SOFTWARE REQUIREMENTS: A TUTORIAL

"The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements...No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later." [Brooks 87]

1. Introduction

Deciding precisely what to build and documenting the results is the goal of the requirements phase of software development. Experience suggests that requirements are the biggest software engineering problem for developers of large, complex systems. While there is considerable disagreement on how to solve the problem, few would disagree with Dr. Brooks' assessment that no other part of a development is as difficult to do well or as disastrous in result when done poorly. The purpose of this tutorial is to help the reader understand why the conceptually simple notion of "deciding what to build" is so difficult in practice, where the state of the art does and does not address these difficulties, and what hope we have for doing better in the future.

This paper does not survey the literature but seeks to provide the reader with an understanding of the underlying issues. There are currently many more approaches to requirements than one can cover in a short paper. This diversity is the product of different views about which of the many problems in requirements is pivotal and different assumptions about the desirable characteristics of a solution. This paper attempts to impart a basic understanding of the many facets of the requirements problem and the tradeoffs involved in attempting a solution. Thus forearmed, the reader may make his own assessment of the claims of different requirements methods and their likely effectiveness in addressing his particular needs.

We begin with basic terminology and some historical data on the requirements problem. We examine the goals of the requirements phase and the problems that can arise in attempting those goals. As in Brooks' article, much of the discussion is motivated by the distinction between the difficulties inherent in what one is trying to accomplish ("essential" difficulties) and those one creates through inadequate practice ("accidental" difficulties). We discuss how a disciplined software engineering process helps address the accidental difficulties and why the focus of a disciplined process is on producing a written specification of the detailed technical requirements. We examine current approaches to requirements in terms of the specific problems each approach seeks to address. Finally, we examine technical trends, including recent work at the Naval Research Laboratory, and discuss where significant advances are likely to occur in the future.

2. Requirements and the Software Life Cycle

A variety of software life-cycle models have been proposed with an equal variety of terminology. Davis [Davis 88] provides a good summary. While differing in the detailed decomposition

of the steps (e.g., prototyping models) or in the surrounding management and control structure (e.g., to manage risk), there is general agreement on the core elements of the model. Figure 1 from [Davis 93] gives a version of the common model that illustrates the relationship between the software development stages and the related testing and acceptance phases.

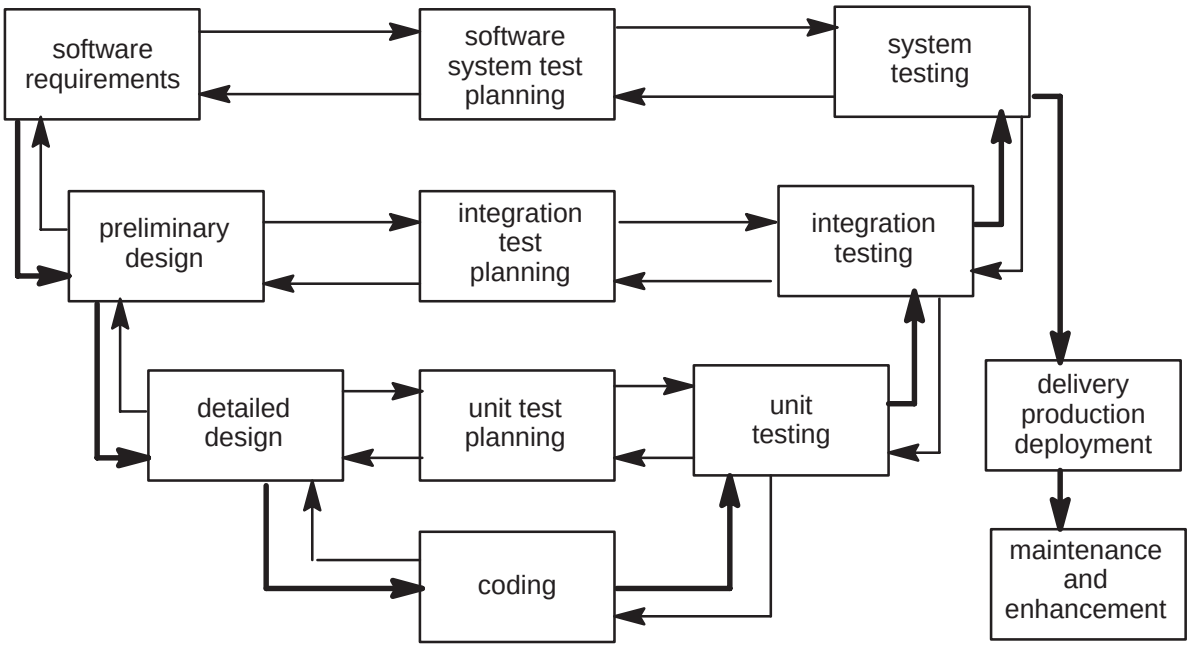

Figure 1. Software Life Cycle

When software is created in the context of a larger hardware and software system, system requirements are defined first followed by system design. System design includes decisions about which parts of the system requirements will be allocated to hardware and which to software. For software-only systems, the life cycle model begins with analysis of the software requirements. From this point on, the software development model is the same whether or not the software is part of a larger system, as shown in Figure 2 [Davis 93]. For this reason, the remainder of our discussion does not distinguish whether or not software is developed as part of a larger system. For an overview of system versus software issues, the reader is referred to Dorfman and Thayer's survey [Thayer 90].

In a large system development, the software requirements specification may play a variety of roles. For example, the requirements typically document what the customer expects to be delivered and may provide the contractual basis for the development. For managers it may provide the basis for scheduling a yardstick for progress. To the system engineers, it defines the role of software within the overall system and its relation to the hardware components.. For the software designers, it specifies the constraints on behavior and performance. For coders it defines the range of acceptable system behavior and is the final authority on the outputs produced. For testers, it is the basis for test planning and verification. For some systems the requirements are also used by such diverse groups as marketing and governmental regulators.

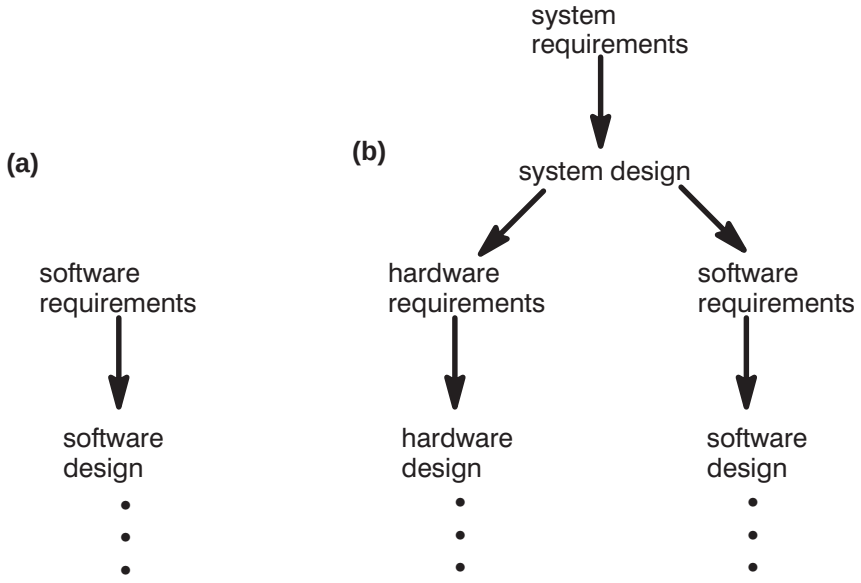

Figure 2. Development paths (a) software. (b) systems

It is common practice to classify software requirements as "functional" or "non–functional." While definitions vary, "functional" typically refers to requirements constraining the visible behavior of the system, specifically the required values of the outputs for given inputs. "Non–functional" typically refers to all other constraints including, but not limited to, performance, dependability, maintainability, reusability, and safety requirements.

Neither the terminology nor the classification into functional and non-functional is effective. The word "function" is one of the most overloaded in computer science and its only rigorous meaning, that of a mathematical function, is not what is meant here. The classification is not terribly useful because it partitions classes of requirements with markedly similar qualities (e.g., timing of outputs from output values) while grouping others that have nothing in common except what they are not (e.g., political issues and fault tolerance constraints).

A more useful distinction is between what can be described as "behavioral requirements" and "developmental quality attributes" with the following definitions:

- *Behavioral requirements* – Behavioral requirements define the allowed, observable behavior of the system in terms of the outputs required for given inputs and system state. The concept of state includes time, the software state, and the hardware state so security, correctness, performance, and fault-tolerance are behavioral requirements.
- *Developmental quality attributes* – Developmental quality attributes constrain the qualities of the system's static construction. These include properties like testability, maintainability, changeability, maintainability, and reusability.

Behavioral requirements have in common that they are properties of the run-time behavior of the system. Behavioral requirements can be defined in terms of inputs, outputs, and state, and objectively validated by changing those quantities and observing the system behavior. Developmental quality attributes have in common that they are properties of how the system is created

and its static representation. Assessments of quality attributes are relativistic in the sense that there is no common or absolute scale against which their achievement can be measured (e.g., one can talk about one design being more maintainable than another but not about its being maximally maintainable).

While there is a large body of literature in both the academic and development communities on the modeling, specification, and analysis of behavioral requirements, there is relatively little significant work on the rigorous specification of developmental attributes. Since the rigorous specification of developmental quality attributes remains a relatively unexplored area of requirements, the following discussion necessarily focuses on behavioral requirements and on techniques for developing specifications with the desired quality attributes rather than on techniques for specifying those attributes.

3. A Big Problem

Requirements problems are persistent, pervasive, and costly. Evidence is most readily available for the large software systems developed for the U.S. Government since the results are a matter of public record. As soon as software became a significant part of such systems, developers identified requirements as a major source of problems. For example, developers of the early Ballistic Missile Defense System noted that:

> In nearly every software project that fails to meet performance and cost goals, requirements inadequacies play a major and expensive role in project failure [Alford 79].

Nor has the problem mitigated over the intervening years. A recent study of problems in mission critical defense systems identified requirements as a major problem source in two thirds of the systems examined [GAO 92]. Likewise, studies by Lutz [Lutz 92] identified functional and interface requirements as the major source of safety related software errors in NASA's Voyager and Galileo spacecraft.

Results of industry studies in the 1970's described by Boehm [Boehm 81], and since replicated a number of times, showed that requirements errors are the most costly. These studies all produced the same basic result: the earlier in the development process an error occurs and the later the error is detected, the more expensive it is to correct. Moreover, the relative cost rises quickly. As shown in Figure 3, an error that costs a dollar to fix in the requirements phase may cost 100 to 200 times as much to fix if it is not corrected until the system is fielded or in the maintenance phase.

The costs of such failures can be enormous. For example, the 1992 GAO report notes that one system, the Cheyenne Mountain Upgrade, will be delivered eight years late, exceed budget by $600 million, and have less capability than originally planned. Prior GAO reports [GAO 79] suggest that such problems are the norm rather than the exception. While data from private industry is less readily available, there is little reason to believe that the situation is significantly different.

Stage	Relative Repair Cost
Requirements	1 – 2
Design	5
Coding	10
Unit test	20
System test	50
Maintenance	200

Figure 3. Relative cost to repair a software error in different stages

In spite of presumed advances in software engineering methodology and tool support, the requirements problem has not diminished. This does not mean that the apparent progress in software engineering is illusory. While the features of the problem have not changed, the applications have grown significantly in capability, scale, and complexity. A reasonable conclusion is that the growing ambitiousness of our software systems has outpaced the gains in requirements technology, at least as such technology is applied in practice.

4. Why are Requirements Hard?

It is generally agreed that the goal of the requirements phase is to establish and specify precisely what the software must do without describing how to do it. So simple seems this basic intent that it is not at all evident why it is so difficult to accomplish in practice. If what we want to accomplish is so clear, why is it so hard? To understand this, we must examine more closely the goals of the requirements phase, where errors originate, and why the nature of the task leads to some inherent difficulties.

Most authors agree in principle that requirements should specify "what" rather than "how." In other words, the goal of requirements is to understand and specify the *problem* to be solved rather than the *solution*. For example, the requirements for an automated teller system should talk about customer accounts, deposits, and withdrawals rather than the software algorithms and data structures. The most basic reason for this is that a specification in terms of the problem captures the actual requirements and does not over constrain the subsequent design or implementation. Further, solutions are typically more complex, more difficult to change, and harder to understand (particularly for the customer) than a specification of the problem.

Unfortunately, distinguishing "what" from "how" itself represents a dilemma. As Davis [Davis 88], among others, points out, the distinction between what and how is necessarily a function of perspective. A specification at any level of abstraction in a system can be viewed as describing the "what" for the next level. Thus customer needs may define the "what" and the system decomposition into hardware and software the corresponding "how". The system decomposition defines the "what" the behavior of a software component, the "how," and so on. The upshot is that requirements cannot be effectively discussed at all without prior agreement on the which system one is talking about and at what level of abstraction. One must agree on what constitutes the *problem space* and what constitutes the *solution space* – the analysis and specification of requirements then properly belongs in the problem space.

In discussing requirements problems one must also distinguish the development of large, complex systems from smaller efforts (e.g., developments by a single or small team of programmers). Large system developments are multi-person efforts. They are developed by teams of tens to hundreds of programmers. The programmers work in the context of an organization typically including management, systems engineering, marketing, accounting, and quality assurance. The organization itself must operate in the context of outside concerns also interested in the software product, including the customer, regulatory agencies, and suppliers.

Even where only one system is intended, large systems are inevitably multi-version as well. As the software is being developed, tested, and even fielded, it evolves. Customers understand better what they want, developers understand better what they can and cannot do within the constraints of cost and schedule, and circumstances surrounding development change. The results are changes in the software requirements and, ultimately, the software itself. In effect, several versions of a given program are produced if only incrementally. Such unplanned changes occur in addition to the expected variations of planned improvements.

The multi-person, multi-version nature of large system development introduces problems that are both quantitatively and qualitatively different from those found in smaller developments. For example, scale introduces the need for administration and control functions with the attendant management issues that do not exist on small projects. The quantitative effects of increased complexity in communication when the number of workers rises are well documented by Brooks [Brooks 75]. In the following discussion, it is this large system development context we will assume since that is the one in which the worst problems occur and where the most help is needed.

Given the context of multi-person, multi-version development, our basic goal of specifying what the software must do can be decomposed into the following subgoals:

1. Understand precisely what is required of the software.
2. Communicate the understanding of what is required to all of the parties involved in the development.
3. Provide a means for controlling the production to ensure that the final system satisfies the requirements (including managing the effects of changes).

It follows that the source of most requirements errors is in the failure to adequately accomplish one of these goals, i.e.:

1. The developers failed to understand what was required of the software by the customer, end user, or other parties with a stake in the final product.
2. The developers did not completely and precisely capture the requirements or subsequently communicate the requirements effectively to other parties involved in the development.
3. The developers did not effectively manage the effects of changing requirements or ensure the conformance of down-stream development steps including design, code, integration, test, or maintenance to the system requirements.

The end result of such failures is a software system that does not perform as desired or expected, a development that exceeds budget and schedule or, all too frequently, failure to deliver any working software at all.

4.1 Essential Difficulties

Even our more detailed goals appear reasonably straight forward, why then do so many development efforts fail to achieve them? The short answer is that the *mutual* satisfaction of these goals, in practice, is inherently difficult. To understand why, it is useful to reflect on some points raised by Dr. Brooks [Brooks 87] on why software engineering is hard and on the distinction he makes between essential difficulties – those inherent in the problem, and the accidental difficulties – those introduced through imperfect practice. For though requirements are inherently difficult, there is no doubt that these difficulties are many times multiplied by the inadequacies of current practice. The following essential difficulties attend each of the requirements goals:

- *Comprehension.* People do not know what they want. This does not mean that people do not have a general idea of what the software is for. Rather, they do not begin with a precise and detailed understanding of what functions belong in the software, what the output must be for every possible input, how long each operation should take, how one decision will affect another, and so on. Indeed, unless the new system is simply a reconstruction of an old one, such a detailed understanding at the outset is unachievable. Many decisions about the system behavior will depend on other decisions yet unmade and expectations will change as the problem (and attendant costs of alternative solutions) is better understood. Nonetheless, it is this precise and richly detailed understanding of expected behavior that is needed to create effective designs and develop correct code.

- *Communication.* Software requirements are difficult to communicate effectively. As Dr. Brooks points out, the conceptual structures of software systems are complex, arbitrary, and difficult to visualize. The large software systems we are now building are among the most complex man-made artifacts ever attempted. That complexity is arbitrary in the sense that it is an artifact of people's decisions and prior construction rather than a reflection of fundamental properties (as, for example, in the case of physical laws). To make matters worse, most of the conceptual structures in software have no readily comprehensible physical analogue so they are difficult to visualize.

  In practice, comprehension suffers under all of these constraints. We work best with regular, predictable structures, can comprehend only a limited amount of information at one time, and understand large amounts of information best when we can visualize it. Thus the task of capturing and conveying software requirements is inherently difficult.

  The inherent difficulty of communication is compounded by the diversity of purposes and audiences for a requirements specification. Ideally a technical specification is written for a particular audience. The brevity and comprehensibility of the document depend on assumptions about common technical background and use of language. Such commonality typically does not hold for the many diverse groups (e.g., customers, systems engineers, managers) that must use a software requirements specification.

- *Control.* Inherent difficulties attend control of software development as well. The arbitrary and invisible nature of software makes it difficult to anticipate which requirements will be met easily and which will decimate the project's budget and schedule if they can be fulfilled at all. The low fidelity of software planning has become a cliche yet the requirements are the only available basis for planning or for tracking to a plan.

  This situation is made incalculably worse by by software's inherent malleability. Of all the problems bedeviling software mangers, few evoke such passion as the difficulties of dealing with frequent and arbitrary changes to requirements. For most systems, such changes remain a fact of life even after delivery. The continuous changes make it difficult to develop stable specifications, plan effectively, or control cost and schedule. For many industrial developers, change management is the most critical problem in requirements.

- *Inseparable concerns.* In seeking solutions to the forgoing problems, we are faced with the additional difficulty that the issues cannot easily be separated and dealt with piecemeal. For example, developers have attempted to address the problem of changing requirements by baselining and freezing requirements before design begins. This proves impractical because of the comprehension problem – the customer will not fully know what he wants until he sees it. Similarly, the diversity of purposes and audiences is often addressed by writing a different specification for each. Thus there may be a system specification, a set of requirements delivered to customer, a distinct set of technical requirements written for the internal consumption of the software developers, and so on. However this solution vastly increases the complexity, provides an open avenue for inconsistencies, and multiplies the difficulties of managing changes.

  These issues represent only a sample of the inherent relationships between different facets of the requirements problem. The many distinct parties with an interest in a system's requirements, the many different roles the requirements play, and the interlocking nature of software's conceptual structures all introduce dependencies between concerns and impose conflicting constraints on any potential solution.

  The implications are twofold. First we are constrained in the application of our most effective strategy for dealing with complex problems – divide and conquer. If a problem is considered in isolation, the solution is likely to aggravate other difficulties. Effective solutions to most requirements difficulties must simultaneously address more than one problem. Second, developing practical solutions requires making difficult tradeoffs. Where different problems have conflicting constraints, compromises must be made. Because the tradeoffs result in different gains or losses to the different parties involved, effective compromises require negotiation. These issues are considered in more detail when we discuss the properties of a good requirements specification.

4.2 Accidental Difficulties

While there is no doubt that software requirements are inherently difficult to do well, there is equally no doubt that common practice unnecessarily exacerbates the difficulty. We use the term "accidental" in contrast to "essential," not to imply that the difficulties arise by chance but that

they are the product of common failings in management, elicitation, specification, or use of requirements. It is these failings that are most easily addressed by improved practice.

- *Written as an afterthought.* It remains common practice that requirements documentation is developed only after the software has been written. For many projects, the temptation to rush into implementation before the requirements are adequately understood proves irresistible. This is understandable. Developers often feel like they are not really doing anything when they are not writing code; managers are concerned about schedule when there is no visible progress on the implementation. Then too, the intangible nature of the product mitigates toward early implementation. Developing the system is an obvious way to understand better what is needed and make visible the actual behavior of the product. The result is that requirements are written as an afterthought (if at all). They are not created to guide the developers and testers but treated as a necessary evil to satisfy contractual demands.

  Such after-the-fact documentation inevitably violates the basic goal of defining what the system must do rather than the how since it is a specification of the code as written. It is produced after the fact so it is not planned or managed as an essential part of the development but is thrown together. In fact, it is not even available in time to guide implementation or manage development.

- *Confused in purpose.* Because there are so many potential audiences for a requirements specification, with different points of view, the exact purpose of the document becomes confused. It is used to sell the product to the customer so it includes marketing hype extoling the product's virtues. It is the only documentation of what the system does so it provides introductory, explanatory, and overview material. It is a contractual document so it is intentionally imprecise to allow the developer latitude in the delivered product or the customer latitude in making no-cost changes. It is the vehicle for communicating decisions about software details to designers and coders so it incorporates design and implementation. The result is a document in which it is unclear which statements represent real requirements and which are more properly allocated to marketing, design, or other documentation. It is a document that attempts to be everything to everyone and ultimately serves no one well.

- *Not designed to be useful.* Often in the rush to implementation little effort is expended on requirements. The requirements specification is not expected to be useful and, indeed, this turns out to be a self-fulfilling prophecy. Because the document is not expected to be useful little effort is expended on designing it, writing it, checking it, or managing its creation and evolution. The most obvious result is poor organization. The specification is written in English prose and follows the author's stream of consciousness or the order of execution.

  The resulting document is ineffective as a technical reference. It is unclear which statements represent actual requirements. It is unclear where to put or find particular requirements. There is no effective procedure for ensuring that the specification is consistent or complete. The is no systematic way to manage requirements changes. The

specification is difficult to use and difficult to maintain. It quickly becomes out of date and loses whatever usefulness it might originally have had.

- *Lacks essential properties.* Lack of forethought, confusion of purpose, or lack of careful design and execution all lead to requirements that lack properties critical to good technical specifications. The requirements, if documented at all, are redundant, inconsistent, incomplete, imprecise, and inaccurate.

Where the essential difficulties are inherent in the problem, the accidental difficulties result from a failure to gain or maintain intellectual control over what is to be built. While the presence of the essential difficulties means that there can be no "silver bullet" that will suddenly render requirements easy, we can remove at least the accidental difficulties through a well thought out, systematic, and disciplined development process. Such a disciplined process provides a stable foundation for attacking the essential difficulties.

5. Role of a Disciplined Approach

The application of discipline in analyzing and specifying software requirements can address the accidental difficulties. While there is now general agreement on the desirable qualities of a software development approach, the field is insufficiently mature to have standardized the development process. Nonetheless, it is useful to examine the characteristics of an idealized process and its products to understand where current approaches weak and which current trends are promising. In general, a complete requirements approach will define:

- *Process:* The (partially ordered) sequence of activities, entrance and exit criteria for each activity, which work product is produced in each activity, and what kind of people should do the work.
- *Products:* The work products to be produced and, for each product, the information needed to produce it, the information it contains, the expected audience, and the acceptance criteria the product must satisfy.

Currently, there is little uniformity in different authors' decomposition of the requirements phase or in the terminology for the activities. Davis [Davis 88] provides a good summary of the variations. Following Davis' integrated model and terminology [Davis 93], the requirements phase consists of two conceptually distinct but overlapping activities corresponding to the first two goals for requirements enumerated above:

1. *Problem analysis:* The goal of problem analysis is to understand precisely what problem is to be solved. It includes identifying the exact purpose of the system, who will use it, the constraints on acceptable solutions, and the possible tradeoffs between conflicting constraints.
2. *Requirements specification:* The goal of requirements specification is to create a document, the Software Requirements Specification (SRS), describing exactly what is to be built. The SRS captures the results of problem analysis and characterizes the set of acceptable solutions to the problem.

In practice, the distinction between these activities is conceptual rather than temporal. Where both are needed, the developer typically switches back and forth between analysis of the problem and documentation of the results. When problems are well understood, the analysis phase may be virtually non-existent. When the system model and documentation are standardized or based on existing specifications, the documentation paradigm may guide the analysis [Hester 81].

5.1 Problem Analysis

Problem analysis is necessarily informal in the sense that there is no effective, closed end procedure that will guarantee success. It is an information acquisition, collating, and structuring process through which one attempts to understand all the various parts of a problem and their relationships. The difficulty in developing effective understanding of large, complex software problems has motivated considerable effort to structure and codify problem analysis.

The basic issues in problem analysis are:

- How to elicit a complete set of requirements from the customer or other sources?
- How to decompose the problem into intellectually manageable pieces?
- How to organize the information so it can be understood?
- How to communicate about the problem with all the parties involved?
- How to resolve conflicting needs?
- How to know when to stop?

5.2 Requirements Specification

For substantial developments, the effectiveness of the requirements effort depends on how well the SRS captures the results of analysis and how useable the specification is. There is little benefit to developing a thorough understanding of the problem if that understanding is not effectively communicated to customers, designers, implementors, testers, and maintainers. The larger and more complex the system, the more important a good specification becomes. This is a direct result of the many roles the SRS plays in a multi-person, multi-version development [Parnas 86]:

1. The SRS is the primary vehicle for agreement between the developer and customer on exactly what is to be built. It is the document reviewed by the customer or his representative and often is the basis for judging fulfillment of contractual obligations.
2. The SRS records the results of problem analysis. It is the basis for determining where the requirements are complete and where additional analysis is necessary. Documenting the results of analysis allows questions about the problem to be asked only once during development.
3. The SRS defines what properties the system must have and the constraints on its design and implementation. It defines where there is, and is not, design freedom. It helps ensure that requirements decisions are made explicitly during the requirements phase, not implicitly during programming.

4. The SRS is the basis for estimating cost and schedule. It is management's primary tool for tracking development progress and ascertaining what remains to be done.
5. The SRS is the basis for test plan development. It is the tester's chief tool for determining the acceptable behavior of the software.
6. The SRS provides the standard definition of expected behavior for the system's maintainers and is used to record engineering changes.

For a disciplined software development, the SRS is the primary technical specification of the software and the primary control document. This is an inevitable result of the complexity of large systems and the need to coordinate multi-person development teams. To ensure that the right system is built one must first understand the problem. To ensure agreement on what is to be built and the criteria for success, the results of that understanding must be written down. The goal of a systematic requirements process is thus the development of a set of requirements that effectively communicate the results of analysis.

Requirements' accidental difficulties are addressed through the careful analysis and specification of a disciplined process. Rather than developing the specification as an afterthought, requirements are understood and specified before development begins. One knows what one is building before attempting to build it. The SRS is the primary vehicle for communicating requirements between the developers, managers, and customers so the document is designed to be useful to that purpose. A useful document is maintained.

6. Requirements for the Software Requirements Specification

The goals of the requirements process, the attendant difficulties, and the role of the requirements specification in a disciplined process determine the properties of a "good" requirements specification. These properties do not mandate any particular method but do describe the characteristics an effective method must address.

In discussing the properties of a good SRS, it useful to distinguish **semantic** properties from **packaging** properties [Faulk 92]. Semantic properties result from *what* the specification says (i.e., its meaning or semantics). Packaging properties result from how the requirements are written down – the format, organization, and presentation of the information. The semantic properties determine how precisely an SRS captures the software requirements. The packaging properties determine how useable the resulting specification is. Figure 4 illustrates the classification of properties of a good SRS:

SRS Semantic Properties	SRS Packaging Properties
Complete	Modifiable
Implementation independent	Readable
Unambiguous and consistent	Organized for reference and review
Precise	
Verifiable	

Figure 4. Classification of SRS properties

An SRS that satisfies the semantic properties of a good specification is:

- *Complete.* The SRS defines the set of acceptable implementations. It should contain all the information needed to write software that is acceptable to the customer and no more. Any implementation that satisfies every statement in the requirements is an acceptable product. Where information is not available before development begins, areas of incompleteness must be explicitly indicated [Parnas 86].
- *Implementation independent.* The SRS should be free of design and implementation decisions unless those decisions reflect actual requirements.
- *Unambiguous and Consistent.* If the SRS is subject to conflicting interpretation, the different parties will not agree on what is to be built or whether the right software has been built. Every requirement should have only one possible interpretation. Similarly, no two statements of required behavior should conflict.
- *Precise.* The SRS should define exactly the required behavior. For each output, it should define the range of acceptable values for every input. The SRS should define any applicable timing constraints such as minimum and maximum acceptable delay.
- *Verifiable.* A requirement is verifiable if it is possible to determine unambiguously whether a given implementation satisfies the requirement or not. For example, a functional requirement is verifiable if it is possible to determine, for any given test case (i.e., an input and an output), whether the output represents an acceptable behavior of the software given the input and the system state.

An SRS that satisfies the packaging properties of a good specification is:

- *Modifiable.* The SRS must be organized for ease of change. Since no organization can be equally easy to change for all possible changes, the requirements analysis process must identify expected changes and the relative likelihood of their occurrence. The specification is then organized to limit the effect of likely changes.
- *Readable.* The SRS must be understandable by the parties that use it. It should clearly relate the elements of the problem space as understood by the customer to the observable behavior (as related by the behavioral requirements) of the software.
- *Organized for reference and review.* The SRS is the primary technical specification of the software requirements. It is the repository for all the decisions made during analysis about what should be built. It is the document reviewed by the customer or his representatives. It is the primary arbitrator of disputes. As such the document must be organized for quick and easy reference. It must be clear where each decision about the requirements belongs. It must be possible to answer specific questions about the requirements quickly and easily.

To address the difficulties associated with writing and using an SRS, a requirements approach must provide techniques addressing both semantic and packaging properties. It is also desirable that the conceptual structures of the approach treat the semantic and packaging properties as distinct concerns (i.e., as independently as possible). This allows one to change the presentation of the SRS without changing its meaning.

In aggregate, these properties of a good SRS represent an ideal. Some of the properties may be unachievable, particularly over the short term. For example, a common complaint is that one cannot develop complete requirements before design begins because the customer does not yet fully understand what he wants or is still making changes. Further, different SRS "requirements" encourage conflicting solutions. A commonly cited example is the use of English prose to express requirements. English is readily understood but notoriously ambiguous and imprecise. Conversely, formal languages are precise and unambiguous but often difficult to read.

Although the ideal SRS may be unachievable, it remains important to have an ideal [Parnas 86]. Such an ideal:

- Provides a basis for standardizing an organization's processes and products,
- Provides a standard against which progress can be measured, and,
- Provides guidance – it helps developers understand what needs to be done next and when they are finished.

Because it is so often true that (1) requirements cannot be fully understood before at least starting to build the system and (2) a perfect SRS cannot be produced even when the requirements are understood, some approaches advocated in the literature do not even attempt to produce a definitive SRS. For example, some authors advocate going directly from a problem model to design or from a prototype implementation to the code. While such approaches may be effective on some developments, they are inconsistent with the notion of software development as an *engineering* discipline. The development of technical specifications is an essential part of a controlled engineering process. This does not mean that the SRS must be entire or perfect before anything else is done but that its development is a fundamental goal of the process as a whole. That we may currently lack the ability to write good specifications in some cases does not change the fact that it is useful and necessary to try.

7. State of the Practice

Over the years, a large number of analysis and specification techniques have evolved. The general trend has been for software engineering techniques to be applied first to coding problems (e.g., complexity, ease of change), then to similar problems occurring earlier and earlier in the life cycle. Thus the concepts of structured programming led eventually to structured design and analysis; more recently, the concepts of object oriented programming have led to object oriented design and analysis. The following discussion characterizes the major schools of thought and provides pointers to instances of methods in each school. The general strengths and weaknesses of the various techniques are discussed relative to the requirements difficulties and the desirable qualities of analysis and specification methods.

It is characteristic of the immature state of requirements as a discipline that the more specific one gets, the less agreement there is. There is not only disagreement in terminology, approach, and the details of different methods, there is not even a commonly accepted classification scheme. The following general groupings are based on the evolution of the underlying concepts and the key distinctions that reflect paradigmatic shifts in requirements philosophy.

7.1 Functional Decomposition

Functional decomposition was originally applied to software requirements to abstract from coding details. Function decomposition focuses on understanding and specifying what processing the software is required to do. The general strategy is to define the required behavior as a mapping from inputs to outputs. Ideally, the analysis proceeds top down, first identifying the function associated with the system as a whole. Each subsequent step decomposes the set of functions into steps or sub-functions. The result is a hierarchy of functions and the definitions of the functional interfaces. Each level of the hierarchy adds detail about the processing steps necessary to accomplish the more abstract function above. The function above controls the processing of its subfunctions. In a complete decomposition, the functional hierarchy specifies the "calls" structure of the implementation. One example of a methodology based on functional decomposition is Hamilton and Zeldin's Higher Order Software [Hamilton 76].

The advantage of functional decomposition is that the specification is written using the language and concepts of the implementors. It communicates well to the designers and coders. It is written in terms of the solution space so the transition to design and code is straight forward.

Common complaints are that functional specifications are difficult to communicate, introduce design decisions prematurely, and difficult to use or change. Because functional specifications are written in the language of implementation, people who are not software or systems experts find them difficult to understand. Since there are inevitably many possible ways of decomposing functions into subfunctions, the analyst must make decisions that are not requirements. Finally, since the processing needed in one step depends strongly on what has been done on the previous step, functional decomposition results in components that are closely coupled. Understanding or changing one function requires understanding or changing all the related functions.

As software has increased in complexity and become more visible to non-technical people, the need for methods addressing the weaknesses of functional decomposition has likewise increased. However, in spite of its drawbacks, functional specification remains in wide use.

7.2 Structured Analysis

Structured analysis was developed primarily as a means to address the accidental difficulties attending problem analysis and, to a lesser extent, requirements specification, using functional decomposition. Following the introduction of structured programming as a means to gain intellectual control over increasingly complex programs, structured analysis evolved from functional decomposition as a means to gain intellectual control over system problems.

The most basic assumption behind structured analysis is that the accidental difficulties can be addressed by a systematic approach to problem analysis using [Svoboda 90]:

- a common conceptual model for describing all problems,
- a set of procedures suggesting the general direction of analysis and an ordering on the steps,
- a set of guidelines or heuristics supporting decisions about the problem and its specification, and
- a set of criteria for evaluating the quality of the product.

While structured analysis still contains the decomposition of functions into subfunctions, the focus of the analysis shifts from the processing steps to the data being processed. The analyst views the problem as constructing a system to transform data. He analyzes the sources and destinations of the data, determines what data must be held in storage, what transformations are done on the data, and the form of the output.

Common to the structured analysis approaches is the use of data flow diagrams and data dictionaries. Data flow diagrams provide a graphic representation of the movement of data through the system (typically represented as arcs) and the transformations on the data (typically represented as nodes). The data dictionary supports the data flow diagram by providing a repository for the definitions and descriptions of each data item on the diagrams. Required processing is captured in the definitions of the transformations. Associated with each transformation node is a specification of the processing the node does to transform the incoming data items to the outgoing data items. The form of the specification is either another data flow diagram or a textual description called a “MiniSpec”. A MiniSpec may be expressed in a number of different ways including English prose, decision tables, or a procedure definition language (PDL).

Structured analysis approaches originally evolved for management information systems (MIS). Examples of widely used strategies include those described by DeMarco [DeMarco 78] and Gane and Sarson [Gane 79]. “Modern” structured analysis was introduced to provide more guidance in modeling systems as data flows as exemplified by Yourdon [Yourdon 89]. Extensions have been made to structured analysis to adapt the methods to the needs of embedded control systems by adding notations to capture control behavior. These techniques are collectively known as structured analysis/real time (SA/RT). Major variations of SA/RT have been described by Ward and Mellor [Ward 86] and Hatley and Pirbhai [Hatley 87]. A good summary of structured analysis concepts with extensive references is given by Svoboda [Svoboda 90].

Contributions of structured analysis include the notion that there should be a systematic (and hopefully predictable) approach to analyzing a problem, decomposing it into parts, and describing the relationships between the parts. It includes the notion that there should be a common set of conceptual structures for analyzing problems and communicating the results. Structured analysis techniques also introduced graphical specifications to support communication to a wide audience.

While structured analysis techniques have continued to evolve and have been widely used, there remain a number of common criticisms. When used in problem analysis, a common complaint

is that structured analysis provides insufficient guidance. Analysts have difficulty deciding which parts of the problem to model as data, what to model as transformations, and what should be aggregated. While the gross steps of the process are reasonably well defined, there is only very general guidance (in the form of heuristics) on what specific questions the analyst needs to answer next. Similarly, practitioners find it difficult to know when to stop decomposition and addition of detail. Particularly in the hands of less experienced practitioners, data flow models continue to incorporate a variety of detail that more properly belongs to design or implementation and the diagrams themselves become unmanageably complex.

These difficulties result from the weak constraints imposed by the conceptual model. A goal of the developers of structured analysis was to create a very general approach to modeling systems; in fact, one that could be applied equally to model human enterprises, hardware applications, software applications of different kinds, and so on. Unfortunately, such generality can be achieved only by abstracting away any semantics that are not common to all of the types of systems one models. The model itself can provide little guidance relevant to a particular system. Since the conceptual model applies equally to requirements analysis and design analysis, its semantics provide no basis for distinguishing the two. Similarly, such models can support only very weak syntactic criteria for assessing the quality structured analysis specifications. For example, the test for completeness and consistency in data flow diagrams is limited to determining that the transformations at each level are consistent in name and number with the data flows of the level above.

This does not mean one cannot develop data flow specifications that are easy to understand, represent requirements without introducing design decisions, communicate effectively with the user, and so on. Clearly, at least for systems of modest complexity, it is possible to do so. The problem is that the weakness of the conceptual model means that a specification's quality depends too much on the experience, insight, and expertise of the analyst. The developer must provide the necessary discipline because the model itself is relatively unconstrained.

Finally, there is also the problem that the creation of an SRS meeting our quality criteria is difficult. While data flow diagrams have proven useful for visualizing gross relationships in the system, such a graphical approach is unsuitable for capturing mathematical relations and unwieldy for capturing detailed specifications of value, timing, or accuracy. In general, the data flow diagrams and attendant dictionaries do not, themselves, provide any support for organizing an SRS to satisfy the packaging goals of readability, ease of reference and review, or reusability. In fact, for many of the published methods, there is no explicit process step, structure, or guidance for producing an SRS as a distinct document.

7.3 Operational Specification

The operational[1] approach focuses on addressing two of the essential requirements dilemmas. The first is that we often do not know exactly what should be built until we build it. The second is the problem inherent in moving from a particular specification of requirements (what to build) to a design that satisfies those requirements (how to build it). The closer the requirements specifi-

cation is to the design, the easier the transition but the more likely it is that design decisions are made prematurely.

The operational approach seeks to address these problems, among others, by supporting development of executable requirements specifications. Key elements of an operational approach are a formal specification language and an engine for executing well-formed specifications written in the language. Operational approaches may also include automated support for analyzing properties of the formal specification and for transforming the specification into an equivalent implementation. A good description of the operational approach, its rationale, and goals is given by Zave [Zave 84].

The underlying reasoning about the benefits of the operational approach is as follows:

- Making the requirements specification itself executable obviates the dilemma that one must build the system to know how to build it. The developer writes the requirements specification in the formal language. The specification may then be executed to validate that the customer's needs have been captured and the right system specified (e.g., one can apply scenarios and test cases). The approach is easier and more cost effective than conventional prototyping since the specification and the "prototype" are the same thing.
- Operational specifications allow the developer to abstract from design decisions while simplifying the transition from requirements to design and implementation. Transition to design and implementation is both simple and automatable because the behavioral requirements are already expressed in terms of computational mechanisms. Design decisions concerning efficiency, resource management, and target language realization are abstracted from in the computational model.

For general applications, operational approaches have achieved only limited success. This is at least in part due to the failure to achieve the necessary semantic distinction between the operational computational model and conventional programming. The benefits of the approach are predicated on the assumption that the operational model can be written in terms of the problem domain without the need to introduce conceptual structures belonging to the solution domain. In practice, this goal has proven elusive. To achieve generality, operational languages have typically had to introduce implementation constructs. The result is not a requirements specification language but a higher-level programming language. As noted by Parnas [Parnas 85b] and Brooks [Brooks 87], the specification ends up giving the solution method rather than the problem statement.

Where operational approaches have enjoyed some success is in highly-constrained and well understood problem domains. These are areas where it has been possible to essentially enumerate the set of alternatives or provide a theory completely characterizing the domain. Examples in computer science include compilers and lexical analyzers; applications of both types can now be generated from formal specifications of the respective languages and grammars.

1. We use the term "operational" here specifically to denote approaches based on executable specifications in the sense of Zave [Zave 84]. The term is sometimes used to contrast with axiomatic specification – that is not the meaning here.

7.4 Object Oriented Analysis (OOA)

There is currently considerable discussion in the literature, and little agreement, on exactly what should and should not be considered "object oriented." OOA has evolved from at least two significant sources, information modeling and object oriented design. Each has contributed to current views of OOA, and the proponents of each emphasize somewhat different sets of concepts. For the purposes of this tutorial, we are not interested in which method is by some measure "more object oriented" but in the distinct contributions of the object oriented paradigm to analysis and specification. For an overview of OOA concepts and methods see Balin's article [Balin 94]; Davis' book [Davis 93] includes both discussion and examples. Examples of recent approaches self-described as object oriented include work by Rumbaugh [Rumbaugh 91], Coad and Yourdon [Coad 91], and Shlaer and Mellor [Shlaer 88].

Most OOA techniques differ from structured analysis in the approach to decomposing the problem into parts and in the methods for describing the relationships between the parts. In OOA, the analyst decomposes the problem into a set of interacting objects based on the entities and relationships extant in the problem domain. An object encapsulates a related set of data, processing, and state (thus, a significant distinction between object oriented approaches and structured analysis is the encapsulation of both data and related processing together). Objects provide externally accessible functions, typically called services or methods. Objects may hide information about their internal structure, data, or state from other objects. Conversely, they may provide processing, data, or state information through the services defined on the object interface. Dynamic relationships between objects are captured in terms of message passing (i.e., one object sends a message to invoke a service or respond to an invocation). The analyst captures static relationships in the problem domain using the concepts of aggregation and classification. Aggregation is used to capture whole/part relationships. Classification is used to capture class/instance relationships (also called "is-a" or inheritance relationships).

The structural components of OOA (e.g., objects, classes, services, aggregation) support a set of analytic principles. Of these, two directly address requirements problems:

1. From information modeling comes the assumption that a problem is easiest to understand and communicate if the conceptual structures created during analysis map directly to entities and relationships in the problem domain. This principle is realized in OOA through the heuristic of mapping objects to real-world entities and relationships between objects to existing or required relationships in the problem domain. The object paradigm is used to model both the problem and the relevant problem context.
2. From early work on modularization by Parnas [Parnas 72] and abstract data types, by way of object oriented programming and design, come the principles of information hiding and abstraction. The principle of information hiding guides one to limit access to information on which other parts of the system should not depend. For example, requirements that are likely to change, should be hidden from other parts of the system. Information hiding is implemented in object oriented approaches by encapsulating the information in an object and limiting the access through the object interface. The principle of abstraction says that only the relevant or essential information should be presented. Abstraction is implemented in OOA by defining object interfaces that provide access only to the essence of

the data or state information encapsulated by an object (conversely hiding the accidentals).

The principles and mechanisms of OOA provide a basis for attacking the essential difficulties of comprehension, communication, and control. The principle of problem domain modeling helps guide the analyst in distinguishing requirements (what) from design (how). Where the objects and their relationships faithfully model entities and relationships in the problem, they are understandable by the customer and other domain experts; this supports early comprehension of the requirements.

The principles of information hiding and abstraction, with the attendant object mechanisms, provide a basis for addressing the essential problems of control and communication. Objects are used to divide the requirements into distinct parts, abstract from details, and control dependencies between the parts. For change management, objects can hide requirements likely to change. To improve communication, objects hide irrelevant detail and define abstractions providing only the essential information. This provides a basis for managing complexity and improving readability. Likewise objects provide a basis for constructing reusable requirements units of related functions and data.

The potential benefits of OOA are often diluted by the way the key principles are manifest in particular methods. While the conceptual structures of OOA are intended to model essential aspects of the application domain, this goal is typically not supported by an corresponding conceptual model of the domain behavior. Like structured analysis, object modeling mechanisms and techniques are intended to be generic rather than application specific. One result is insufficient guidance in developing appropriate object decompositions. Just as structured analysis practitioners have difficulty choosing appropriate data flows and transformations, OOA practitioners have difficulty choosing appropriate objects and relationships.

In practice, one finds the notion that one can develop the structure of a system or a requirements specification based on physical structure is often oversold. It is true that the elements of the physical world are often quite stable (especially relative to software details) and that real-world based models have intuitive appeal. It is not, however, the case that everything that must be captured in requirements has a physical analog. An obvious example is shared state information. Further, many real world structures are themselves arbitrary and likely to change (e.g., where two hardware functions are put on one platform to save weight). While the notion of basing requirements structure on physical structure is a useful heuristic, more is needed to develop a good SRS.

The lack of a strong conceptual model rooted in the problem domain also has the result that OOA specification languages often default to mechanisms more appropriate to the solution domain (this is no doubt also due to OOA's origins in object oriented design and programming). Symptomatic of this problem is that analysts find themselves debating about object language features and their properties rather than about the properties of the problem. A common example is the use of message passing complete with message passing protocols where one object uses information defined in another. In the problem domain it is often irrelevant whether information is actively solicited or passively received. In fact there may be no notion of messages or transmission

at all. Nonetheless one finds analysts debating about which object should initiate a request and the resulting anomaly of passive entities modeled as active. For example, to get information from a book one might request that the book "read itself" and "send" the requested information in a message. Such decisions are about OOA mechanisms or design, not about requirements.

A more serious complaint is that most current OOA methods inadequately address developing a good SRS. The focus of all of the OOA techniques cited above is on problem analysis. If the SRS is addressed at all, the assumption is that the principles applied to problem understanding and modeling are sufficient, when results are written down, to produce a good specification. Experience suggests otherwise. As we have discussed, there are inherently tradeoffs that must be made in developing a specification that meets the need of any particular project. Making effective tradeoffs requires forethought about the characteristics of the problem and the desired characteristics of the specification. Achieving good results requires a disciplined and thoughtful approach to the SRS itself, not just the problem.

7.5 The SCR A-7E OFP

Where most of the techniques thus far discussed have focused on problem analysis, the requirements work at the United States Naval Research Laboratory (NRL) focused on issues in developing a good SRS. The NRL team led by David Parnas, initiated the Software Cost Reduction (SCR) project in 1978 to demonstrate the feasibility and effectiveness of advanced software engineering techniques by applying them to a real program, the Operational Flight Program (OFP) for the A-7E aircraft. To demonstrate that (then academic) techniques like information hiding, formal specification, abstract interfaces, and cooperating sequential processes could help make software easier to understand, maintain, and change, the A-7E OFP was re-engineered.

Since no existing documentation adequately captured the software requirements, the first step was to develop an effective SRS. In this process, the SCR project identified a number of properties a good SRS should have and a set of principles for developing requirements documentation [Heninger 80]. These included the principle of separation of concerns to aid readability and make the document easy to change, and the use of formal, mathematically based specifications in place of prose to make the document precise, concise, consistent, and complete.

While other requirements approaches have stated similar objectives, the SCR project is unique in having applied software engineering principles to develop a standard SRS organization, a specification method, review method [Parnas 85a], and notations consistent with those principles. The SCR project is also unique in making publicly available a complete, model SRS of a significant system [Alspaugh 92].

A number of issues were left unresolved by the original SCR work. While the product of the requirements analysis was well documented, the underlying process and method were never fully described. Since the original effort was to re-engineer an existing system, it was not clear how effective the techniques would be on a new development. Since the developers of the A-7E requirements document were researchers, it was unclear whether industrial developers would find the rather formal method and notation useable, readable, or effective. Finally, while the A-7E

SRS organization is reasonably general, many of the specification techniques are targeted to real-time, embedded applications. As discussed in the following section, more recent work by Parnas, NRL, and others has addressed many of the open questions about the SCR approach.

8. Trends and Emerging Technology

While improved discipline will address requirement's accidental difficulties, addressing the essential difficulties requires technical advances. Significant trends, in some cases backed by industrial experience, have emerged over the past few years that offer some hope for improvement:

- *Domain specificity:* Requirements methods will provide improved analytic and specification support by being tailored to particular classes of problems. Historically requirements approaches have been advanced as being equally useful to a wide variety of types of applications. For example, structured analysis methods were deemed to be based on conceptual models that were "universally applicable" (e.g., [Ross 77]); similar claims have been made for object oriented approaches.

  Such generality comes at the expense of ease of use and amount of work the analyst must do for any particular application. Where the underlying models have been tailored to a particular class of applications, the properties common to the class are embedded in the model. The amount of work necessary to adapt the model to a specific instance of the class is relatively small. The more general the model, the more decisions that must be made, the more information that must be provided, and the more tailoring that must be done. This provides increased room for error and, since each analyst will approach the problem differently, makes solutions difficult to standardize. In particular, such generality precludes standardization of sufficiently rigorous models to support algorithmic analysis of properties like completeness and consistency.

  Similar points have been expressed in a recent paper by Jackson [Jackson 94]. He points out that some of the characteristics separating real engineering disciplines from what is euphemistically described as "software engineering" are well understood procedures, mathematical models, and standard designs specific to narrow classes of applications. Jackson points out the need for software methods based on the conceptual structures and mathematical models of behavior inherent in a given problem domain (e.g., publication, command and control, accounting, and so on). Such common underlying constructs can provide the engineer guidance in developing the specification for a particular system.

- *Practical formalisms:* Like so many of the promising technologies in requirements, the application of formal methods is characterized by an essential dilemma. On one hand, formal specification techniques hold out the only real hope for producing specifications that are precise, unambiguous, and demonstrably complete or consistent. On the other, industrial practitioners widely view formal methods as impractical. Difficulty of use, inability to scale, readability, and cost are among the reasons cited. Thus, in spite of significant technical progress and a growing body of literature, the pace of adoption by industry has been extremely slow.

  In spite of the technical and technical transfer difficulties, increased formality is necessary. Only by placing development on a mathematical basis will we be able to

acquire the intellectual control over the medium needed to develop complex systems with any assurance of success. The solution is better formal methods – methods that are practical given the time, cost, and personnel constraints of industrial development.

Engineering models and the training to use them are *de rigueur* in every other discipline that builds large, complex, or safety-critical systems. Builders of a bridge or skyscraper who did not employ mathematical models to predict reliability and safety would be held criminally negligent in the event of failure. It is only the relative youth of the software discipline that permits us to get away with less. But, we cannot expect great progress overnight. As Jackson [Jackson 94] notes, the field is sufficiently immature that "the prerequisites for a more mathematical approach are not in place." Further, many of those practicing our craft lack the background required of licensed engineers in other disciplines [Parnas 89]. Nonetheless, sufficient work has been done to show that more formal approaches are practical and effective in industry. For an overview of formal methods and their role in practical developments, the reader is referred to Rushby's summary work [Rushby 93].

- *Improved tool support:* It remains common to walk into the office of a software development manager and find the shelves lined with the manuals for CASE tools that are not in use. In spite of years of development and the claims of manufacturers, many industrial developers have found the available requirements CASE tools of marginal benefit.

  Typically, the fault lies not so much with the tool vendor but with the underlying method or methods the tool seeks to support. The same generality and lack of strong underlying conceptual model that makes the methods weak limits the benefits of automation. Since the methods do not adequately constrain the problem space and offer little specific guidance, the corresponding tool cannot actively support the developer in making difficult decisions. Since the model and SRS are not standardized, its production eludes automated support. Since the underlying model is not formal, only trivial syntactic properties of the specification can be evaluated. Most such tools provide little more than a graphic interface and requirements data base.

  Far more is now possible. Where the model, conceptual structures, notations, and process are standardized, significant automated support becomes possible. The tool can use information about the state of the specification and the process to guide the developer in making the next step. It can use standardized templates to automate rote portions of the SRS. It can use the underlying mathematical model to determine to what extent the specification is complete and consistent. While only the potential of such tools has yet been demonstrated, there are sufficient results to project the benefits.

- *Integrated paradigms:* One of the Holy Grails of software engineering has been the integrated software development environment. Much of the frustration in applying currently available methods and tools is the lack of integration, not just in the tool interfaces, but in the underlying models and conceptual structures. Even where an approach works well for one phase of development, the same techniques are either difficult to use in the next phase or there is no clear transition path. Similarly tools are either focused on a small subset of the many tasks (e.g., analysis but not documentation) or attempt to address the entire life cycle but support none of it well. The typical

development employs a hodgepodge of software engineering methodologies and ad hoc techniques. Developers often build their own software to bridge the gap between CASE platforms.

In spite of a number of attempts, the production of a useful integrated set of methods and supporting environment has proven elusive. However, it now appears that there is sufficient technology available to provide, if not a complete solution, at least the skeleton for one.

The most significant methodological trend can be described as convergent evolution. In biology, convergent evolution denotes a situation where common evolutionary pressures lead to similar characteristics (morphology) in distinct species. An analogous convergence is ongoing in requirements. As different schools of thought have come to understand and attempt to address the weaknesses and omissions in their own approaches, the solutions have become more similar. In particular, the field is moving toward a common understanding of the difficulties and common assumptions about the desired qualities of solutions. This should not be confused with the bandwagon effect that often attends real or imaginary paradigm shifts (e.g., the current rush to object oriented everything). Rather it is the slow process of evolving common understanding and changing conventional practices.

Such trends and some preliminary results are currently observable in requirements approaches for embedded software. In the 1970's the exigencies of national defense and aerospace applications resulted in demand for complex, mission critical software. It became apparent early on that available requirements techniques addressed neither the complexity of the systems being built nor the stringent control, timing and accuracy constraints of the applications. Developers responded by creating a variety of domain specific approaches. Early work by TRW for the U.S. Army on the Ballistic Missile Defense system produced the Software Requirements Engineering Method (SREM) [Alford 77] and supporting tools. Such software problems in the Navy led to the SCR project. Ward, Mellor, Hatley, and Pirbhai ([Ward 86], [Hatley 87]) developed extensions to structured analysis techniques targeted to real time applications. Work on the Israeli defense applications led Harel to develop statecharts [Harel 87] and the supporting tool Statemate.

The need for high-assurance software in mission and safety critical systems also led to the introduction of practical formalisms and integrated tools support. TRW developed REVS [Davis 77] and other tools as part of a complete environment supporting SREM and other phases of the life cycle. The SCR project developed specification techniques based on mathematical functions and tabular representations [Heninger 80]. These allowed a variety of consistency and completeness checks to be performed by inspection. Harel introduced a compact graphic representation of finite state machines with a well-defined formal semantics. These features were subsequently integrated in the Statemate tool that supported symbolic execution of statecharts for early customer validation and limited code generation. All of these techniques began to converge on an underlying model based of finite state automata.

More recent work has seen continuing convergence toward a common set of assumptions and similar solutions. Recently, Ward and colleagues have developed the Real-Time Object Oriented Modeling (ROOM) method [Selic 94]. ROOM integrates concepts from operational specifica-

tion, object oriented analysis, and statecharts. It employs an object oriented modeling approach with tool support. The tool is based on a simplified statechart semantics and supports symbolic execution and some code generation. The focus of ROOM currently remains on problem modeling, transition to design, and execution rather than formal analysis.

Nancy Leveson and her colleagues have adapted statecharts to provide a formally based method for embedded system specification [Jaffe 91]. The approach has been specifically developed to be useable and readable by practicing engineers. It employs both the graphical syntax of statecharts and a tabular representation of functions similar to those used in the SCR approach. Its underlying formal model is intended to support formal analysis of system properties, with an emphasis on safety. The formal model also supports symbolic execution. These techniques have been applied to develop a requirements specification for parts of the Federal Aviation Administration's safety critical Traffic Alert and Collision Avoidance System (TCAS) [Leveson 94].

8.1 Naval Research Laboratory SCR-Related Work

Many of the issues discussed in this paper are being addressed in follow-on work to the Naval Research Laboratory's SCR project. This includes work by Parnas and colleagues on domain-specific requirements models, work by Faulk and colleagues on techniques for industrial-strength requirements specification, and work by Heitmeyer and colleagues at NRL on formal requirements methods and tools.

Work by Parnas and colleagues has extended the SCR requirements method to define a standard mathematical model for embedded system requirements. This model has been described by Parnas and Madey [Parnas 91] and applied to a small embedded system by van Schouwen [Van Schouwen 90]. Parnas has also applied some of the SCR techniques to the validation of the safety shutdown software for the Darlington Nuclear Reactor [Parnas 90].

Work at the Software Productivity Consortium by Faulk and colleagues [Faulk 92] has integrated the SCR approach with object oriented and graphic techniques and defined a complete requirements analysis process including a detailed process for developing an SRS that satisfies the "goodness" criteria discussed above. The method and an example are described in The Consortium Requirements Engineering Method guidebook [Faulk 93]. These techniques have been applied effectively in development of requirements for Lockheed's avionics upgrade on the C–130J aircraft [Faulk 94]. The C–130J avionics software is a safety–critical system of approximately 100K lines of Ada code.

Work at NRL has extended the early SCR work in a number of areas including formal modeling and analysis of requirements, automated support, and technical transfer. Initial work examined the effectiveness of tool support for specifying and analyzing real-time systems using the Modchart language. Modecharts were developed by Jahanian, Mok and colleagues at the University of Texas to provide a formal, graphical specification language based on the SCR requirements model [Jahanian 88]. This work was extended at NRL to develop a tool set [Rose 94] for specifying requirements using Modecharts. The tool set supported symbolic execution of the Modecharts for validation of specified behavior and support for model checking to determine whether the specifications imply certain classes of safety assertions [Clements 93].

Subsequent work by Heitmeyer and colleagues has extended Parnas and Madey's work by defining a formal model for SCR-style requirements [Heitmeyer 95a]. The formal model provides notation and semantics for representing requirements as well as analytic techniques for demonstrating that a specification written in terms of the model is complete and consistent. This formal model has been used to develop a suite of prototype tools supporting analysis of requirements properties like completeness and consistency. Scale-up of the techniques is being addressed by the development of prototype tools.

Experiments conducted by NRL showed that automated checking would find consistency and completeness errors missed in manual inspections [Heitmeyer 93]. Thus, the current NRL tools provide for the creation and editing of SCR-style specifications and the automated analysis of a specification's consistency and completeness [Heitmeyer 95b]. To support requirements validation (e.g., for the customer), the NRL tool set also supports simulation. This capability permits symbolic execution of the underlying state-machine model and allows one to observe examples of the system behavior based directly on the requirements specification. The simulation achieves low cost and high-fidelity by using the same underlying formal model and data base of requirements used by the rest of the NRL tools [Heitmeyer 95c].

Goals of current NRL work include demonstration that the SCR technologies are practical and effective in industry, and the transfer of SCR methods and tools to industrial developers of high-assurance systems. NRL is working with industrial partners to apply the SCR technology to realistic problems and measure their effectiveness. Results and lessons learned will be used to support transfer of the technology to industry and eventual acquisition of the tool technology by commercial tool developers.

Planned work will extend the NRL technologies to provide the basis for an integrated development paradigm. Where current analysis capabilities address application-independent properties (i.e., model properties like internal completeness and consistency), planned tools will support demonstration of application-dependent properties like safety. These include facilities for tool-supported theorem proving and model checking. Planned work also encompasses automated test vector (inputs and expected outputs) generation, for black box testing, from the same underlying SCR specification and formal model.

While none of the work mentioned can be considered a complete solution it is clear that (1) the work is converging toward common assumptions and solutions, (2) the approaches all provide significantly improved capability to address requirements difficulties, and (3) the solutions can be effectively applied in industry.

9. Conclusions

Requirements are intrinsically hard to do well. Beyond the need for discipline, there are a host of essential difficulties that attend both the understanding of requirements and their specification. Further, many of the difficulties in requirements will not yield to technical solution alone. Addressing all of the essential difficulties requires the application of technical solutions in the

context of human factors such as the ability to manage complexity or communicate to diverse audiences. A requirements approach that does not account for both technical and human concerns can have only limited success. For developers seeking new methods, the lesson is caveat emptor. If a vendor tells you his method makes requirements easy, keep a hand on your wallet.

Nevertheless, difficulty is not impossibility and the inability to achieve perfection is not a mandate for surrender. While all of the approaches discussed have significant weaknesses, they all contribute to the attempt to make requirements analysis and specification a controlled, systematic, and effective process. Though there is no easy path, experience confirms that the use of **any** careful and systematic approach is preferable to an ad hoc and chaotic one. Further good news is that if the requirements are done well, chances are much improved that the rest of the development will also go well. Unfortunately, ad hoc approaches remain the norm in much of the software industry.

A final observation is that the benefits of good requirements come at a cost. Such a difficult and exacting task cannot be done properly by personnel with inadequate experience, training, or resources. Providing the time and the means to do the job right is the task of responsible management. The time to commit the best and brightest is before, not after, disaster occurs. The monumental failures of a host of ambitious developments bear witness to the folly of doing otherwise.

10. Further Reading

Those seeking more depth than this tutorial can provide should read Alan Davis' book *Software Requirements: Objects, Functions, and States* [Davis 93]. In addition to a general discussion of issues in software requirements, Davis illustrates a number of problem analysis and specification techniques with a set of common examples. Davis also provides a comprehensive annotated bibliography. For a better understanding of software requirements in the context of systems development, the reader is referred to the book of collected papers edited by Thayer and Dorfman, *System and Software Requirements Engineering* [Thayer 90]. This tutorial work contains in one volume both original papers and reprints from many of the authors discussed above. The companion volume, *Standards, Guidelines, and Examples on System and Software Requirements Engineering* [Dorfman 90b] is a compendium of international and U.S. government standards relating to system and software requirements and provides some illustrating examples.

11. Acknowledgements

C. Colket at SPAWAR and E. Wald at ONR helped support development of this report. Thanks also to Art Pyster and Merlin Dorfman for supporting the initial version of this work. The quality of this paper has been much improved thanks to thoughtful reviews by Paul Clements, Connie Heitmeyer, Jim Kirby, Bruce Labaw, Richard Morrison, and David Weiss.

REFERENCES

[Alford 77] Alford, M., "A Requirements Engineering Methodology for Real-Time Processing Requirements," *IEEE Transactions on Software Engineering*, v. 3, no. 1, January 1977, pp. 60−69.

[Alford 79] Alford, M. and J. Lawson, "Software Requirements Engineering Methodology (Development)," *RADC-TR-79-168*, U.S. Air Force Rome Air Development Center, June 1979.

[Alspaugh 92] Alspaugh, T., S. Faulk, K. Britton, R. Parker, D. Parnas, and J. Shore, *Software Requirements for the A-7E Aircraft*, NRL/FR/5530-92-9194. Washington, D.C.: Naval Research Laboratory, 1992.

[Balin 94] Balin, S., "Object-Oriented Requirements Analysis," in *Encyclopedia of Software Engineering*, J. Marciniak ed., John Wiley & Sons, N.Y., 1994, pp.740−756.

[Basili 81] Basili, V., and D. Weiss, "Evaluation of a Software Requirements Document by Analysis of Change Data," *Proceedings of the Fifth International Conference on Software Engineering*, San Diego, California, March 1981, pp. 314−323.

[Boehm 81] Boehm, B., *Software Engineering Economics*, Prentice Hall, New Jersey, 1981.

[Boehm 94] Boehm, B., P. Bose, E. Horowitz, and M. Lee, "Software Requirements as Negotiated Win Conditions," in *Proceedings of the First International Conference on Requirements Engineering*, Colorado Springs, Colorado, April 18−22, 1994, pp. 74−83.

[Brooks 75] Brooks, F., *The Mythical Man-Month*, Addison-Wesley, 1975.

[Brooks 87] Brooks, F., "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, April 1987, pp. 10−19.

[CECOM 89] *Software Methodology Catalog: Second Edition*, Technical report C01-091JB-0001-01, U.S. Army Communications-Electronics Command, Fort Monmouth, New Jersey, March, 1989.

[Clements 93] Clements, P., C. Heitmeyer, B. Labaw, and A. Rose, "MT: A Toolset for Specifying and Analyzing Real-Time Specification," in *Proceedings of the Real-Time Systems Symposium*, Raleigh-Durham, North Carolina, December 1993, pp. 12−22.

[Coad 90] Coad, P., and E. Yourdon, *Object Oriented Analysis*, Prentice Hall, New Jersey, 1990.

[Davis 77] Davis, C. and C. Vick, "The Software Development System," *IEEE Transactions on Software Engineering*, v. 3, no. 1, January, 1977, pp. 69–84.

[Davis 88] Davis, A., "A Taxonomy for the Early Stages of the Software Development Life Cycle," *Journal of Systems and Software*, September, 1988, pp. 297–311.

[Davis 93] Davis, A., *Software Requirements (Revised): Objects, Functions, and States*, Prentice Hall, New Jersey, 1993.

[DeMarco 78] DeMarco, T., *Structured Analysis and System Specification*, Prentice Hall, New Jersey, 1978.

[Dorfman 90b] Dorfman, M., and R. Thayer, eds., *Standards, Guidelines, and Examples on System and Software Requirements Engineering*, IEEE Computer Society Press, Los Alamitos, California, 1990.

[Faulk 92] Faulk, S., J. Brackett, P. Ward, and J. Kirby, Jr., The Core Method for Real-Time Requirements, *IEEE Software*, Vol. 9, No. 5, September 1992.

[Faulk 93] Faulk, S., L. Finneran, J. Kirby Jr., and A. Moini, *Consortium Requirements Engineering Guidebook*, Version 1.0, SPC-92060-CMC, Software Productivity Consortium, Herndon, Virginia, 1993.

[Faulk 94] Faulk, S., L. Finneran, J. Kirby, S. Shah, and J. Sutton, " Experience Applying the CoRE Method to the Lockheed C-130J," *Proceedings of the Ninth Annual Conference on Computer Assurance*, IEEE 94CH3415-7, Gaithersburg, Maryland, June 1994, pp. 3–8.

[GAO 79] U.S. General Accounting Office, *Contracting for Computer Software Development–Serious Problems Require Management Attention to Avoid Wasting Additional Millions*, Report FGMSD-80-4, November 1979.

[GAO 92] U.S. General Accounting Office, *Mission Critical Systems: Defense Attempting to Address Major Software Challenges*, GAO/IMTEC-93-13, December 1992.

[Gane 79] Gane, C., and T. Sarson, *Structured Systems Analysis,* Prentice Hall, New Jersey, 1979.

[Hamilton 76] Hamilton, M. and S. Zeldin, "Higher Order Software−A Methodology for Defining Software," *IEEE Transactions on Software Engineering*, v. 2, no. 1, January 1976, pp 9−32.

[Harel 87] Harel, D., "Statecharts: a Visual Formalism for Complex Systems," Science of Computer Programming 8, 1987, pp. 231−274.

[Hatley 87] Hatley, D., and I. Pirbhai, *Strategies for Real-Time Specification*, Dorset House, New York, New York, 1987.

[Heitmeyer 95a] Heitmeyer, C., B. Labaw, and D. Kiskis, "Consisency Checking of SCR-Style Requirements Specifications," in *Proceedings, IEEE International Symposium on Requirements Engineering*, March 1995.

[Heitmeyer 95b] Heitmeyer, C., R. Jeffords, and B. Labaw. *Tools for Analyzing SCR-Style Requirements Specifications: A Formal Foundation*, NRL Technical Report NRL−7499, U.S. Naval Research Laboratory, Washington, DC, 1995.

[Heitmeyer 95c] Heitmeyer, C., A. Bull, C. Gasarch, and B. Labaw, "SCR*: A Toolset for Specifying and Analyzing Requirements," *Proceedings of the 10th IEEE Conference on Computer Assurance (COMPASS)*, Gaithersburg, MD, June 1995.

[Heninger 80] Heninger, K., "Specifying Software Requirements for Complex Systems: New Techniques and Their Application", *IEEE Transactions on Software Engineering*, v. 6, no. 1, January, 1980.

[Hester 81] Hester, S. , D. Parnas, and D. Utter, "Using Documentation as a Software Design Medium", *Bell System Technical Journal*, v. 60, no. 8, October 1981, pp 1941−1977.

[Hsai 94] Hsai, P., J. Samuel, J. Gao, D. Kung, Y. Toyoshimi, and C. Chen, "Formal Approach to Scenario Analysis," *IEEE Software*, March 1994, pp. 33−41.

[Jackson 83] Jackson, M., *System Development*, Prentice Hall, New Jersey, 1983.

[Jackson 94] Jackson, M., "Problems, Methods, and Specialization," *IEEE Software*, November 1994, pp. 57−62.

[Jaffe 91] Jaffe, M., N. Leveson, M. Heimdahl, and B. Melhart, "Software Requirements Analysis for Real-Time Process-Control Systems", *IEEE Transactions on Software Engineering*, Vol. 17, No. 3, March 1991, pp. 241-257.

[Jahanian 88] Jahanian, F., R. Lee, and A. Mok, "Semantics of Modchart in Real-Time Logic," in *Proceedings of the 21st Hawaii International Conference on System Sciences*, January, 1988.

[Leveson 94] Leveson, N., M. Heimdahl, H. Hildreth, and J. Reese, "Requirements Specification for Process-Control Systems," *IEEE Transactions on Software Engineering*, Vol. 20, No. 9, September 1994.

[Lutz 93] Lutz, R., "Analyzing Software Requirements Errors in Safety-Critical Embedded Systems," *Proceedings, IEEE International Symposium on Requirements Engineering*, January 4–6, 1993, pp. 126–133.

[Parnas 72] Parnas, D., "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, v. 15, no. 12, December 1972, pp. 1053−1058.

[Parnas 85a] Parnas, D. and D. Weiss, "Active Design Reviews: Principles and Practices," in *Proceedings of the Eighth International Conference on software Engineering*, London, England, August 1985.

[Parnas 85b] Parnas, D., "Software Aspects of Strategic Defense Systems," *American Scientist*, September 1985, pp. 432−440.

[Parnas 86] Parnas, D., and P. Clements, "A Rational Design Process: How and Why to Fake It," *IEEE Transactions on Software Engineering*, v. 12, no. 2, February 1986, pp. 251−257.

[Parnas 90] Parnas, D., G. Asmis, J. Kendall, "Reviewable Development of Safety Critical Software", International conference on Control and Instrumentation in Nuclear Installations, Glasgow, Scotland, May 1990.

[Parnas 91] Parnas, D., and J. Madey, *Functional Documentation for Computer Systems Engineering (Version 2)*, CRL Report No. 237,. McMaster University, Hamilton, Ontario, Canada, September 1991.

[Parnas 89] Parnas, D., *Education for Computing Professionals*, Technical Report 89−247, Department of Computing and Information Science, Queens University, Kingston, Ontario, 1989.

[Potts 94] Potts, C., K. Takahashi, A. Anton, "Inquiry-Based Requirements Analysis," *IEEE Software*, March 1994, pp. 21−32.

[Shlaer 88] Shlaer, S. and S. Mellor, *Object−Oriented Systems Analysis: Modeling the World in Data*, Prentice Hall, New Jersey, 1988.

[Rose 94] Rose, A., M. Perez, P. Clements, *Modechart Toolset User's Guide*, U.S. Naval Research Laboratory Memorandum Report NRL/MR/5540−94−7427, Washington, D.C., February, 1994.

[Ross 77] Ross, D. and K. Schoman Jr., "Structured Analysis for Requirements Definitions," *IEEE Transactions on Software Engineering*, v. 3, no. 1, January 1977, pp. 6−15.

[Rumbaugh 91] Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, New Jersey, 1991.

[Rushby 93] Rushby, J., *Formal Methods and the Certification of Critical Systems*, CSL Technical Report SRI-CSL-93-07, SRI International, Menlo Park, California, November, 1993.

[Selic 94] Selic, B., G. Gullekson, and P. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, 1994.

[Svoboda 90] Svoboda, C., "Structured Analysis," in *Tutorial: System and Software Requirements Engineering*, R. Thayer and M. Dorfman, eds., IEEE Computer Society Press, Los Alamitos, California, 1990, pp. 218−237.

[Thayer 90] Thayer, R. and M. Dorfman, eds., *Tutorial: System and Software Requirements Engineering,* IEEE Computer Society Press, Los Alamitos, California, 1990.

[Van Schouwen 90] *The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application to Monitoring Systems,* Technical Report 90-276, Queens University, Hamilton, Ontario, 1990.

[Ward 86] Ward, P., and S. Mellor, *Structured Development for Real-Time Systems,* Volumes 1, 2, and 3, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Yourdon 89] Yourdon, E. *Modern Structured Analysis*, Yourdon Press/Prentice Hall, 1989.

[Zave 82] Zave, P., "An Operational Approach to Requirements Specification for Embedded Systems," *IEEE Transactions on Software Engineering*, v. 8, no. 3, May 1982, pp. 250−269.