requirements specification. A *software requirements specification* (SRS) is a document that describes all the externally observable behaviors and characteristics expected of a software system. Generally, a *quality* SRS is one that contributes to successful, cost-effective creation of software that solves real user needs. Specifically, a *quality* SRS is one that exhibits the following qualities:

1. Unambiguous	13. Electronically Stored
2. Complete	14. Executable/Interpretable
3. Correct	15. Annotated by Relative Importance
4. Understandable	16. Annotated by Relative Stability
5. Verifiable	17. Annotated by Version
6. Internally Consistent	18. Not Redundant
7. Externally Consistent	19. At Right Level of Detail
8. Achievable	20. Precise
9. Concise	21. Reusable
10. Design Independent	22. Traced
11. Traceable	23. Organized
12. Modifiable	24. Cross-Referenced

The purpose of this paper is to provide the beginnings for definitions of metrics suitable for these qualities.

This work was supported in part by a grant from the El Pomar Foundation. Affiliations: Davis: U. of Colorado at Colorado Springs, Colorado Springs, CO 80933-7150; Overmyer: Minot State U.; Jordan: Institute for Defense Analyses; Caruso, Dandashi, Dinh, Ledeboer: George Mason University.; Kincaid: Calspan; Reynolds, Ta: Mitre; Sitaram: STX Hughes; Theofanos: Oak Ridge Lab.

There are many errors being introduced into SRS's. In 1981, Basili's requirements specification team reported 88 errors in the 400 page A-7E Operational Flight Program software requirements specification [BAS81]. And that SRS was written by a group of requirements writing experts. Also in the late 1970's, Celko reported that applying requirements checking tools to an SRS for an existing Army management information system revealed the presence of many hundreds of errors [CEL81]. DeMarco, as quoted by Tavolato and Vincena [TAV84], reports that 56% of all errors ever made on a software development effort can be traced to errors in the SRS. Boehm reports that 45% of all errors made on software development efforts at TRW can be traced to either requirements or preliminary design [BOE75]. Obviously, if we can better understand how to recognize and measure quality in an SRS, we will be better equipped to detect errors in the SRS.

To make matters worse, SRS errors need to be detected during the requirements phase, or the cost to repair them will grow significantly. Three analyses [DAL77, BOE76, FAG74] provide conclusive evidence that the later in the life cycle an error is detected and repaired, the more it will cost. These show a 200:1 ratio between detecting and repairing an error during requirements vs. maintenance phases. It is only with data from Boehm [BOE75] that we can also see that the reason for the cost increase is that errors are remaining latent. That is, if we detect an SRS error when writing it, all we do is fix it. If we detect that same SRS error during design, we must fix both the design as well as the SRS. If we detect the same SRS error during coding, we must fix the code, design, and SRS, etc. [MIZ83]. If we can better understand how to recognize SRS quality, we will be better equipped to detect SRS errors and thus prevent them from remaining, and thus costing more to detect and repair. Let us not fail to recognize that there are two different general classes of requirements errors: knowledge errors and specification errors. *Knowledge errors* are caused by not knowing what the true requirements are. *Specification errors* are caused by not

0-8186-3740-4/93 $3.00

knowing how to adequately specify requirements. Knowledge errors can be reduced through prototyping [AND89, DAV92]. However, there may exist knowledge errors that cannot be found until after the system is deployed. There is little excuse for specification errors.

The above list of SRS qualities is a compilation of lists made by others. See Figure 1. These authors however have not attempted to provide useful ways of measuring SRS quality. The implications if we ignore SRS quality are [DAV93]:

- The resulting software may not satisfy user needs
- Multiple interpretations may cause disagreements between customers and developers
- It may be impossible to thoroughly test [DAV90a]
- The wrong system might be built.

As attempts are made to achieve quality in an SRS, one must be careful to recognize that although quality is attainable, perfection is not. Any of the above 24 quality attributes can be achieved, but often at the expense of other attributes. On any one given project, requirements writers need to agree as to which quality attributes are most important, and strive for those.

II. SRS Quality Attributes

A quality SRS is one that exhibits the 24 attributes listed in the introduction, i.e., is devoid of any errors that would violate these attributes. The following 24 subsections (1) define each attribute, (2) provide ideas on measuring the attribute, (3) provide the attribute with a recommended weight relative to other attributes, and (4) describe types of activities that can be used to optimize presence of that attribute. In all cases, we assume there are n_r requirements in the SRS, and the set of all these requirements is denoted as R. In addition, we assume that there are n_f functional requirements (R_f) and n_{nf} non-functional (i.e., ilities) requirements (R_{nf}) in the SRS, where $n_r = n_f + n_{nf}$, and $R = R_f \cup R_{nf}$.

2.1 Unambiguous

An SRS is *unambiguous* if and only if every requirement stated therein has only one possible interpretation [IEE84]. Ambiguity is a function of the backgrounds of the reader. For example, "generate a dial tone" may be ambiguous to non-telephony people because they do not realize that standards exist that demand a dial tone be of a specific frequency. Due to these standards, telephony people in domestic systems may see the term as totally unambiguous. Strangely, telephony people in domestic and international systems would once again find it ambiguous due to conflicting standards.

Certain languages are inherently more ambiguous than other languages. Perhaps there is a measure of inherent ambiguity of various languages. Deterministic finite state machines (FSM), Petri nets (PN), decision trees (DT), propositional calculus, predicate calculus and many others all have well defined semantics and thus suffer from no inherent ambiguity. Natural language or any formalism that includes natural language (e.g., structured English) has much inherent ambiguity. Once you choose to use less ambiguous forms of expression, the specific choice will be driven primarily by expressive power and suitability for the aspect of the system, than by its inherent ambiguity. Since ambiguity is primarily in the eyes of the reader, one way to measure it is via review, i.e., as the percentage of requirements that have been interpreted in a unique manner by all its reviewers, i.e.,

$$Q_1 = \frac{n_{ui}}{n_r}$$

where n_{ui} is the number of requirements for which all reviewers presented identical interpretations. This ranges from 0 (every requirement has multiple interpretations) to 1 (every requirement has a unique interpretation). Because unambiguity is so critical to project success, we recommend a weight of 1, i.e., $W_1 = 1$.

Replacing natural language with formal notations, e.g., FSMs, PNs, DTs, greatly decreases ambiguity in the SRS but almost always at the expense of understandability[1]. A better approach is to *augment* natural language with more formal models. That way, the advantages of both natural and formal languages are preserved.

2.2 Complete

An SRS is *complete* if:

- o Everything that the software is supposed to do is included in the SRS [DAV93]
- o Responses of the software to all realizable classes of input data in all realizable classes of situations is included [IEE84]
- o All pages numbered; all figures and tables numbered, named, and referenced; all terms defined; all units of measure provided; and all referenced material present [IEE84]
- o No sections marked "To Be Determined" [DAV93].

[1] Decision trees are one of the few exceptions. When applicable, they can be used with no explanation and can be easily understood by the layperson.

Requirements Quality Factor	Reference															
	BOE 74	ALF 76	BEL 76	DAV 79	BAS 81	ZAV 81	CEL 83	IEE 84	NCC 87	ESA 87	DOD 88	JPL 88	CAR 90	DAV 90	ROM 90	DAV 93
Unambiguous		X	X	X	X	X	X	X	X	X		X		X	X	X
Complete	X	X	X	X	X	X	X	X	X	X		X		X	X	X
Correct		X	X		X							X		X	X	X
Understandable		X		X		X	X	X	X	X	X			X	X	X
Verifiable		X	X			X		X		X	X	X		X	X	X
Consistent (Internal)	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X
Consistent (External)			X						X	X	X	X			X	X
Achievable		X	X							X					X	
Concise													X			X
Design Independent		X	X	X		X						X				X
Traceable	X	X						X		X		X		X	X	X
Modifiable		X			X	X		X						X	X	X
Electronically Stored									X							
Interpretable/ Prototypable						X									X	
Annotated by Relative Importance										X		X		X		X
Annotated by Relative Stability										X				X		X
Annotated by Version																
Not Redundant		X	X	X				X		X				X		X
At Right Level of Detail															X	
Precise						X			X	X					X	
Reusable																
Traced		X					X	X			X	X		X		X
Organized		X							X						X	X
Cross-Referenced																

Figure 1. Attributes of an SRS

Obviously, if an SRS is incomplete by the first meaning, users will not be satisfied when the system is deployed. If an SRS is incomplete by any other definition, developers are likely to make assumptions about intended behavior, and those assumptions may be false, leading once again to unsatisfied users.

Given the first definition, completeness is extremely difficult to measure; it is generally agreed that the more requirements we include in an SRS (or see in a system),

the more new requirements we will think of. We are thus trying to measure a moving target. However, using the second definition, there are some metrics that do make sense. For example, completeness implies that the function *f(state, stimulus) --> (state, response)* is defined for all elements in the cross product *state* x *stimulus*. Assuming we count numbers of inputs, i.e., stimuli (n_i) specified in the SRS, and numbers of states (n_s) defined in or implied by the SRS, then we know their product (n_i x n_s) is the total number of function values that must be specified. If we now count the actual unique functions (n_u) specified (Note that $n_u <= n_f$ because some of the n_f functions could be redundant), we can measure completeness by the equation,

$$Q_2 = \frac{n_u}{n_i \times n_s}.$$

This measures percentage of necessary functions specified. It may be useful in well understood, bounded, problem domains. It does not address completeness of non-functional requirements. Jaffe, et al. [JAF91] have a done a remarkable job of delineating all types of requirements that must be present in a FSM-based SRS in order to declare it complete.

In less understood, less bounded, problem domains, it is likely that stimuli and states specified in the SRS are themselves incomplete. Alexander [ALE90] provided ideas that may be of help here. Figure 2 provides an omniscient view of all requirements for a system, i.e., assume we are able to look to the future and ascertain all requirements that users will ever need. Block A represents requirements that we know, and that we know are applicable to this problem; these are the requirements typically captured in an SRS. Block B represents requirements that we know, but have not really thought about or verbalized; these are typically uncovered during interviews or brainstorming. Of course, once uncovered, they move to block A. Block C represents requirements that we know we need, but don't understand them well enough to describe them; these are typically uncovered during prototyping. Once uncovered, they move to block A. Block D represents potential requirements that we don't know, and that we don't even know we don't know. Prototyping may help uncover these because sometimes seeing one feature makes us aware of another. Once uncovered, they tend to move to block B. Arrows in Figure 2 show requirements migration. Notice the trend is all requirements moving to block A. All requirements in block A is equivalent to the first definition of completeness. A measure of the percentage of requirements that are in block A could be an effective measure of completeness, i.e.,

$$\frac{n_A}{n_A + n_B + n_C + n_D}$$

where n_A, n_B, n_C, and n_D are numbers of requirements in blocks A, B, C, and D, respectively. Values range from 0 (totally incomplete) to 1 (complete). Since we do not know how to measure the areas of blocks C or D, an alternative might be:

$$Q_{2'} = \frac{n_A}{n_A + n_B}.$$

Once again, values range from 0 (totally incomplete) to 1 (complete).

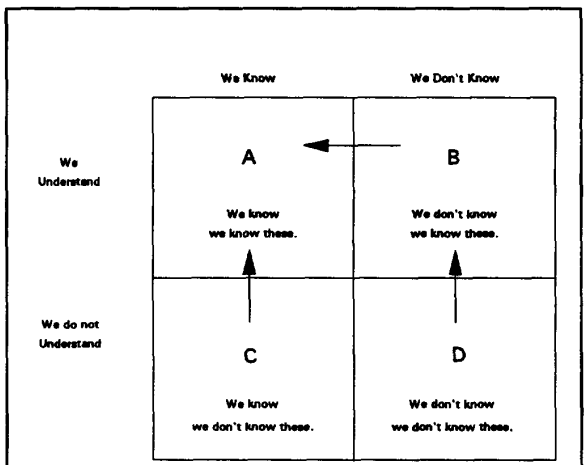

Figure 2. Alexander's Requirements Completeness Model

Another alternative is to measure local completeness, i.e., percentage of all recognized requirements that have been documented in the SRS. Figure 3 is a variation of Figure 2 where the vertical axis represents whether or not a requirement appears in the SRS, and the vertical axis represents the degree to which a requirement is understood. Block A represents requirements that we know, and that we have captured. Block B represents requirements that have been documented, but are either poorly specified, abstractly stated, or not yet validated. Once firm, they move to block A. Block C represents requirements that we know we need, but have not yet specified. Once documented, they move to block A. Block D represents potential requirements that we don't understand well enough to document. If we choose to specify them abstractly (as a place holder), they move to block B. If we choose to investigate their validity first, say via a prototype, and we grow to understand them, then they move to block C. Arrows in Figure 3 show

requirements migration. Notice the trend is all requirements moving to block A. Given this model, we could measure completeness as the percentage of requirements in the SRS that are well understood, i.e.,

$$Q_{2'} = \frac{n_A}{n_r}$$

or alternatively as the percentage of known requirements that have been documented in the SRS, i.e.,

$$Q_{2''} = \frac{n_r}{n_A + n_B + n_C + n_D}$$

where n_A, n_B, n_C, and n_D are numbers of requirements in A, B, C, and D, respectively. In both cases, $n_r = n_A + n_B$ and values range from 0 (totally incomplete) to 1 (complete). Regardless of which is used, we recommend a weight of approx. .7, i.e., $W_2 = .7$ because completeness is critical to project success but difficult to measure.

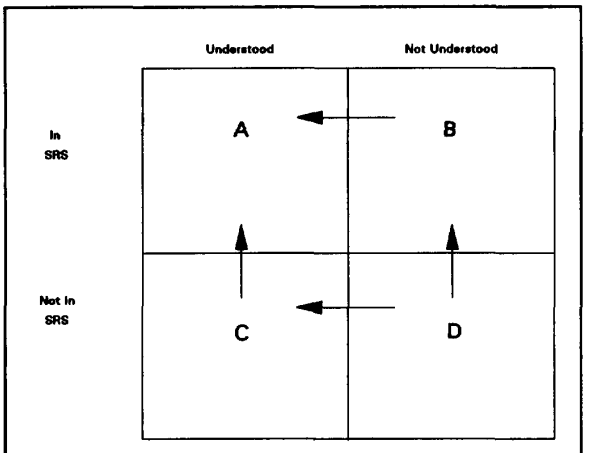

Figure 3. Local Requirements Completeness Model

To achieve completeness by any definition, reviews of the SRS by customer or user are essential. Prototypes also help raise awareness of new requirements and help us better understand poorly or abstractly defined requirements [AND89, DAV92].

2.3. Correct

An SRS is *correct* if and only if every requirement represents something required of the system to be built [DAV93], i.e., every requirement in the SRS contributes to the satisfaction of some need.

Since the term correctness applies to an individual requirement and an entire SRS, one convenient way of measuring correctness of an SRS might be to measure the percentage of individually correct requirements, i.e.,

$$\frac{n_C}{n_C + n_I}$$

where n_C and n_I are the numbers of correct and incorrect requirements, respectively, and $n_r = n_C + n_I$. Values range from 0 (totally incorrect) to 1 (totally correct). Ironically, if we could measure correctness by the above formula, we would have to know which requirements were incorrect, and we would remove them, making it 100% correct! Thus applying the above formula will always result in a score of 1. A more practical, but less theoretically satisfying, is to measure percentage of requirements in the SRS that have been validated. We arrive at a more practical measure of completeness:

$$Q_3 = \frac{n_C}{n_C + n_{NV}} = \frac{n_C}{n_r}$$

where n_C and n_{NV} are numbers of correct and not (yet) validated requirements, respectively, and once again, $n_r = n_C + n_{NV}$. Because correctness is so critical to project success, we recommend a weight of 1, i.e., $W_3 = 1$.

There is no oracle against which to validate correctness of a requirement. The only technique is to involve people who have the problem or mission. In effect, they serve as oracles. They can read and study the SRS, or can witness or manipulate a prototype.

2.4 Understandable

An SRS is *understandable* if all classes of SRS readers can easily comprehend the meaning of all requirements with a minimum of explanation. Readers include customers, users, project managers (PM), software developers, and testers. In general, the first three desire ease of reading, and thus natural language is ideal. Obviously, if users and customers cannot understand the SRS, they cannot intelligently approve it, leaving success of the product outcome to chance. In general, the last two desire to ascertain precisely what the system is expected to do, and thus formal language is ideal. Obviously, if designers and testers cannot understand the SRS, it is impossible to build or test the system. The burden of creating an understandable SRS falls on the shoulders of the writers; it is not the readers' responsibility to learn everything writers know in order to digest the SRS.

Measuring understandability is difficult. If we could measure the degree of understandability on a scale, we have a graph like Figure 4. A point reflects the degree to which an SRS is understandable by two categories of readers. Use of a technique may contribute to moving this point, e.g., adding DTs (which appear regularly in common literature without semantic explanation) to an

SRS in an appropriate manner would move the point toward the northeast, i.e., increased understandability by both parties. Adding PNs to an SRS in an appropriate manner would move the point toward the southeast, i.e., increased understandability by developers and testers, but decreased understandability by customers, users, and PMs. The only measure we can conceive of is

$$Q_4 = \frac{n_{ur}}{n_r}$$

where n_{ur} is the number of requirements for which all reviewers thought they understood. This ranges from 0 (every requirement understood) to 1 (no requirement understood). Because understandability is so critical to project success, we recommend a weight of 1, i.e., $W_4=1$.

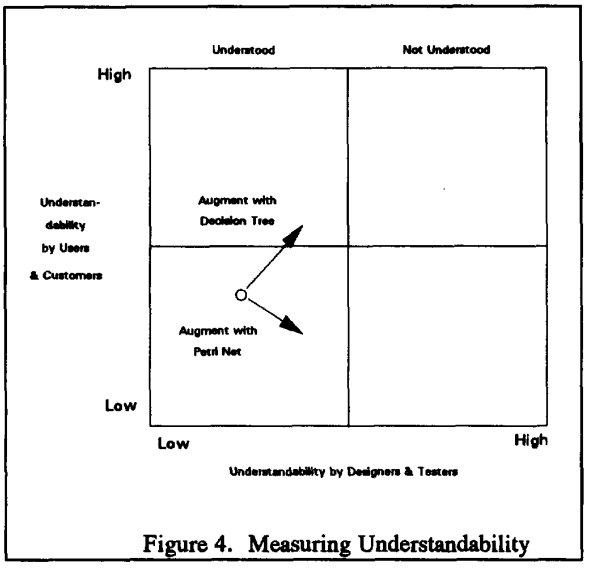

Figure 4. Measuring Understandability

A variety of techniques are available to determine and/or improve SRS understandability. First is to let representatives of all reader classes read it and comment. Assuming this is done repeatedly, and the SRS is revised accordingly, it may converge upon understandability. Augmentation with a prototype can improve effective understandability because it is often easier to see a prototype's behavior than to read a document.

2.5 Verifiable

An SRS is *verifiable* if there exist finite, cost effective techniques that can be used to verify that every requirement stated therein is satisfied by the system as built. Some requirements are easy to test: WHENEVER THE BUTTON X IS BEING PRESSED, THE LIGHT L SHALL BE LIT. Others are difficult to test: THE SOFTWARE SHALL EXHIBIT A FRIENDLY EASY-TO-USE INTERFACE WITH THE USER. There are a variety of reasons why a requirement may be difficult to verify:

a. Ambiguous. Any requirement with ambiguity will fare poorly for verifiability. If multiple interpretations exist for a requirement, there is no way to verify it [DAV90a].
b. Undecideable. Any requirement that is equivalent to the halting problem renders it unverifiable. Thus the requirement THE SYSTEM SHALL NEVER HALT is not verifiable.
c. Not worth cost (financial or life). For example, the requirement, IN THE CASE OF A REACTOR MELTDOWN, THE SYSTEM SHALL REDUCE THE DEATHS OF PERSONNEL WITHIN A 20 MILE RADIUS BY AT LEAST 80% is not worth the cost to test.

Measurement of verifiability is difficult. When verifiability is related to ambiguity, we have already seen it is impossible to adequately measure (see Section 2.1). When verifiability is related to the halting problem, the requirement either is or is not verifiable. A measure of percentage of requirements whose verification is undecideable is not helpful. There are some measurement avenues for cost-effectiveness or finiteness of the verification approach. If $c(r_i)$ and $t(r_i)$ are the cost and time necessary to verify presence of requirement r_i, then

$$Q_5 = \frac{n_r}{n_r + \sum_i c(r_i) + \sum_i t(r_i)}$$

measures inherent SRS verifiability where 0 means very poor verifiability, and 1 means very good verifiability. Verifiability is relative important to project success, so we recommend a weight of .7, i.e., $W_5=.7$.

Techniques to help verifiability are (1) all techniques described above for ambiguity, (2) knowledge of undecideability and review for its presence, and (3) review of SRS by experienced testers who can determine high cost or schedule testing implications.

2.6. Internally Consistent

An SRS is *internally consistent* if and only if no subset of individual requirements stated therein conflict [IEE84].

Measuring internally consistency is easier if we think of the SRS as defining a function that maps inputs and states into outputs and states, i.e,. treat it as an FSM. A consistent SRS is now one that can be described as a deterministic FSM. Any nondeterminism implies the SRS defines two different system responses or next states in identical situations. Assuming that we enumerate all stimuli (n_i) specified and all states (n_s) defined in or

implied by the SRS, then we know there should be exactly (n_i x n_s) total function values that must be specified to be complete, consistent and non-redundant. We now count the actual unique functions (n_u) specified (Note: $n_u <= n_f$), and we count how many of them are nondeterministic (n_n), i.e., how many of them map the same point in the domain into different points in the range. Then a measure of internal consistency is the percentage of unique functions that are deterministic:

$$Q_6 = \frac{n_u - n_n}{n_u}.$$

Values range from 0 (100% internally inconsistent) to 1 (100% internally consistent). Internal consistency is critical to project success, so we recommend a weight of 1, i.e., $W_6=1$.

Internal consistency is most easily achieved with tools like RLP [DAV79a] or REVS [ALF85]. Both provide consistency error reports for SRSs specified using multiple FSMs. Most CASE tools do consistency checking for data flow diagrams (DFD) and rudimentary consistency checking among DFDs and FSMs.

2.7 Externally Consistent

An SRS is *externally consistent* if and only if no requirement stated therein conflicts with any already baselined project documentation. These baselined documents include system-level requirements specifications (RS), statements of work (SOW), white papers, an earlier version of the SRS to which this new SRS must be upward compatible, and RSs for other systems to which this system must interface.

Measuring external consistency is more difficult than internal consistency. The best measure we can arrive at is the percentage of requirements that are consistent with all other documents, i.e.,

$$Q_7 = \frac{n_{EC}}{n_{EC} + n_{EI}} = \frac{n_{EC}}{n_r}$$

where n_{EC} is the number of requirements in the SRS that are consistent with all other documents and n_{EI} is the number that is not. Note that $n_r = n_{EC} + n_{EI}$. External consistency is critical to project success, so we recommend a weight of 1, i.e., $W_7=1$.

To ensure external consistency one must create and maintain full cross-references between all requirements and relevant statements made in other documents (see Section 2.22). However, maintaining external consistency may entail more than this. For example, it might be that a software development plan (SDP) or a development contract states that development effort must consume no more than $1M or last no more than 18 months, but the SRS defines so many requirements that it is impossible to meet cost or schedule. Here the problem is not individual requirements but the combined effect of all requirements. The SRS must be reviewed simultaneously with all possible conflicting documents, including SOW, development contract, and SDP.

2.8 Achievable

An SRS is *achievable* if and only if there could exist at least one system design and implementation that correctly implements all the requirements stated in the SRS. Achievability, Q_8, is a measure of the existence of a single system and thus has a discrete value of 1 or 0, i.e., a set of requirements are either achievable or given acceptable development resources they are not. A weight of $W_8=1$ is appropriate. The best way to ensure achievability is to construct a working prototype of parts of the system where achievability may be in doubt.

2.9 Concise

An SRS is *concise* if it is as short as possible without adversely affecting any other quality of the SRS. Thus if we have two SRS's that describe the identical system, with identical measures of qualities for the 23 other quality attributes, then the shorter one is better.

One way to measure conciseness is to count pages. However, comparative SRS sizes are only important after we are sure they describe identical systems. In general, determining if two SRSs describe identical systems is undecideable. The ultimate in conciseness is the null SRS; this should earn a 1. The worst case of conciseness is an SRS of infinite size; and score zero. One metric that exhibits these properties is the hyperbole:

$$Q_9 = \frac{1}{size + 1}$$

where *size* is the number of pages. An appropriate weight is probably $W_9=.2$.

Major reductions in SRS size are rarely possible without adversely effecting other qualities. The primary exception is when writers are prone to baroque writing, e.g., THE CHECK PRINTING FUNCTION OF THE PAYROLL SYSTEM SHALL PROVIDE THE CAPABILITY TO VALIDATE CHECK AMOUNTS can be shortened to THE PAYROLL SYSTEM SHALL VALIDATE CHECK AMOUNTS and score higher for conciseness *and* understandability.

2.10 Design-Independent

An SRS is *design independent* if and only if there exist more than one system design and implementation that correctly implements all requirements stated in the SRS. The purpose of the SRS is to express desired external

behavior to a degree that user satisfaction is guaranteed and a maximum number of designs exist to satisfy those needs and behaviors. It is okay to describe external behavior of a solution system using an FSM as long as it is clearly stated that the solution system must behave externally the same way that the FSM behaves externally. It is not okay to include an FSM and imply that the solution system must be designed as an FSM.

Let us assume that the requirements in the SRS include some (R_E) that describe pure external behavior, and some (R_I) that directly address architecture or algorithms of the solution (Note that R = R_E U R_I). Then there exists some number of actual solution system designs ($D(R_E \cup R_I)$) that satisfy all requirements, and some number of actual solution system designs ($D(R_E)$) that satisfy only those external behavior requirements. Design independence can be measured as the percentage of possible solution systems that are eliminated by adding the overly constraining requirements:

$$Q_{10} = \frac{D(R_E \cup R_I)}{D(R_E)}.$$

Values range from 0 (highly design dependent) to 1 (design independent). Projects can still succeed with poor design independence, but their success becomes hampered. For that reason, we give it weight W_{10}=.5.

One effective technique to ensure design independence is to have designers review the SRS. In general, designers take pride in their ability to synthesize an optimal design. Therefore they are likely to have a tremendous ego investment in finding ways to reduce design dependence of the SRS.

2.11 Traceable

An SRS is *traceable* if and only if it is written in a manner that facilitates the referencing of each individual requirement [DAV93]. During design and test it is essential to know which requirements are being supported by the component or verified by the test. Without this, it is impossible to design or test in a quality manner.

An SRS is either traceable or it is not. An SRS could contain some traceable requirements and some not traceable. However, this should render the entire document untraceable. Traceability, Q_{11}, earns a score of 1 if it exhibits any of the qualities described below, or 0 if it does not. There are a variety of effective techniques for achieving traceability [DAV93]:

- Number every paragraph hierarchically. You can later refer to any by a paragraph and sentence number, e.g., requirement 2.3.2.4s3 refers to the requirement in sentence 3 of paragraph 2.3.2.4.
- Number every paragraph hierarchically and include only one requirement in any paragraph. You can refer to any by a paragraph number.
- Number every requirement with a unique number in parentheses immediately after the requirement.
- Use a convention for indicating a requirement, e.g., always use the word *shall* in a sentence containing a requirement; then use a simple *shall*-extraction tool to extract and number all sentences with *shall*.

2.12 Modifiable

An SRS is *modifiable* if its structure and style are such that any changes can be made easily, completely, and consistently [IEE84]. There are two primary reasons for modifiability: (1) needs always evolve, and (2) the SRS, like all complex software-related documents, will contain errors. As needs evolve, the SRS will be modified to capture new, record changes to old, or delete obsolete requirements. Obviously, modifiability is enhanced if the SRS is also traceable (see Section 2.11), in machine readable form (see Section 2.13), traced (see Section 2.22),organized (see Section 2.23), and cross-referenced (see Section 2.24). Modifiability is also enhanced if it includes a table of contents and index.

Since most factors are already included in other metrics, we will measure modifiability, Q_{12}, here as: 1 if table of contents and index are present and 0 otherwise. Its weight is highly dependent on the application.

We know that inherent modifiability of a *program* is related to the degrees of cohesion exhibited by its components and coupling existent between components [YOU79]. As defined by Yourdon and Constantine, these measures make no sense for requirements, but perhaps similar measures can be developed for SRSs so we can measure cohesion of an SRS section or degree of interrelatedness between two SRS sections.

2.13 Electronically Stored

An SRS is *electronically stored* if and only if the entire SRS is in a word processor, it has been generated from a requirements database or has been otherwise synthesized from some other form. Usually, an SRS is either stored electronically or it is not. However, one could measure the percentage of the volume of the SRS that has been electronically stored and call it Q_{13}. Its weight is application dependent.

2.14 Executable/Interpretable/Prototypable

An SRS is *executable*, *interpretable*, or *prototypable* if and only if there exists a software tool capable of inputting the SRS and providing a dynamic behavioral model. This might be achieved by the SRS being written in a language that (1) is directly understood by a computer, or (2) is translatable into a language directly

understood by a computer, or (3) can be interpreted by a software tool and thus simulated.

SRSs may be partially written in an executable, interpretable, or prototypable language. Therefore the metric Q_{14} ranges from 0 (entirely not executable) to 1 (entirely executable). Its weight is highly dependent on the application.

The best technique to ensure executability is to use any commercially available tool that provides such execution of requirements: PAISLey [ZAV86], RAPID [WAS86], and STATEMATE [HAR88]. Dozens of DFD-based CASE tools claim to also provide such executability, but execution is limited to DFDs where behavior of each bubble is defined using some behavioral model (e.g., FSMs, DTs), and the semantics of the DFDs are augmented with execution precedence rules. These CASE tools can also be used but power, versatility, and requirements orientation of behavioral models are superior to DFD-based specifications.

2.15 Annotated by Relative Importance

An SRS is *annotated by relative importance* if a reader can easily determine which requirements are of most importance to customers, which are next most important, etc. This is needed to allocate dollars sensibly, and determine priorities when budgets are inadequate.

Typically, an SRS is either annotated by relative importance or not. Obviously we can calculate the percentage of requirements that are annotated and use that as a measure, Q_{15}. Its weight is application dependent.

One way to achieve this is to suffix every requirement with (M), (D) and (O) to denote that this requirement is mandatory, desirable, or optional.

2.16 Annotated by Relative Stability

An SRS is *annotated by relative stability* if a reader can easily determine which requirements are of most likely to change, which are next most likely, etc. Designers need this to help determine where to build in flexibility. Knowing the relative stability can help a team decide whether or not to build in that flexibility.

Typically, an SRS is either annotated by relative stability or not. Obviously we can calculate the percentage of requirements that are annotated and use that as a measure, Q_{16}. Its weight is application dependent.

One way to achieve this is to suffix every requirement with (H), (M) and (L) to denote whether the probability of change is high, medium, or low.

2.17 Annotated by Version

An SRS is *annotated by version* if a reader can easily determine which requirements will be satisfied in which versions of the product. Both customers and designers obviously need to know this.

Like the previous two annotations, and SRS is either annotated by version or not. The percentage of requirements annotated by version is a reasonable measure, Q_{17}. It is assumed that an SRS written for just one version of the software is fully annotated (by default) and thus scores a 1. Its weight is application dependent.

The most common way of annotating requirements by version is to add a column in the margin for each version of software to be produced. "X"s are placed beside each requirement in the respective columns.

2.18 Not Redundant

An SRS is *redundant* if the same requirement is stated more than once. Unlike the other 23 attributes, redundancy is not necessarily bad. Often redundancy can be used to increase readability of the SRS significantly. The only problem that redundancy causes is when an SRS is revised. If all occurrences of a redundant requirement are not changed then the SRS becomes inconsistent.

If we count the actual functions (n_f) specified, and the actual unique functions (n_u) specified, then a measure of nonredundancy in an SRS is the percentage of unique functions that are not repeated, i.e.,

$$Q_{18} = \frac{n_f}{n_u}.$$

Values range from 0 (completely redundant) to 1 (no redundancy). Weight will usually be 0.

Since redundancy is not necessarily bad, no technique should be applied specifically. There are techniques that reduce the risks involved in using redundancy. These include incorporation of an index and cross references among any redundant requirements.

2.19 At Right Level of Abstraction/Detail

Requirements can be stated at many levels of abstraction. These examples of requirements range from most abstract to most detailed, but all are in the requirements domain:

a. SYSTEM SHALL PROVIDE COMMUNICATIONS.

b. SYSTEM SHALL PROVIDE VOICE COMMUNICATIONS.

c. TELEPHONE SYSTEM SHALL PROVIDE VOICE COMMUNICATIONS.

d. TELEPHONE SYSTEM SHALL PROVIDE LOCAL CALLS, CALL FORWARDING, LONG DISTANCE CALLS....

e. TELEPHONE SHALL PROVIDE LONG DISTANCE CALLS WHERE USER LIFTS IDLE PHONE, DIAL TONE IS HEARD WITHIN 3 SECONDS, USER DIALS NINE, DISTINCTIVE DIAL TONE IS HEARD WITHIN 2 SECONDS, ETC.

The right level of detail is a function of how the SRS is being used. Generally, the SRS should be specific enough that any system built that satisfies the requirements in the SRS satisfies all user needs, and abstract enough that all systems that satisfy all user needs also satisfy all requirements. Thus, an SRS being used for a contract between customer and developer should be relatively specific to ensure the customer knows what is being acquired, and there are a minimum of surprises.

We can certainly develop ways to subjectively measure the abstraction level of an SRS. All we do is assign a number to each of the five above examples, examine a requirement, decide which example it is closest to, and assign it that value. The level of abstraction of the SRS, Q_{19}, is then the average of the values of each of its constituent requirements. The problem with this is that our goal is not to measure the *level* of abstraction of the SRS but to measure the *appropriateness* of the level of abstraction. This is highly scenario dependent and cannot be measured.

2.20 Precise

An SRS is *precise* if and only if (a) numeric quantities are used whenever possible, and (b) the appropriate levels of precision are used for all numeric quantities. Thus, THE SYSTEM SHALL EXHIBIT FAST RESPONSE TIME is not precise as THE SYSTEM SHALL FULLY RESPOND TO EVERY REQUEST WITHIN 2 SECONDS. Also, THE SYSTEM SHALL DISPLAY THE WAIT TIMES is not as good as THE SYSTEM SHALL DISPLAY THE WAIT TIMES TO THE NEAREST TENTH OF A SECOND. Also, assuming that the nearest tenth of a second is all that is needed, this requirement exhibits inappropriate levels of precision: THE SYSTEM SHALL DISPLAY THE WAIT TIMES TO THE NEAREST NANOSECOND.

2.21 Reusable

An SRS is *reusable* if and only if its sentences, paragraphs and sections can be easily adopted or adapted for use in a subsequent SRS. Much research is underway concerning reuse of design and code. Little extends to the requirements domain.

Ideally, reusability should be measured on results rather than potential. Thus a score of 1 should be given to an SRS whose contents have been fully reused by later SRSs and 0 to an SRS none of whose contents have been. Unfortunately, metrics are more useful if they can be established at the time of SRS creation rather than many years later. An alternative is to measure SRS reusability as the potential for SRS reuse. In the case of reuse of design and code, research results have helped us recognize what makes a component reusable, although results are not consistent or conclusive. In the case of requirements reuse, no research results are available. The next paragraph will introduce some experimental requirements reuse properties. When more information becomes known, a reusability metric of "percentage of paragraphs that exhibit reuse properties" can be used. However, it will have the same problems as for design and code. Given a software system, the percentage of components that are data abstractions (or have any of the many other qualities that increase potential reuse) does not yield a reasonable reusability metric because there exist data abstractions that are not reusable.

Little is known about techniques to optimize potential reuse of requirements specifications. Here are some avenues from the most to the least understood:

1. Write SRS sections using "symbolic constants," e.g., in the performance section, use a word processor symbolic constant for key response times. Then, later applications with similar functionality but with different response times can simply change the value in the symbolic constant.
2. Use formal models. The specific FSMs, DTs, PNs, and statecharts are unlikely to be reusable, but their presence will likely cause the next SRS writer to reuse the concept of employing such models.
3. Create library of *abstract requirements types*. These are generic requirements paragraphs that are instantiated by providing tailoring information about characteristics of a particular application. The actual SRS becomes a series of instantiations.

2.22 Traced

An SRS is *traced* if and only if the origin of each of its requirements is clear [DAV93]. This implies that every requirement that has a basis is cross-referenced to that basis. Typical bases include: system-level RSs, system-level design documents, hardware RSs, SOWs, contracts, white papers/research reports, and SDPs. Any of these documents may hold a clue as to the reason why a particular requirement exists. For example, a requirement THE SYSTEM SHALL REPORT THE CURRENT POSITION OF ALL SHIPS NO LESS OFTEN THAN EVERY SECOND may exist because an earlier white paper reported the maximum possible ship speed, and an earlier system-level RS reported the resolution and scale of the display medium. In this case, the above requirement should be cross-referenced to both of the earlier documents.

Measuring the level of traced-ness is impossible. Ideally, we want to measure the following:

$$\frac{\text{Number of requirements traced to their origins}}{\text{Number of requirements that have origins}}$$

Unfortunately, the only way to measure the denominator is to examine the SRS for such cross-references! Thus, the above fraction will always have a value of 1.

There are two techniques to record traces. First is to include explicit cross-references in parentheses following each requirement in the SRS. Second is to record all requirements in a database. Each requirement is a record. A field is used for cross-references. The SRS becomes a retrieval of the database and might include or exclude the cross-references.

2.23 Organized

An SRS is *organized* if and only if its contents are arranged so that readers can easily locate information and logical relationships among adjacent sections is apparent. One way is to follow any of the many SRS standards [DOR90]. Certainly boiler plate sections of all SRS standards are roughly equivalent. Primary differences concern organization of detailed requirements. There are many ways to organize these. However, given any particular system, there are probably only a few right ways. Some alternatives are [DAV93] (in all cases assume that detailed requirements are in Section 3):

1. group the functional requirements by *class of user*. For example, an elevator control system SRS might include a Section 3.1 for all passenger requirements, 3.2 for all fireperson requirements, and 3.3 for all maintenance person requirements.
2. group the functional requirements by *common stimulus*. For example, an automated helicopter landing system SRS might include a Section 3.1 for all requirements relating to gusts of wind, 3.2 for all relating to being out of fuel, 3.3 for all relating to breakage of landing gear, etc.
3. group the functional requirements by *common response*. For example, a payroll system SRS might include a Section 3.1 for all requirements relating to generation of paychecks, 3.2 for all relating to generation of a report of all current employees with their monthly salaries, etc.
4. group the functional requirements by *feature*. For example, a payroll system SRS might include a Section 3.1 for all requirements relating to local calls, 3.2 for all relating to long distance calls, 3.3 for all relating to conference calls, etc.
5. group the functional requirements by *object*. For example, an airline reservation system SRS might include a Section 3.1 for all requirements relating to seats, 3.2 for all relating to flight segments, 3.3 for all relating to travel agents, 3.4 for all relating to tickets, etc.

Organization is purely subjective; we do not believe it can be measured.

To achieve useful organization, (1) follow a standard, and (2) use one of the above five organizational models which renders the SRS most easily understood.

2.24 Cross-Referenced

An SRS is *cross referenced* if and only if cross-references are used in the SRS to relate sections containing requirements to other sections containing:

- identical (i.e., redundant) requirements
- more abstract or more detailed descriptions of the same requirements
- requirements that depend on them or on which they depend (see related discussion of coupling in Section 2.12).

Any well-written SRS will describe requirements at a variety of levels, usually from the most abstract to the most detailed. To increase understandability many SRSs include redundancy. All SRSs will include requirements with interdependency. Thus, all SRSs should include cross-references. Like traced (see Section 2.22), there is no way to determine how many cross references are appropriate in an SRS. For this reason, any attempt to measure cross-references is fallacious.

The same techniques that work for traced (see Section 2.22) work for cross-references. Either use explicit in-text cross-references or preferably store all requirements in a database and use specific fields to store the three above types of cross references.

III. SRS Quality: A Compromise

A perfect SRS is impossible. For example, if we remove all ambiguity, we will add so much formality that it would no longer be understandable by a non-computer expert. If we remove all redundancy, it becomes difficult to read. If we go overboard with completeness, we lose conciseness. There are some qualities for which we can strive without adversely affecting others: correct, internally consistent, externally consistent, achievable, design-independent, organized, traced, traceable, all annotations, electronically stored and cross-referenced.

There may be some value in an overall rating of the quality of an SRS. The presence of some of SRS qualities appears to be essential for all applications and thus have been given weights of 1. Others seem less important in general and have lower weights. The actual weights for all the attributes must be assigned by each project to be meaningful. To have an overall quality on a scale from 0 to 1, we have:

$$Q = \frac{\sum_{i=1}^{18} W_i Q_i}{\sum_{i=1}^{18} W_i}$$

The above equation is a gross simplification; it is useful primarily for people who need to see just one number that states the quality of an SRS. More meaningful are the values for the entire vector Q_i.

IV. Summary

In summary, this paper defined 24 qualities that SRSs should exhibit. In 18 cases, it has provided a metric. It is hoped that in the future, we (or others) will be able to more fully expand the list of qualities, and will provide more complete measures for all qualities.

Acknowledgments

The authors thank P. Aiken, F. Armour, C. Gorski, S. Paey, S. Park, M. Ricker, G. Santacruz, and R. Sonnemann for many hours of stimulating discussion on the subject of this paper, J. Berdon and T. Nakajima for helpful feedback on the paper's contents, and K. Baugh for typing it.

References

[ALE90] Alexander, L., *Selection Criteria for Alternative Software Life-Cycle Process Models*, Software Engineering M. S. Thesis, Fairfax, Virginia: George Mason University, 1990.

[ALF76] Alford, M., and I. Burns, "R-nets: A Graph Model for Real-Time Software Requirements," *Symposium on Computer Software Engineering*, New York: Polytechnic Press, 1976, pp. 97-108.

[ALF85] Alford, M., "SREM at the Age of Eight: *The Distributed Computing Design System.*" *IEEE Computer, 18*, 4 (April 1989), pp. 36-46.

[AND89] Andriole, S., *Storyboard Prototyping*, Wellesley, Massachusetts: QED, 1989.

[BAS81] Basili, V., and D. Weiss, "Evaluation of a Software Requirements Document by Analysis of Change Data," *Fifth IEEE Int'l Conf. on Software Eng.*, 1981, pp. 314-323.

[BOE75] Boehm, B., et al., "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software," *IEEE Trans. Software Eng., 1*, 1 (March 1975), pp. 125-133.

[BOE76] Boehm, B., "Software Engineering," *IEEE Trans. Computers, 25*, 12 (December 1976), pp. 1226-1241.

[CAR90] Caruso, J., private communication, Fairfax, Virginia, Fall 1990.

[CEL83] Celko, J., et al., "A Demonstration of Three Requirements Language Systems," *ACM SIGPLAN Notices, 18*, 1 (January 1983), pp. 9-14.

[DAL77] Daly, E., "Management of Software Development," *IEEE Trans. Computers, 3*, 3 (May 1977), pp. 229-242.

[DAV79] Davis, A., and T. Rauscher, "Formal Techniques and Automated Processing to Ensure Correctness in Requirements Specifications," *IEEE Specifications of Reliable Software Conf.*, 1979, pp. 15-35.

[DAV79a] Davis, A., et al., "RLP: An Automated Tool for the Processing of Requirements." *IEEE COMPSAC '79*, 1979.

[DAV90] Davis, A., *Software Requirements: Analysis and Specification*, Englewood Cliffs, New Jersey: Prentice Hall, 1990.

[DAV90a] Davis, A., "System Testing: Implications of Requirements Specifications," *Information and Software Technology, 32*, 6 (July/August 1990), pp. 407-414.

[DAV92] Davis, A., "Operational Prototyping: A New Development Approach," *IEEE Software, 9*, 5 (September 1992), pp. 70-78.

[DAV93] Davis, A., *Software Requirements: Objects, Functions, and States* (Second Edition), Englewood Cliffs, New Jersey: Prentice Hall, 1993.

[DOD88] Department of Defense, *Military Standard: Defense System Software Development*, DOD-STD-2167A, Washington, D.C., 1988.

[DOR90] Dorfman, M., and R. Thayer, *Standards, Guidelines, and Examples on System and Software Requirements Engineering*, Washington D.C.: IEEE Computer Society Press, 1990.

[ESA87] European Space Agency, *ESA Software Engineering Standards*, Noordwijk, Netherlands: ESA Publications Division, 1987.

[FAG74] Fagan, M., *Design and Code Inspections and Process Control in the Development of Programs*, IBM Report IBM-SDD-TR-21-572, December 1974.

[HAR88] Harel, D., et al., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems." *IEEE Tenth Int'l Conf. on Software Eng.*, 1988.

[IEE84] Institute for Electrical and Electronics Engineers, *IEEE Guide to Software Requirements Specifications*, Standard 830-1984, New York: IEEE Computer Society Press, 1984.

[JAF91] Jaffe, M., et al., "Software Requirements Analysis for Real-Time Process-Control Systems," *IEEE Trans. Software Engineering, 17*, 3 (March 1991), pp. 241-258.

[JPL88] Jet Propulsion Laboratory, *Software Requirements Analysis Phase*, JPL-D-4005, Pasadena, California, 1988.

[MIZ83] Mizuno, Y., "Software Quality Improvement," *IEEE Computer, 15*, 3 (March 1983), pp. 66-72.

[NCC87] National Computer Centre Ltd., *The STARTS Guide*, Manchester, England, 1987.

[ROM90] Rombach, H., *Software Specifications: A Framework*, Curriculum Module SEI-CM-11-2.1, Pittsburgh, Pennsylvania: Software Engineering Institute, January 1990.

[TAV84] Tavolato, P., and K. Vincena, "A Prototyping Methodology and its Tool," in *Approaches to Prototyping*, R. Budde, ed., Berlin: Springer-Verlag, 1984, pp. 434-446.

[WAS86] Wasserman, A., et al., "Developing Interactive Information System with the User Software Engineering Methodology." *IEEE Trans. Software Eng., 12*, 2 (February 1986), pp. 325-345.

[WEB86] *Webster's Third New International Dictionary*, Springfield, Massachusetts: Merriam-Webster, Inc., 1986.

[YOU79] Yourdon, E., and L. Constantine, *Structured Design*, Englewood Cliffs, New Jersey: Prentice-Hall, 1979.

[ZAV81] Zave, P., and R. Yeh, "Executable Requirements for Embedded Systems," *Fifth IEEE Int'l Conf. on Software Eng.*, 1981, pp. 295-304.