Report Number: WUCS-84-6

1984-06-01

A Taxonomy of Requirements Specification Techniques

Gruia-Catalin Roman

A taxonomy is introduced and used as a backdrop against which current state-of-the-art in the requirements engineering field is reviewed. The emphasis is on identifying general trends and issues rather than offering the reader a literature survey. The contents of a requirements specification is presented in light of the consensus reached by both theoreticians and practitioners. The desirable proper ties of a requirements may alter the relative significance of difference properties. Finally, the classification of requirements specification techniques is approached from a total system design perspective. The paper shows that, despite significant growth, the requirements area still faces a... Read complete abstract on page 2.

A Taxonomy of Requirements Specification Techniques

Gruia-Catalin Roman

Complete Abstract:

A taxonomy is introduced and used as a backdrop against which current state-of-the-art in the requirements engineering field is reviewed. The emphasis is on identifying general trends and issues rather than offering the reader a literature survey. The contents of a requirements specification is presented in light of the consensus reached by both theoreticians and practitioners. The desirable proper ties of a requirements may alter the relative significance of difference properties. Finally, the classification of requirements specification techniques is approached from a total system design perspective. The paper shows that, despite significant growth, the requirements area still faces a number of important unresolved issues including the need for: broader formal foundation for both functional and non-functional requirements, greater degree of formality and automation, new requirements development methods, and higher level of integration in the overall design process.

A TAXONOMY OF REQUIREMENTS SPECIFICATION TECHNIQUES

GRUIA-CATALIN ROMAN

WUCS-84-6

DEPARTMENT OF COMPUTER SCIENCE
WASHINGTON UNIVERSITY
SAINT LOUIS, MISSOURI 63130

June 1984

ABSTRACT

A taxonomy is introduced and used as a backdrop against which current state-of-the-art in the requirements engineering field is reviewed. The emphasis is on identifying general trends and issues rather than offering the reader a literature survey. The contents of a requirements specification is presented in light of the consensus reached by both theoreticians and practitioners. The desirable properties of a requirements specification are justified from a functionalist viewpoint and it is suggested that changes in the way one uses the requirements may alter the relative significance of different properties. Finally, the classification of requirements specification techniques is approached from a total system design perspective. The paper shows that, despite significant growth, the requirements area still faces a number of important unresolved issues including the need for: broader formal foundation for both functional and non-functional requirements, greater degree of formality and automation, new requirements development methods, and higher level of integration in the overall design process.

TABLE OF CONTENTS

1. INTRODUCTION

Recent years have been marked by an increased interest in the area of system requirements specification. This is due to the realization that in the absence of an accurate statement of purpose the designer may solve the wrong problem, a state of affairs which often leads to disastrous consequences for all parties involved. The economic realities of system development are such that discrepancies between the delivered system and the needs it must fulfill may cost in excess of a 100 times what would have been required if the errors were discovered during the initial problem definition and, in some extreme cases, they may even render useless the entire system [BOEH81].

The concept of requirements specification, however, is much more general than one might be lead into believing by looking at its most common interpretation--a definition of user needs. As a matter of fact top-down design may be viewed as a recursive application of a paradigm in which one starts with a requirements specification for some component and proceeds to construct a design specification for it. Included in the design specification are the requirements for a set of subcomponents that will have to be designed next.

```
 ----------             ----------------
 |        |             | Banking      |
 | Teller | <-------->  | Operations   |
 | Model  |             | Model        |
 |        |             |              |
 ----------             ----------------

BANKING SYSTEM REQUIREMENTS
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
BANKING SYSTEM DESIGN

 ----------             ----------------            ----------------
 |        |             | Terminal     |            | Database     |
 | Teller | <-------->  | Management   | <--------> | Management   |
 | Model  |             | Subsystem    |            | Subsystem    |
 |        |             | Requirements |            | Requirements |
 ----------             ----------------            ----------------
```

Figure 1: From Requirements to Design--A Recursive Paradigm.

In the simplest terms, the distinction between requirements and design specifications may be reduced to the difference between stating WHAT functions must be provided by some component and stating HOW these functions must be carried out. One immediate consequence of this fact is that requirements specifications must facilitate ease of understanding while design specifications must enable faithful rendering of physical and logical structures needed to realize the requirements.

The use of a component's design as an implicit statement of its requirements is generally not a good practice. Even for simple and well understood functions (e.g., sorting), the design may turn out to be exceedingly complex when demanding design constraints are applied (e.g., sorting in linear time). The ensuing loss of requirements traceability and separability may cause serious maintenance problems--one can not tell if a particular function is required or is a consequence of some design constraint which is no longer significant.

The purpose of this article is to bring about a greater awareness of several important issues concerning requirements specifications: (1) the role they play in the context of the full system development life-cycle, (2) the diversity of forms they assume, and (3) the problems we continue to face in this critical area. The discussion will concentrate on the means for expressing the requirements rather than ways of generating them. This will assure a clean separation from design specification concerns which are outside the scope of this article. The presentation starts with a brief review of requirements specification contents and concerns. It is followed by a discussion of various classification criteria for existing requirements specification techniques. The discussion provides the backdrop against which the requirements specification issues are being highlighted.

2. REQUIREMENTS SPECIFICATION CONTENTS

The requirements specification serves as the acceptability criteria against which any proposed realization of some component is judged. This section reviews the kinds of information that ought to be included in a requirements specification document independent of the nature of the component for which the requirements are written. The "component" may be a whole system, a software package or a hardware device.

As shown by Yeh [YEH82], among others, requirements fall into two general categories: functional and non-functional. (The latter are also called constraints.) The functional requirements capture the nature of the interaction between the component and its environment. The non-functional requirements restrict the domain of acceptable realizations by placing constraints on the types of solutions one might consider.

2.1. Functional Requirements

The construction of the functional requirements involves modelling the relevant internal states and behavior of both the component and its environment. Balzer and Goldman [BALZ79] have noted that the model, often called a <u>conceptual model</u>, must be cognitive in nature, i.e., it must involve concepts relevant to the milieu in which the component is used and should not include concepts related to its design or implementation.

The conceptual model is incomplete unless the environment with which the component interacts is also modeled. There are several reasons for this. First, if the environment is not well understood it is unlikely for the requirements, as specified, to reflect the actual needs the component must fulfill. Second, when the boundary between the component and its environment is flexible, the component complexity may be reduced by imposing certain constraints on the way the environment is expected to behave. A user, for instance, may be required to save periodicly the file being edited although the editor could be designed to be fool proof--convenience is traded for improved performance or reduced development cost. Third, the behavior of the environment is a significant factor affecting the complexity of the component design. A real-time system, for instance, is significantly less complex when the environment is known to generate stimuli at precise points in time as compared with the case when the interval between succesive stimuli is arbitrary.

Aside from defining the functional validation criterion for the component design, the conceptual model is also an important vehicle for communication between designers and users, in the problem definition stage, and between designers, during the remainder of the development life-cycle. The importance of the conceptual model increases with the complexity of the task, the risk factors associated with the project and the number of people involved.

2.2. Non-functional Requirements

The degree of complexity associated with a particular design is also determined by the nature of the non-functional requirements that must be satisfied. A constraint such as high reliability, for instance, may raise significantly both the cost of the system and the level of effort associated with its design and testing.

Additional complications stem from the fact that formal specification of the non-functional requirements is difficult to accomplish. First, some constraints (e.g., response to failure) are related to possible design solutions which are not known at the time the requirements are written while others (e.g., human factors) may be determined only after complex empirical evaluations. Second, many constraints (e.g., maintainability) are not formalizable given current state-of-the-art. Third, not all constraints are explicit. As in the case of other engineering disciplines, a software or hardware designer is expected to follow certain generally accepted rules of the trade without having them explicitly stated. Actually, one may even be justified to speculate these days that a successful law suit could be brought against any company that delivers on some contract a large monolithic program. Finally, there is a great diversity of types of non-functional requirements.

A simple classification of the non-functional requirements provides sufficient evidence of their great diversity and of the difficulty involved in developing a unified and comprehensive theory on which to base some formal specification technique. We separate the non-functional requirements into six main categories:

- interface constraints
- performance constraints
- operating constraints
- life-cycle constraints
- economic constraints
- political constraints

Interface constraints deal with the precise definition of the means of interaction between the component and its environment. In the case of some application program, for instance, the environment may consist of system users, the operating system, the hardware and a software package. The functional requirements for this program must capture the demands and services associated with each one of these environmental entities but not the syntax of the procedure invocations, the interrupt addresses or the screen format. These details are interface constraints that should not affect what the program does but the way it does it.

Performance constraints are usually separated into three categories: time/space bounds, reliability and security. We add to these three survivability. The first category covers requirements concerning response time, workload, throughput, available storage space, etc. It is our expectation that user oriented measures such as productivity will also become increasingly important in the definition of requirements for systems that provide direct production support. Reliability constraints deal with both the availability of physical components and the integrity of the information maintained or supplied by some component. Similarly, security constraints span physical considerations such as emission standards and logical issues such as permissible information flows (e.g., for secure operating systems) and information inference (e.g., from statistical summaries about the database contents). Survivability is a requirement associated not only with defense systems but also with every day data processing where off-site copies of the database are kept to prevent loss in case of fire.

Operating constraints include physical constraints (e.g., size, weight, power, etc.), personnel availability, skill level considerations, accessibility for maintenance, environmental conditions (e.g., temperature, radiation, etc.), spatial distribution of components, etc.

Life-cycle constraints fall into two broad categories: those that pertain to qualities of the design and those that impose limitations on the development, maintenance and enhancement process. In the first group we include maintainability, enhanceability, portability, flexibility, reusability of components, expected market or production life span, upward compatibility, integration into a family of products, etc. Failure to satisfy any of these constraints may not compromise the initial delivered component but may result in increased life-cycle costs and an overall shorter life for the component. In the second group we place development time limitations, resource availability and methodological standards. The latter include design techniques, tool usage, quality assurance programs, programming standards, etc.

Economic constraints represent considerations relating to immediate and long term costs. They may be limited in scope to the component at hand (e.g., development cost) but, most often, they involve global marketing and production objectives. A high life-cycle cost may be accepted in exchange for some other tangible or intangible benefits.

Political constraints deal with policy and legal issues. A company's unwillingness to use a competitor's device and the obligation to use a certain percentage of indigenous equipment in some foreign country illustrate the type of issues falling in this category of non-functional constraints.

3. REQUIREMENTS VERSUS PRODUCT DOCUMENTATION

The product documentation is a statement of those aspects of the component that must be known by anyone wanting to use it. As such, a product documentation differs from a requirements specification primarily in terms of the readership it addresses. The functional requirements are identical but only a subset of the non-functional requirements may be relevant to the component user. The typical subset includes the interface, performance and operating constraints.

The similarity between the two types of specifications suggests that essentially the same techniques may be used for both. This holds the promise for significant improvements in the quality and uniformity of software product documentation. The approach is economically attractive (the requirements ought to be developed anyway) and should appease the current dissatisfaction with software product documentation which is generally characterized by incomplete functional descriptions and the absence of any performance data.

The relevance of requirements specification techniques for product documentation appears to have received little or no formal recognition in the literature but the case when the user manual is written before the system is designed, for instance, is an implicit acknowledgment of the dual role requirements can play.

4. IMPORTANT CONCERNS REGARDING REQUIREMENTS SPECIFICATION

Growing interest in the requirements specification has been accompanied by the emergence of general guidelines regarding the properties of a good specification. Our own attempt to organize and to find the motivation for these guidelines led us to adopting a functionalist viewpoint: a property of a requirements specification is desirable if it satisfies some identifiable need of the design process. This approach suggests that the way requirements are used determines the kind of properties they ought to have. Some properties are needed because the requirements must be read, others because designs must be checked against the requirements, yet others because requirements change with time during development and enhancement. Aspects that contribute to having a good requirements specification today may lose their significance in the future if the design process changes its character due to increased levels of automation or other factors.

We provide below a list of desirable requirements specification properties. The list is compiled form several sources [DUBO82, ZAVE81] and is annotated from a functionalist perspective.

Appropriateness refers to the ability of some specification to capture, in a manner which is straight forward and free of design considerations, those concepts that are germane to the component's role in the environment for which it is intended (business data processing, process control, communication hardware, etc.) Its absence may render the generation of the requirements impossible or very cumbersome.

A related property is _conceptual cleanliness_. It covers notions such as simplicity, clarity, and ease of understanding. It is needed above all because people are involved in developing and using the requirements. When the requirements are generated and used by design tools alone, conceptual cleanliness is usually sacrificed for the sake of _computational efficiency_.

Constructability deals with the existence of a systematic (potentially computer-assisted) approach to formulating the requirements. This is in recognition of the fact that the mere availability of a requirements specification formalism is not sufficient to make it useful, particularly on large problems.

Both humans and tools having to examine the requirements benefit from a _structuring_ that emphasizes separation of concerns and _ease of access_ to frequently needed information.

Precision, _lack of ambiguity_, _completeness_ and _consistency_ are important because the requirements represent the criteria against which the component acceptability is judged. Lack of precision (e.g., "large main memory") is defined as the impossibility to develop a procedure for determining if some realization does or does not meet some particular requirement. Ambiguity is present whenever two or more interpretations may be attached to a particular requirement--this is different from the case when several possible realizations are equally acceptable. A requirement specification is incomplete if some relevant aspect has been left out and is inconsistent if parts of the specification contradict one another. Both completeness and consistency require the existence of criteria against which one may evaluate the specification. While some of them may be included in the semantics of the requirements specification language others may not. This is especially difficult to accomplish when one needs to address the consistency between multiple related specifications such as a human interface prototype and a data-flow model of the functionality.

Consistency and completeness checks, the verification of the design against requirements, and other analytic activities presuppose the analyzability of the requirements by mechanical or other means. The higher the degree of formality the more likely is that requirements may be analyzed by some mechanical means thus opening the way to the use of tools.

Testability is defined as the availability of cost effective procedures which allow one to verify if the design and/or realization of some component satisfies its functional and non-functional requirements. This property is probably the most important one and, at the same time, the most difficult one to achieve. To illustrate the complexity involved in guaranteeing testability one may want to think of the difficulties associated with program verification where the code is checked against a set of assertions stated in predicate calculus. The problem of checking a system design against the system requirements is several orders of magnitude harder to solve and is complicated further by the fact that requirements may be application oriented.

Traceability and executability of the requirements are often adequate substitutes for testability. Traceability refers to the ability to cross-reference items in the requirements specification with items in the design specification. Without assuring testability, some help is thus provided to the designer in his/her effort to check that all requirements have been considered. Executability implies the possibility to construct functional simulations of the component from its requirements specification prior to starting the design or implementation. The requirements are validated by the user/designer through experimentation with the functional simulation.

Finally, in recognition of the fact that requirements are built gradually over long periods of time and continue to evolve throughout the component's life-cycle, the specifications must be tolerant of incompleteness and adaptable to changes in the nature of the needs being satisfied by the component, they must exhibit economy of expression, and they must be easily modifiable.

5. CLASSIFICATION CRITERIA

The objectives of this section are two-fold: to provide a framework useful in thinking about and in evaluating proposed techniques and to construct a cohesive picture of the current state-of-the-art thus revealing the issues facing this area today.

We propose five classification criteria which, when applied to a requirements specification technique, help one establish its position in the requirements engineering field, i.e., its domain of applicability, intrinsic limitations and basic philosophy:

- formal foundation
- scope
- level of formality
- degree of specialization
- specialization area
- development method

The remainder of the section discusses the relevance of each criterion.

5.1. Formal Foundation

Significant advances in the requirements field are determined by the strength of its theoretical foundation which is the basis for subsequent automation. A class of components for which formal functional requirements are routinely built is represented by compilers and interpreters. Their requirements are given by the syntax and semantics of the language for which they are constructed. Standard methods are available today for the definition of both syntax and semantics [PAGA82]. Of particular interest to the broader area of requirements specification are the three types of semantic models currently in use: denotational (the meaning of a program is stated as a mathematical function), axiomatic (the meaning of a program is stated by providing the axioms and inference rules needed to prove programs correct) and operational (the meaning of the program is given by the result of executing it on an abstract machine).

These models are expected to play an increasingly important role in the field. To date, they clearly influenced the work on the specification of functional requirements for individual programs [LISK79]. For pure procedures, axiomatic specifications take the form of input/output assertions and operational specifications are represented by simple and clear algorithms that perform the same function as the intended program but ignore any performance issues. (They are not intended for use by the actual program.) Abstract objects (appearing in object oriented design) may be specified by sets of axioms relating the operations permissible over each object or, operationally, by showing how each operation uses and modifies some abstract representation of the object.

A number of successful attempts have been made in the use of programming language semantic models as the basis for new requirements specification techniques but their full potential has not yet been established.

> However, a good grasp of the principles behind the semantic models for programming languages is important in any type of design activity that involves writing or interpreting requirements and more emphasis should be placed on including this kind of knowledge in the designers' training together with more traditional formal computer science.

A designer documenting an Ada (Ada is a registered trademark of the U.

S. Government, Ada Joint Program Office) package specification, for instance, could benefit to a great extent from some knowledge of how to specify abstract objects. Similarly, a designer will find that, by not understanding the operational specification concept, will be more likely to misinterpret requirements written in a language such as RSL [ALFO79] which has an operational nature.

Efforts to develop system level requirements specifications (i.e., the component is assumed to be a computer-based system) have explored a number of alternate formal foundations. Some approaches are based on the use of finite-state machines. Others emphasize dataflow or stimulus-response paths. Attempts have also been made to model system functionality as a community of communicating processes while data-oriented modeling of the requirements has been stimulated by efforts in the database area.

The use of finite-state machines offers elegance and a great degree of analizability. RLP (Requirements Language Processor) [DAVI79], for instance, treats system processing as a mapping that takes the current system state and an incoming stimulus and produces a new system state and a response. Redundancy, incompleteness and inconsistency in the definition of the finite-state machine are related to corresponding problems in the requirements specification.

Dataflow models are among the most popular in use today. The typical dataflow model consists of processing activities and data arcs showing the flow of data between the activities. Processing is triggered by the presence of data in the input queues associated with each activity. What makes dataflow attractive is the fact that it is very well suited for modelling the structure and behavior of most human organizations. SADT [ROSS77] and PSL/PSA [TEIC77] illustrate two distinct uses of dataflow. SADT is a requirements "blueprint language" that stresses accurate communication of ideas by graphical means; while PSL/PSA stresses the use of a requirements database and automated tools for the development and analysis of dataflow type requirements.

Techniques using stimulus-response paths decompose the requirements with respect to the processing that must be carried out subsequent to the receipt of each stimulus. The approach, rooted in the needs of the real-time processing, is widely known primarily due to the development of RSL [ALFO79].

The activities identified in both dataflow and stimulus-response models may be easily simulated by using communicating concurrent processes. Formally, a process is represented by a set of states and by a state transition mapping. This view is shared by all the techniques that model requirements using communities of processes. Fundamental differences occur mostly in the manner in which communication is being defined. In PAISLey [ZAVE81] asynchronous interactions are specified by means of function applications. (So called "exchange functions" allow the passing of information via their arguments and returned values). Jackson [JACK78] uses unbounded queues defined such that only one process may "write" to each queue and only one process may "read" from each queue. IORL [TELE82] provides a set of eight communication primitives that permit the establishment of communication paths and the exchange of data.

Data-oriented techniques concentrate on the specification of the system state represented by the data that needs to be maintained. Built-in data manipulation primitives are used to construct specifications for the system functionality. CSDL [ROUS79], for instance, is a conceptual schema definition language used to structure one's knowledge of the application area. Based on semantic network modelling and other techniques proven successful in the artificial intelligence field, CSDL provides a highly abstract and intuitive representations of knowledge.

The techniques we have cited are representative of the significant progress registered during the last decade but also indicative of some of the limitations of the current state-of-the-art.

> Since no technique is equally appropriate for all applications or comprehensive in its coverage of the requirements issues significant effort should be directed toward evaluating the potential of proven techniques in new contexts.
>
> Furthermore, there is a need to expand the formal foundation of the requirements area.

It is clear, for instance, that probabilistic concepts have not received appropriate attention, that logic based models are just beginning to be exploited, that axiomatic methods for dealing with the specification of behavior (event sequencing) is only a theoretical concern, etc.

5.2. Scope

The scope is defined by the type of requirements the specification technique attempts to express. Some techniques limit themselves to functional requirements, others are concerned solely with particular non-functional requirements (e.g., reliability), while others cover functionality and a selected subset of the non-functional requirements. SREM [ALFO79], for instance, falls in the last category. By employing stimulus-response paths to model the system functionality, SREM makes the formal specification of processing time constraints relatively easy.

There are two major difficulties in attempting to expand the scope of the current specification techniques.

> First, despite progress in the ability to express adequately the functionality, there are still major difficulties with the establishment of a formal foundation for most of the non-functional requirements.
>
> Second, broad integration of functional and non-functional requirements has not been accomplished.

If these issues are not addressed, requirements engineering environments will fail to achieve the level of integration and mechanization expected from them. It should be remembered that the severity of the constraints determines the complexity of the design and that much of the design evaluation effort is invested in checking if the constraints are met.

5.3. Level of Formality

The level of formality is determined by the extent to which a specification language may be understood by some machine. The typical user manual for some software package is generally characterized by a certain degree of structure but a lack of formality because it is written in a natural language. The pseudo-code possesses a somewhat higher degree of formality because the structured constructs state unambiguously the flow of control among the statements of the specification which are written in a natural language. (NOTE: Pseudo-code is generally used to develop design specifications but, in the proper context, it may serve as a language for specifying requirements. The behavior of a VLSI chip, for instance, may be specified using pseudo-code prior to starting the logic design.)

PSL/PSA [TEIC77] represents a next step toward formality. Relationships between arbitrary entities (e.g., "inputs A and B generate output C") may be formally captured without any concern as to their meaning. The burden of interpreting the meaning of the entities (e.g., A, B, and C) and of the relationships (e.g., "generate") rests with some consensus among designers. Completely formal specification techniques do exist but for highly specialized classes of problems.

> There is great need to reach increasing levels of formality. However, in the absence of proper automated tools, the designers' ability to cope with formality is a major stumbling block.

A dramatic illustration of the advantages of formal specifications comes from the database area where a number of models have been proposed [TSIC82] with the relational model being the simplest and the cleanest among them. These models are in fact requirements specifications for a class of components we call databases--they describe the desired functionality and not the way the database is implemented.

5.4. Degree of Specialization

Specialization of the requirements technique to a particular type of component is motivated in part by the desire to achieve high degree of analyzability and direct representation of the concepts used by the designer. The former helps to increase the potential for automation while the latter makes the technique easy to use. Taken to its extreme, however, specialization may lead to problems of its own. They include increased training costs, lack of flexibility, the need to validate the applicability of the technique for each new project, personnel compartimentalization, etc.

Current techniques cover the entire spectrum from domain specific, to domain sensitive, and, finally, to domain independent. An extreme case of specialization is represented by requirements techniques that address such a small class of components that automatic generation of the component from the requirements becomes possible. Problem oriented languages can be viewed as part of this category. Most system requirements specification techniques tend to fall into the category we call domain sensitive. Both RSL and PSL, for instance, are adequate for problems which fall outside their primary domains of applicability and adapt to the specifics of particular problems by means of designer defined extensions. The SADT notation is representative for the domain independent techniques. So is the use of formal logic in the definition of requirements.

Because there are merits associated with both specialization and generality, the developer of a requirements specification technique must evaluate carefully the trade-offs between the two directions.

> Ideally, one would like to have a technique that is general enough to be useful for a large number of problems (e.g., real-time control, data processing, databases, etc.) and, at the same time, is capable of defining the problem specific concepts needed (or convenient to use) in each case (e.g., internal and external events, report formats, relations, etc.).

One way this might be achieved is to have a single unified formal foundation (e.g., logic) and a single human interface style, along with problem specific concepts and semantic constraints.

5.5. Specialization Area

To understand fully the opportunities for specialization and the diversity of needs requirements specification techniques must satisfy at different points in the development life-cycle, one simply has to consider what is involved in the development of a complete system, i.e., a software/hardware aggregate. The Total System Design (TSD) Framework proposed by Roman et al. [ROMA84] separates the development of such systems into six stages:

- Problem Definition
- System Design
- Software Design
- Machine Design
- Circuit Design
- Firmware Design

Among the stages of the TSD framework the problem definition stage is distinguished in two ways: it is application oriented and it involves no design activities. Its sole purpose is to assure that a clear understanding of the problem has been achieved and a statement of the system requirements has been generated prior to the start of the system design. Because of the large number of applications that are exploiting the use of computer systems, specialized requirements specification techniques have been developed for use with this stage. Some are aimed at large classes of applications sharing some common feature (e.g., real-time processing) while others address the needs of specific application domains (e.g., geographic data processing). The growing interest in expert systems for requirements specification will, most likely, increase the pressure toward specialization. This diversification, however, will not necessarily undermine the need for highly integrated requirements engineering environments. On the contrary, the common logic foundation of such systems may actually enhance the opportunities for integration.

Requirements work is often assumed to be limited to the problem definition stage. This kind of narrow definition may be a convenient simplifying assumption if one is interested in the problem definition stage alone but, if taken seriously, it may suggest a fundamental lack of understanding of the very essence of the design process. The design of each component entails the specification of both functional and non-functional requirements for a set of subcomponents from which the component will be eventually assembled. Variability in the nature of the requirements techniques occurs along the development life-cycle due to changes in the nature of the components involved. Because a distributed system may be viewed as a set of processing nodes and communication lines, the system design stage is tasked with the development of the software and hardware requirements for each node and communication line. These requirements are subsequently used by the software and machine design stages, respectively. The same paradigm occurs also within a stage, even though (unfortunately) the requirements for some components (e.g., input/output assertions for procedures) are not always formally generated.

This state of affairs suggests that any attempt to separate requirements and design specifications is counter productive when one deals with the design stages.

> The key to across life-cycle integration of design activities rests with the ability to relate design and requirements specifications.

Since the design of a component (e.g., system) must satisfy all the functional and non-functional requirements specified for it, this means that, at design time, one needs to show that the component's requirements are satisfied as long as the subcomponents (e.g., subsystems) will meet the their requirements. When the subcomponent requirements are not included as part of the design this is impossible to do.

Consequently, design languages must have the ability to specify the requirements for the types of subcomponents they identify and must overcome the current emphasis on functionality alone by incorporating formally an increasing number of non-functional requirements. In the case of a software design language, for instance, it is not sufficient to have the ability to state the logic of a procedure using pseudo-code. One must also be able to state its requirements using pre- and post-assertions, let say. For, otherwise, little may be said of the design's correctness until all procedures are designed and, for a large system, this is a major drawback!

Furthermore, system development environments must provide the analytic power necessary to show that the requirements for some component are satisfied if the requirements for its subcomponents are met. (The requirements for the component and for its subcomponents may actually be defined using different techniques!)

5.6. Development Method

Recent years brought about a new distinguishing factor among requirements specification techniques: the development method. While the prevailing approach is to state the requirements completely before proceeding with the design, rapid prototyping has made significant gains in popularity. As recent studies [BOEH84] show, both methods have advantages and disadvantages. Rapid prototyping seems to lead to less code, less effort and ease of use while the traditional approach is characterized by better coherence, more functionality, higher robustness, and ease of integration. More importantly, these results could be interpreted as suggesting the need to use a mixed approach where one uses a subset of the requirements to develop rapid prototypes which in turn lead to further clarification and refinement of the original requirements. (Because of the relation between requirements and product documentation we consider that even pure rapid prototyping does lead to a statement of requirements. It should be noted that while requirements as such may never be generated, product documentation is still needed--at least in the form of on-line user documentation.)

Two more exotic methods are also making their beginnings in the requirements field. The first one is represented by efforts to introduce expert knowledge-based systems into the process of developing the requirements. (The 7'th International Conference on Software Engineering in March 1984, for instance, included one session on this topic.) The second one relates to defining requirements for situations where the problem is extremely ill specified (e.g., a medical diagnosis system). This situation is very common in the artificial intelligence community and the usual solution is not to specify the "functionality" but an evaluation procedure and a set of related acceptance criteria (e.g., 90% agreement with some group of experts on a predefined set of cases).

Many of the shortcomings we see in the today's approaches are due to the underlying assumptions being made about how requirements ought to be developed.

> While current strategies tend to structure the requirements specification process, it is hoped that future requirements development stategies will provide instead a milieu for reasoning about the problem at hand.

However, the systematic investigation of new requirements development strategies has just been started and its full impact remains still to be determined.

6. CONCLUSIONS

Although it is our hope that the taxonomy introduced in this paper--in conjunction with an increasing number of important empirical evaluations [BOEH84, SCHE84]--will contribute to crystalization of current knowledge of requirements specification techniques, we used it primarily as a backdrop against which we reviewed the current state-of-the-art in the requirements engineering field. The emphasis has been on identifying general trends and issues rather than offering the reader a complete literature survey. We have looked at the contents of a requirements specification in light of the consensus reached by both theoreticians and practitioners. The desirable properties of a requirements specification have been justified from a functionalist viewpoint and it has been suggested that changes in the way one uses the requirements may alter the relative significance of different properties. Finally, the classification of requirements specification techniques has been approached from a total system design perspective.

The paper has shown that, despite significant growth, the requirements area still faces a number of important unresolved issues and suffers form a lack of crystalization. The formal foundation of the field must be broaden by evaluating the capabilities of different types of formalisms (e.g., logic, probability theory, etc.). A theoretical foundation for the specification of non-functional requirements still needs to be established. The degree of formality must be increased in order to reach greater levels of automation. The designers' abilities to deal with formality must be enhanced through proper training and new forms of automation that take into consideration the human factor and incorporate more domain specific concepts in the requirements. New methods for developing requirements specifications must be considered. A major integration effort must be undertaken for the purpose of establishing a unified formal foundation that could bring together application and design oriented specifications, functional and non-functional requirements, the life-cycle phases, and design and requirements definition activities.

Work on requirements specification techniques must overcome the current conceptual fragmentation of the field. This requires the emergence of a consensus with regard to a growing number of issues, the development of an understanding about what to be expected from the use of a particular technique in some given set of circumstances, the refinement of current evaluation methods, and the development of highly integrated design/requirements engineering facilities.

7. REFERENCES

[ALFO79] Alford, M., "Requirements for Distributed Data Processing Design," Proc. 1'st Int. Conf. on Distributed Computing Systems, pp. 1-14, October 1979.

[BALZ79] Balzer, R. and Goldman, N., "Principles of Good Software Specification and Their Implications for Specification Languages," Proc. Specifications of Reliable Software Conf., pp. 58-67, April 1979.

[BOEH81] Boehm, B. W., Software Engineering Economics, Prentice-Hall, Inc., 1981.

[BOEH84] Boehm, B. W., Gray, T. E. and Seewaldt, T., "Prototyping vs. Specifying: A Multi-Project Experiment," Proc. of 7'th Int'l Conf. on Soft. Eng., pp. 473-484, March 1984.

[DAVI79] Davis, A. M. and Rauscher, T. G., "Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specifications," Proc. Specifications of Reliable Software Conf., pp. 15-35, April 1979.

[DUBO82] Dubois, E., Finance, J. P. and Van Lamsweerde, A., "Towards a Deductive Approach to Information System Specification and Design," International Symposium on Current Issues of Requirements Engineering Environments, Y. Ohno editor, pp. 23-31, OHM/North-Holland Pub. Co., 1982.

[JACK78] Jackson, M. A., "Information Systems: Modelling, Sequencing and Transformations," Proc. of 3'th Int'l Conf. on Soft. Eng., pp. 72-81, May 1978.

[LISK79] Liskov, B., and Berzins, V., "An Appraisal of Program Specifications," Research Directions in Software Technology, P. Wegner, editor, MIT Press, pp. 276-301, 1979.

[PAGA82] Pagan, F. G., Formal Specification of Programming Languages: A Panoramic Primer, Prentice-Hall, Inc., 1982.

[ROMA84] Roman, G.-C., Stucki, M. J., Ball, W. E. and Gillett, W. D., "A Total System Design Framework," Computer 17, No.5, pp. 15-26, May 1984.

[ROSS77] Ross, D. T., "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Trans. on Soft. Eng. SE-3, No. 1, pp. 16-34, January 1977.

[ROUS79] Roussopoulos, N., "CSDL: A Conceptual Schema Definition Language for the Design of Data Base Applications," IEEE Trans. on Soft. Eng. SE-5, No. 5, pp. 481-496, September 1979.

[SCHE84] Scheffer, P. A., Stone, III, A. H. and Rzepca, W. E., "A Large System Evaluation of SREM," Proc. of 7'th Int'l Conf. on Soft. Eng., pp. 172-180, March 1984.

[TELE82] "IORL Version 2 Language Reference Manual," Teledyne Brown Engineering, December 1982.

[TEIC77] Teichroew, D. and Hershey, III, E. A., "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Trans. on Soft. Eng. SE-3, No. 1, pp. 41-48, January 1977.

[TSIC82] Tsichritzis, D. C. and Lochovsky, F. H., Data Models, Prentice-Hall, Inc. 1982.

[YEH82] Yeh, R. T., "Requirements Analysis--A Management Perspective," Proc. of COMPSAC'82, pp. 410-416, November 1982.

[ZAVE81] Zave, P. and Yeh, R. T., "Executable Requirements for Embedded Systems," Proc. of 5'th Int. Conf. on Soft. Eng., pp. 295-304, March 1981.