A Taxonomy of Current Issues in Requirements Engineering

Gruia-Catalin Roman
Washington University, St. Louis

The growing area of requirements specification still needs a broader formal foundation, more automation, new development methods, and a higher level of integration into the overall design process.

In the simplest terms, the design process consists of three activities: the identification of a need, the development of a solution, and the implementation of the solution. Requirements and design specifications describe the engineer's perception of a need and his understanding of the solution. Requirements specifications state the desired functional and performance characteristics of some component independent of any actual realization, while design specifications describe the component's real internal structure and behavior. While requirements specifications facilitate understanding, design specifications faithfully render physical and logical structures that implement the requirements.

A component's design is not necessarily an implicit statement of its requirements. Even for simple and well understood functions, for example, sorting, the design can turn out to be unexpectedly complex when demanding design constraints are applied, for example, sorting in linear time. The ensuing loss of requirements traceability and separability can cause serious maintenance problems—one cannot tell if a particular function is essential or if it is simply a consequence of some design constraint that is no longer significant. Furthermore, without the requirements specification's explicit statement of purpose the designer may solve the wrong problem, a state of affairs that often leads to disastrous consequences.

The economic realities of large systems development, in particular, are such that discrepancies between the delivered system and the needs it must fulfill may cost in excess of 100 times what would have been required if the errors were discovered during the initial problem definition; in some extreme cases, discrepancies may make the entire system useless.[1] For this reason, recent years have been marked by an increased general interest in requirements specification.

The purpose of this article is to increase awareness of several requirements specifications issues: (1) the role they play in the full system development life cycle, (2) the diversity of forms they assume, and (3) the problems we continue to face. The article concentrates on ways of expressing requirements rather than ways of generating them. A discussion of various classification criteria for existing requirements specification techniques follows a brief review of requirements specification contents and concerns.

Requirements specification contents

As shown by Yeh,[2] 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 types of solutions one might consider. Certain kinds of information ought to be included in a requirements specification document independently 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.

Functional requirements. The construction of the functional requirements involves modeling the relevant internal states and behavior of both the component and its environment. Balzer and Goldman[3] have noted that the model, often called a *conceptual model,* must be cognitive in nature, that is, its concepts are relevant to the milieu in which the component is used and not related to its design or implementation. Aside from defining the functional validation criterion for the component design, the conceptual model also helps designers communicate among themselves and with users.

The conceptual model is incomplete unless the environment with which the component interacts is also modeled. If the environment is not well understood, it is unlikely that the requirements, as specified, will reflect the actual needs the component must fulfill. Moreover, since the environment affects the complexity of the component design, constraining the environment can reduce component complexity.

Non-functional requirements. Design complexity is also determined by the nature of the non-functional requirements. 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. Unfortunately, formally specifying the non-functional requirements is difficult. Some constraints (e.g., response to failure) are related to design solutions that are not known at the time the requirements are written. Others (e.g., human factors) may be determined only after complex empirical evaluations. Many constraints (e.g., maintainability) are not formalizable given the current state of the art, and many others are not explicit. Finally, there is a great diversity of types of non-functional requirements, as the following taxonomy shows.

Interface constraints define the ways the component and its environment interact. In some application programs, for instance, the environment may consist of the system users, the operating system, the hardware, and the software packages. The functional requirements for these programs 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 latter details are interface constraints that should not affect what the program does, but the way it does it.

Performance constraints cover a broad range of issues dealing with time/space bounds, reliability, security, and survivability. The first category covers response time, workload, throughput, and available storage space; we expect 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 processing where off-site copies of the database 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.), and spatial distribution of components.

Life-cycle constraints fall into two broad categories: those that pertain to qualities of the design and those that limit 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.

Even for simple and well understood functions, the design can turn out to be unexpectedly complex.

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 or its obligation to use a certain percentage of equipment indigenous to some foreign country illustrate issues falling in this category of nonfunctional constraints.

Requirements specification concerns

Growing interest in requirements specification has been accompanied by the emergence of general guidelines regarding the properties of a good specification. Our own attempt to organize and evaluate these guidelines led us to adopt 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 from several sources[3,4] and is annotated from a functionalist perspective.

Appropriateness refers to the ability of the specification to capture, in a manner that is straightforward and free of implementation 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.). An inappropriate specification technique makes requirements operation cumbersome or impossible.

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 tools alone, conceptual cleanliness is often sacrificed for the sake of enhancing the *computational efficiency* of the automation tools.

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

When the requirements are generated and used by tools alone, conceptual cleanliness is often sacrificed for the sake of enhancing the computational efficiency of the automation tools.

Both humans and tools that have to examine the requirements benefit from *structuring* that emphasizes separation of concerns and from *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. Lack of ambiguity is present whenever two or more interpretations cannot be attached to a particular requirement—this is different from the case when several possible realizations are equally acceptable. A requirements specification is complete if some relevant aspect has not been left out and is consistent if parts of the specification do not 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. Formal definition of completeness and consistency is especially difficult to accomplish when multiple related specifications such as a human interface prototype and a dataflow model of the functionality are involved.

Completeness and consistency 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 it 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 that allow one to verify if the design and/or realization of some component satisfies its functional and nonfunctional 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.

Traceability of the requirements is often used as a substitute 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 effort to check that all requirements have been considered.

Executability is the extent functional simulations of the component can be constructed from its requirements specification prior to starting the design or implementation; it plays an important role in requirements validation.

Finally, in recognition 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 temporary 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.*

Classification criteria

The main objective of this section is to identify the key issues facing the requirements engineering field today. To help organize the presentation, we separate the discussion into five parts, each corresponding to a criterion that can be used to classify existing requirements specification techniques. The selected criteria have engineering relevance leading to meaningful technical comparisons. We illustrate each criterion by discussing how it might be applied to current specification techniques. For a complete survey of the area, we recommend Ramamoorthy and So's paper on the subject.[5] (The classification criteria below are a superset of those used by Ramamoorthy and So.)

(1) *Formal foundation.* The theory that forms the basis for the particular technique (e.g., dataflow, logic, etc.);

(2) *Scope.* The type of requirements addressed by the technique (e.g., functionality, reliability constraints, etc.);

(3) *Level of formality.* The extent to which the specification could be understood by a machine without need for human interpretation;

(4) *Degree of specialization.* The size of the class of problems for which the application of the technique is appropriate;

(5) *Specialization area.* The class of components whose requirements may be conveniently specified through the use of the technique; and

(6) *Development method.* The approach used to construct the specification.

We now discuss the relevance of each criterion and the requirements engineering issues pertaining to it.

Formal foundation. Significant advances in the requirements field are determined by the strength of its theoretical foundation, which is the basis for subsequent automation, and by the extent to which the theoretical results make their way into engineering practice. Efforts to develop practical tools for the specification of system-level requirements, for instance, have exploited a number of alternative formal foundations, including the use of finite-state machines, dataflow, stimulus-response paths, communicating concurrent processes, functional composition, and data-oriented models.

Finite-state machines. The use of finite-state machines offers elegance and a great degree of analyzability. Requirements Language Processor,[6] for example, 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. 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. SADT[7] and PSL/PSA[8] 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. What makes dataflow attractive is that it is very well suited for modeling the structure and behavior of most human organizations. For this reason, new dataflow-based methods, for example, CORE,[9] are still being proposed and evaluated.

Stimulus-response paths. 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 SREM.[10]

Efforts to develop practical tools for the specification of system-level requirements have relied on a number of formal devices.

Communicating concurrent processes. 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 use communities of processes to model requirements. Fundamental differences occur mostly in the manner in which communication is being defined. In PAISLey, asynchronous interactions are specified by means of function applications.[11] (Pairs of "exchange functions" return as values each others arguments.) Jackson uses unbounded queues defined such that only one process may "write" to each queue and only one process may "read" from each queue.[12] IORL provides a set of eight communication primitives that permit the establishment of communication paths and the exchange of data.[13]

Functional composition. The functional composition approach has been promulgated by the Higher Order Software methodology,[14] and is now supported by a tool called USE.IT that allows one to define (graphically) the system's functionality as a composition of mathematical functions.

Data-oriented models. Data-oriented techniques concentrate on the specification of the system state represented by the data that needs to be maintained. The Conceptual Schema Definition Language for instance, helps structure one's knowledge of the application area.[15] Based on semantic network modeling and other techniques proven successful in the artificial intelligence field, CSDL provides highly abstract and intuitive representations of knowledge. While in CSDL the system functionality is defined in terms of built-in data manipulation primitives, other techniques (e.g., see work by Greenspan et al.[16]) provide the means to define system activities in a manner similar to that of defining data objects.

Transferring techniques among different fields shows that important benefits can be derived from evaluating proven techniques in new contexts. There are other similar success stories. Modular programming, for instance, has benefited greatly from the work on programming language semantics.

Compilers and interpreters represent a class of components for which formal functional requirements are routinely built. 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.[17] 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.[18] For pure procedures,

There is a wealth of formal models that have not yet made their way into requirements engineering practice.

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. (These algorithms 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.

New system requirements specification techniques have been successfully based on programming language semantic models, but the full potential of the new techniques 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. In general, designers can benefit from being trained how to exploit existing theoretical results in an engineering setting. A designer documenting an Ada package specification, for instance, can benefit from some knowledge of how to specify abstract objects. Similarly, a designer, by not understanding the operational specification concept, will be more likely to misinterpret requirements written in a language such as RSL (used by SREM), which has an operational nature.

The problem of applying existing theory to practice is by no means solely an education issue. As past surveys of the area show, there is a wealth of formal models that have not yet made their way into requirements engineering practice. It is clear, for instance, that probabilistic concepts have not received appropriate attention, that logic-based models are just beginning to be exploited, that most distributed processing models are still a primarily theoretical concern, etc. The single most important reason why this situation continues to prevail is the high investment required for developing and evaluating a production-version tool. Even for older tools, systematic empirical evaluation such as the one to which SREM was recently subjected[19] are more the exception than the rule.

Finally, we must point out that despite the broad body of formal knowledge that is not being applied, there is still a need to expand the formal foundation of the requirements area. No technique is equally appropriate for all applications or comprehensive in its coverage of the requirements issues. In our own work, for instance, we are currently exploring the use of formal logic in the specification of geographic data-processing requirements, a highly specialized area that, by and large, has been ignored as far as requirements specification techniques are concerned.

Scope. 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, 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 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. We should remember that the severity of the constraints determines the complexity of the design and that much of the design evaluation effort is invested in checking whether the constraints are met.

Level of formality. The level of formality is the extent to which a specification language may be understood by some machine. The typical user manual for a software package has a certain degree of structure, but lacks formality because it is written in a natural language. PSL/PSA 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. Interpreting the meaning of the entities (e.g., *A, B,* and *C*) and of the relationships (e.g., "generate") depends on some consensus among designers. Completely formal specification techniques do exist but they are for highly specialized classes of problems. We need to reach increasing levels of formality. However, without proper automated tools, the designer's ability to be more formal is limited.

A dramatic illustration of the advantages of formal specifications comes from the database area where a number of models have been proposed,[20] with the relational model being the simplest and the cleanest among them. These models are in fact requirements specifications for the class of components we call databases; they describe the desired functionality and not the way the database is implemented.

Degree of specialization. Specilization of the requirements technique to a particular type of component increases the technique's analyzability and makes the designer's concepts more susceptible to direct representation. The former helps to increase the potential for automation, while the latter makes the technique easy to use.

Broad integration of functional and non-functional requirements has not been accomplished.

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 that 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 of 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 engineering environment must evaluate carefully the tradeoffs between the two. The ideal technique 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, 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.) To achieve this, the environment would have to provide the designer with a user-expandable language suitable for the definition of problem-specific concepts and semantic constraints and a tool set sharing a single unified formal foundation and a single human interface style.

Specialization area. To understand fully the opportunities for specialization and the diversity of needs that 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, that is, a software/hardware aggregate. The Total System Design framework proposed by Roman et al.[21] separates the development of such systems into six stages:

- Problem definition,
- System design,
- Software design,
- Machine design,
- Circuit design, and
- 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.

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.

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

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 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 they 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, 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.

The TSD framework goes one step further. It suggests that, for very high performance systems where the design of the software and of the hardware must be tightly coordinated, the designer must have the ability to define the interdependency between software and hardware requirements. In a recently published paper, we provide an example of how this might be accomplished.[22]

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 show, both methods have advantages and disadvantages.[23] 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 that in turn lead to further clarification and refinement of the original requirements. (Note: We consider that even pure rapid prototyping does lead to a statement of requirements. While written requirements specifications may never be generated, product documentation is still needed.)

Within these two broad categories further distinctions may be considered. They would have to include the type of human interface (e.g., linguistic versus graphic), the degree of automation, the problem decomposition or composition rules employed in order to control complexity, etc. Although space limitations do not permit us to discuss these issues in any detail, it must be said that the industrial success of a specification technique is heavily dependent upon its treatment of human factors, that is, the concepts it makes available and the interface style it supports.

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 Seventh International Conference on Software Engineering in March 1984, for instance, included one session on this topic.) The second method defines 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 percent agreement with some group of experts on a predefined set of cases).

Many of the shortcomings we see in 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, future requirements development strategies might 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.

Despite significant growth, the requirements area still faces a number of important unresolved issues and suffers from a lack of crystalization. The formal foundation of the field must be broadened 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 designer's 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 requirements definition and design activities.

Work on requirements specification techniques must overcome the current conceptual fragmentation of the field. This requires the emergence of a consensus on what is to be expected from the use of a particular technique in a given set of circumstances, the refinement of current evaluation methods, and the development of highly integrated design/requirements engineering facilities. □

References

1. B. W. Boehm, *Software Engineering Economics,* Prentice-Hall, Inc., 1981.
2. R. T. Yeh, "Requirements Analysis—A Management Perspective," *Proc. COMPSAC '82,* Nov. 1982, pp. 410-416.
3. R. Balzer and N. Goldman, "Principles of Good Software Specification and Their Implications for Specification Languages," *Proc. Spec. Reliable Software Conf.,* Apr. 1979, pp. 58-67.
4. B. Liskov and S. Zilles, "An Introduction to Formal Specifications of Data Abstractions," *Current Trends in Programming Methodology—Vol. I: Software Specification and Design,* R. T. Yeh, ed., Prentice-Hall, 1977, pp. 1-32.
5. C. V. Ramamoorthy and H. H. So, "Software Requirements and Specifications: Status and Perspectives," *Tutorial: Software Methodology,* C. V. Ramamoorthy and R. T. Yeh, eds., IEEE Computer Society, 1978, pp. 43-164.
6. A. M. Davis and T. G. Rauscher, "Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specifications," *Proc. Spec. Reliable Software Conf.,* Apr. 1979, pp. 15-35.
7. D. T. Ross, "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Trans. Soft. Eng.,* SE-3, No. 1, Jan. 1977, pp. 16-34.
8. D. Teichroew and E. A. Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Trans. Soft. Eng.,* SE-3, No. 1, Jan. 1977, pp. 41-48.
9. G. P. Mullery, "CORE—A Method for Controlled Requirements Specification," *Proc. Fourth Int'l Conf. Soft. Eng.,* Sept. 1979, pp. 126-135.
10. M. Alford, "Requirements for Distributed Data Processing Design," *Proc. First Int'l Conf. Distributed Computing Systems,* Oct. 1979, pp. 1-14.
11. P. Zave and R. T. Yeh, "Executable Requirements for Embedded Systems," *Proc. Fifth Int'l Conf. Soft. Eng.,* Mar. 1981, pp. 295-304.
12. M. A. Jackson, "Information Systems: Modeling, Sequencing and Transformations," *Proc. Third Int'l Conf. Soft. Eng.,* May 1978, pp. 72-81.
13. "IORL Version 2—Language Reference Manual," Teledyne Brown Engineering, Dec. 1982.
14. M. Hamilton and S. Zeldin, "Higher Order Software—A Methodology for Defining Software," *IEEE Trans. Soft. Eng.,* SE-2, No. 1, Mar. 1976, pp. 9-32.
15. N. Roussopoulos, "CSDL: A Conceptual Schema Definition Language for the Design of Data Base Applications," *IEEE Trans. Soft. Eng.,* SE-5, No. 5, Sept. 1979, pp. 481-496.
16. S. J. Greenspan, J. Mylopoulos, and A. Borgida, "Capturing More World Knowledge in the Requirements Specification," *Proc. Sixth Int'l Conf. Soft. Eng.,* Sept. 1982, pp. 225-234.
17. F. G. Pagan, *Formal Specification of Programming Languages: A Panoramic Primer,* Prentice-Hall, Inc., 1982.
18. B. Liskov and V. Berzins, "An Appraisal of Program Specifications," *Research Directions in Software Technology,* P. Wegner, ed., MIT Press, 1979, pp. 276-301.

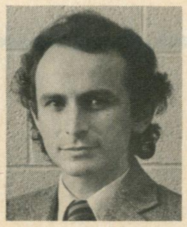
19. P. A. Scheffer, A. H. Stone III, and W. E. Rzepka, "A Large System Evaluation of SREM," *Proc. Seventh Int'l Conf. Soft. Eng.*, Mar. 1984, pp. 172-180.

20. D. C. Tsichritzis and F. H. Lochovsky, *Data Models,* Prentice-Hall, Inc., 1982.

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

22. G.-C. Roman and M. S. Day, "Multifaceted Distributed Systems Specification Using Processes and Event Synchronization," *Proc. Seventh Int'l Conf. Soft. Eng.*, Mar. 1984, pp. 44-55.

23. B. W. Boehm, T. E. Gray, and T. Seewaldt, "Prototyping vs. Specifying: A Multi-Project Experiment," *Proc. Seventh Int'l Conf. on Soft. Eng.*, Mar. 1984, pp. 473-484.



Gruia-Catalin Roman has been on the faculty of Washington University in Saint Louis since 1976; he is an associate professor in the Department of Computer Science. He is also a software engineering consultant for several firms, and he has developed custom methodologies for both private and government organizations.

His current research deals with formal specification of geographic data-processing requirements, distributed-system design models and methodologies, and the dynamics of software development environments. His previous research has been primarily in the area of methodology development and assessment. Other areas in which he has worked include formal languages, biomedical simulations, computer graphics, and distributed databases.

Roman was a Fulbright Scholar at the University of Pennsylvania, where he received a BS degree (1973), an MS degree (1974), and a PhD degree (1976), all in computer science. He is a member of Tau Beta Pi, ACM, and IEEE Computer Society. His address is Department of Computer Science, Washington University, Box 1045, Saint Louis, MO 63130.

SFUN/LIBRARY T.M.

IMSL's Natural Resource for

Evaluating Special Functions in FORTRAN Programming

Evaluation of special functions is essential in solving a wide range of mathematical and statistical problems. In physics, engineering, applied mathematics and other technical fields, reliable results often depend on accurate, verifiable calculation of special functions.

Now there is a straightforward approach to evaluating special functions in mathematical and statistical FORTRAN programming — SFUN/LIBRARY, one of IMSL's Natural Resources. SFUN/LIBRARY is a comprehensive selection of user-callable subroutines and function subprograms for use in FORTRAN program development. SFUN/LIBRARY lets you select complete, fully tested routines, with the assurance of IMSL accuracy and reliability.

SFUN/LIBRARY is the most comprehensive resource of its kind, with routines for evaluating gamma functions, Bessel functions, exponential integrals, error functions, trigonometric and hyperbolic functions, and many others. The system features independent single- and double-precision versions of routines. Both versions may be employed in the same program, allowing great flexibility in problem solving, and assuring verifiable results through cross-checking. Many functions also have complex-argument versions.

SFUN/LIBRARY is designed for ease of use, with logical, recognizable routine names and highly informative diagnostic error messages. Documentation is clear and thorough, featuring both alphabetic and key-word-in-context indexing.

SFUN/LIBRARY is available for Control Data, Data General, Digital Equipment and IBM mainframes, with the service and product support which have made IMSL a world leader in affordable technical software.

SFUN/LIBRARY could be the Natural Resource for you. To find out more, return this coupon to: IMSL, NBC Building, 7500 Bellaire Boulevard, Houston, Texas 77036 USA. In the U.S., call toll-free, 1-800-222-IMSL. Outside the U.S. (and in Texas), call (713) 772-1927. Telex: 791923 IMSL INC HOU.

Please send complete technical information about SFUN/LIBRARY.

Name ______

Dept. ______ Title ______

Organization ______

Address ______

City ______ State ______ Code ______

Area Code/Phone ______

Telex ______

Computer Type ______ CM45

IMSL®

Problem-Solving Software Systems

Reader Service Number 4