



STATUS REPORT: REQUIREMENTS ENGINEERING

We now better understand how to write relatively unambiguous, complete, consistent requirements. But we generally don't. Why is there such a wide gap between the state of the art and the state of the practice?

PEI HSIA
University of Texas at Arlington

ALAN DAVIS
University of Colorado
at Colorado Springs

DAVID KUNG
University of Texas at Arlington

In the early 1970s, researchers began thinking about how to solicit, collect, analyze, and formally specify system and software requirements. Twenty years later, although we have improved our understanding of how to write relatively unambiguous, complete, and consistent requirements, we generally don't.

For the most part, the state of the practice is that requirements engineering produces one large document, written in a natural language, that few people bother to read. Projects that do read and follow the document often build systems that do not satisfy needs. What are we doing wrong? Why is there such a wide gap between the state of the art and the state of the practice?

CRUCIAL PROCESS STEP

The quality of a software product is only as good as the process that creates it. Requirements engineering is one of the most crucial steps in this creation process. Without a well-written requirements specification, developers do not know what to build, customers do not know what to expect, and there is no way

to validate that the system as built satisfies the requirements.

Requirements engineering is the disciplined application of proven principles, methods, tools, and notations to describe a proposed system's intended behavior and its associated constraints. It includes all activities relating to the

- ◆ identification and documentation of customer and user needs,

- ◆ creation of a document that describes the external behavior and the associated constraints that will satisfy those needs,

- ◆ analysis and validation of the requirements document to ensure consistency, completeness, and feasibility, and

- ◆ evolution of needs.

The primary output of requirements engineering is a requirements specification. If it describes both hardware and software, it is a system requirements specification. If it describes only software, it is a software requirements specification. In either case, a requirements specification must treat the system as a black box. It must delineate inputs, outputs, the functional requirements that show external behavior in terms of input, output, and their relationships, and nonfunctional require-

ments and their constraints, including performance, portability, and reliability.

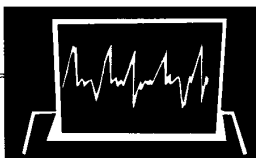
The software requirements specification should be internally consistent; consistent with existing documents; correct and complete with respect to satisfying needs; understandable to users, customers, designers, and testers; and capable of serving as a basis for both design and test.

STATE OF THE PRACTICE

Most software-development organizations agree there should be a set of activities called requirements engineering and that their success is vital to the success of the entire project.

So why is the state of the practice no better than it is? There are several reasons, not all of them obvious:

- ◆ *Requirements are difficult to uncover.* Today we are automating virtually every kind of task — some that were formerly done manually and some that have never been done before. In either kind of application, it is difficult if not impossible to uncover requirements, regardless of the techniques you use. No one can “see” a brand new system in its



entirety. Even if someone could, the description is always incomplete at the start. Users and developers must resort to trial and error to identify problems and solutions.

◆ *Requirements change.*

Because no user can come up with a complete list of requirements at the outset, the requirements get added and changed as the user begins to understand the system and his or her real needs. That is why we always have requirements changes. But, the project schedule is seldom adjusted to reflect these modifications. Fluid requirements make it difficult to establish a baseline from which to design and test. Finally, it is hard to justify spending resources to make a requirements specification "perfect," because it will soon change anyway. This is the biggest problem facing requirements engineering, and there is as yet no technology to overcome it. This characteristic is often used as an excuse to either eliminate or scale back requirements-engineering effort.

◆ *Some methods being used are inappropriate.* In their persistent quest for a silver bullet, engineers and managers are repeatedly drawn toward popular but inappropriate techniques. In the 1970s and early

1980s, many requirements engineers embraced structured analysis. But this technique, which describes interactions that occur in the real world or among software components,

does not let you visualize the overall system's behavior dynamically and tends to lead to premature design.

Now many projects are embracing object-oriented techniques. Although object-oriented techniques are practical and effective for design and coding, it is not clear what they can do for requirements engineering. Object-oriented analysis is an effective way to describe entities in the real world and their interactions, but it has yet to be demonstrated as effective for documenting a system's external behavior as a black box.

◆ *There is over-reliance on CASE tools.* Computer-aided software-engineering tools are often sold as panaceas. These exaggerated claims have created a false sense of trust, which could inflict untold damage on the software industry. CASE tools are as important to developers (including requirements writers) as word processors are to authors. However, you must not rely on requirements-engineering tools without first understanding and establishing requirements-engineering principles, techniques, and process. Furthermore you must have realistic expectations of the tools. The difference between

word processors and CASE tools is that the former are not sold as tools that will change how well you write.

◆ *Training is insufficient.*

Most software engineers know only a few requirements-engineering notations, techniques, and tools. Only a fraction of them have received the proper training to apply these to real-world problems. Additionally, very few software engineers are trained in the application domain. Hence, it is difficult for them to perform requirements engineering on a specific project. As a result, requirements-engineering notations, techniques, and tools are often misused, creating the impression that requirements engineering is neither useful nor effective and reducing confidence in the results of requirements-engineering research. Effective requirements engineering means that you must know many notations and techniques. Unfortunately, today most training emphasizes specific ones, and thus communicates know-how without teaching know-when. This is like instructing carpenters in how to use a drill press and telling them they can build any piece of furniture with it.

◆ *Project schedule is always unrealistically tight.* Because of either lack of planning or unreasonable customer demand, many projects start with insufficient time to do a decent job. Sometimes, even the allocated time is reduced while the project is underway. It is also

customary to reduce the time set apart to analyze requirements, to get started designing and coding, which frequently leads to disaster.

◆ *Developers lack confidence in requirements engineering.* Many large projects have failed in spite of using modern requirements-engineering techniques. Currently, there is no convincing evidence that modern requirements engineering improves productivity and quality, although the goal of understanding user needs early is obviously a worthy one.

◆ *Communication barriers separate developers and users.* Requirements engineering is communication-intensive. Users and developers have different vocabularies, professional backgrounds, and tastes. Developers usually want more precise specifications; users prefer natural language. Selecting either results in misunderstanding and confusion. A partial solution is to use formal models to augment, not replace, natural language.

STATE OF RESEARCH

On the basis of our investigation, we found the following research areas to have significant payoff potential. The relative time-frame during which we expect them to have major effect on practice ranges from short to long term.

Improving natural-language specifications (short-term). Almost all research in requirements engineering today deals with the introduction of more formality into the process. There is a large gap between the state of the practice (informal natural language) and the state of

BECAUSE REQUIREMENTS ARE FLUID, IT IS HARD TO ESTABLISH BASELINES.

the research (formal models). More research must be done to provide natural-language writers with insight into how they can improve their documents without formal models. Although user needs truly are difficult to determine, we still need guiding principles to reduce the ambiguity inherent in a natural-language document when it is written.

Robert Balzer has succeeded in removing some shortcomings by experimenting with partial translation from English into more formal models.¹ Barry Boehm² and Alan Davis and colleagues³ have identified the attributes that a requirements writing team should look for when they review a natural-language specification. Pei Hsia and colleagues have pioneered *scenario-based analysis*,⁴ which can augment both natural- and formal-language specifications.

Rapid prototyping and requirements animation (short-term). In the context of requirements engineering, prototyping is the construction of an executable system model to enhance understanding of the problem and identify appropriate and feasible external behaviors for possible solutions. Prototyping is an effective tool to reduce risk on software projects by early focus on feasibility analysis, identification of real requirements, and elimination of unnecessary requirements.

The most popular prototyping approaches are the throwaway approach and the evolutionary approach. Throwaway prototypes are constructed relatively quickly

and inexpensively and are discarded afterward. The advantage of this approach is rapid feedback. Evolutionary prototypes are constructed in a quality manner and are evolved iteratively into a production system. The advantage of this approach is that nothing is discarded.

As Luqi and Winston Royce have pointed out,⁵ the highest payoff will come from research improving

- ◆ languages for rapid prototype generation,

- ◆ reuse capability to make prototype generation from reusable components more feasible, and

- ◆ tools that generate user interfaces.

Rapid prototyping of software systems includes functional, user-interface, and performance aspects. Functional prototyping is usually accomplished by executable specifications, which can be automatically translated into executable prototypes. User-interface prototyping is usually accomplished by computer-aided tools that capture the intended human-machine interaction and generate code that implements such interactions. Performance prototyping is often accomplished by analytic, probabilistic, and/or simulation models that predict a design's performance and its implementation alternatives.

To improve its technology transfer, rapid prototyping must be assisted by tools for requirements analysis, design and prototype generation. Although some experiments have shown its feasibility, "a

great deal of research and development remain before this technology realizes its full potential."⁵

Requirements clustering (mid-term). Requirements clustering for incremental software delivery has shown promise.⁴ It clusters related requirements to form the subsystems that exhibit certain desired properties. For example, if you want to decompose a system into independent subsystems, you must identify all the basic and essential requirements that are vital to all subsystems before you can partition the rest of the requirements. If the purpose is to deliver software systems like highways, one segment at a time, then you must establish proper criteria for clustering sets of requirements to support such subsystems. Requirements clustering is used to define separately deliverable increments.

Case studies have demonstrated the feasibility of requirements clustering in both the structured and object-oriented paradigms. Further studies in requirements clustering should take into consideration how to deal with requirements change and how to incorporate such change into already defined increments.

Requirements-based testing (mid-term). This area poses many challenges. The first is

the ability to generate system-test plans from a requirements specification. Faced with an NP-hard problem, only heuristic approaches work. Many research efforts have attempted to address this, but none have proved viable in production environments. Perhaps one reason is that today's specifications aren't formal enough.

The second challenge is our inability to monitor how well requirements are covered during system test. The industry has many code-coverage analyzers but none for requirements coverage.

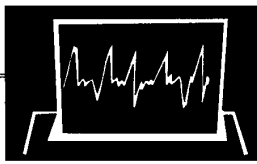
The third challenge is to develop tools that automatically maintain traceability matrices that map requirements to tests. Numerous commercially available tools help you

maintain such traces, but none can or will be able to do this automatically, and none of them are linked to program that cover requirements test.

A recent development in scenario-based prototyping suggests that an intermediate

form may be used to solve these problems. Because scenarios are the end user's perspective of a system,⁶ they can also serve as a basis for acceptance testing. An intermediate form alleviates the traceability problem by having the users identify links between scenarios and requirements, and solves the test-planning problem by using scenarios as test

WE NEED GUIDANCE ON HOW TO WRITE AN UNAMBIGUOUS NATURAL-LANGUAGE DOCUMENT.



templates for requirements. More study needs to be done to substantiate these promises.

Computer-aided requirements engineering (mid-term). Many graphical editors are now available under the CASE umbrella, most of which support a small set of requirements notations. Those that do support multiple notations provide little consistency checking or automatic translation between notations. Also with few exceptions, CASE tools are based primarily on dataflow diagrams, one of the most informal (and thus least effective) notations available. More research is needed to find CASE technology that can support multiple sets of notations, consistency checking across multiple notations, and automatic translation between them.⁷

The major advances in CASE technology during the last few years have been in the quality of the user interface and the addition of new notations with syntax-sensitive editors. The May 1990, March 1992, and May 1992 issues of *IEEE Software* provide excellent coverage of the state of the research in CASE technology.

Requirements reuse (mid-term). Much research is underway in design and code reuse. More research is needed on the advantages and the necessary methods for requirements reuse. For example, what are requirements "components,"

what makes them reusable, how can we store and retrieve them, and how do we write a requirements specification that gives us the highest probability of creating or reusing existing requirements components?

The most significant work thus far in requirements reuse came out of the 1993 IEEE International Symposium on Requirements Engineering (the proceedings are available from the IEEE CS Press). At the symposium, Alister Sutcliffe and Neil Maiden described how to retrieve requirements based on domain knowledge, and Kevin Ryan and Brian Mathews described how to use conceptual graphs to retrieve similar concepts.

Research into methods (mid-term). We need controlled experiments with systematic data collection to demonstrate that methods work. We believe that complex system development probably requires several requirements-engineering methods. To make this meaningful, we need a taxonomy of requirements notations that highlights the similarities and differences among them. Specifically, we must understand precisely how diverse notations overlap semantically,

which will make it easier to do consistency checking across, and translation among, them.⁷

Knowledge engineering (long-term). Requirements engineering is a knowledge-intensive process. Knowledge that is useful in requirements engineering includes general knowledge of requirements analysis and specification as well as application-specific, model-specific, and process-specific knowledge.

Knowledge-based requirements engineering is concerned with applying knowledge-based techniques to the problems of identifying, specifying, verifying, and validating software requirements for large, complex systems. Research challenges in this area include the acquisition, representation, and manipulation of knowledge about all of the above.

In domain modeling, requirements engineering has recently begun dealing with the use of domain-specific knowledge to derive or validate requirements specifications. Research into automating the acquisition of customers' requirements also looks promising.⁸ Automated translation of informal, natural-language specifications into more rigorous, formal specifications¹ has made some early progress.

Formal methods (long-term). Research in formal methods and their associated formal reasoning capabilities has potential for a long-term effect on requirements engineering. In many engineering fields, mathematical models and declarative languages are used in

the requirements and design phases. Formal methods have a sound mathematical basis, often given by a formal specification language.

The potential advantages of using a formal method include

- ♦ promoting a deeper understanding of the functionality and behavior of complex systems,

- ♦ promoting accurate, precise, and unambiguous requirements specifications;

- ♦ allowing rigorous analysis of system, specification, and implementation properties, such as consistency, completeness, realizability, and correctness; and

- ♦ allowing computer-aided manipulation.

The most well-known formal methods are Spec, Z, Vienna Definition Method, and Larch. Like all formal methods, these systems have a sound mathematical basis. They have been applied to specifications of medium-sized, nontrivial problems, especially the functional behavior of sequential programs, abstract data types, and hardware.⁹

Current research on formal methods has focused mainly on developing formal specification languages for representation. Manipulation is often neglected. Long-term research goals in formal methods should include developing practically applicable reasoning methods, expanding their scope and scale of application to more complex systems, and finding the means to make their representations more user-friendly. As pointed out by Jeanette Wing,¹⁰ the chal-

THE SCOPE OF FORMAL METHODS SHOULD BE BROADENED, AND THEY SHOULD BE MADE MORE PRACTICAL AND USER-FRIENDLY.

lenges for formal methods include

- ♦ specifying nonfunctional requirements such as reliability, safety, critical timing constraints, performance and human factors;
- ♦ combining different methods, such as a domain-specific method with a domain-independent method, or an informal method with a formal method;
- ♦ building more usable and more reliable tools to effectively and efficiently manipulate and analyze formal specifications;
- ♦ building reusable specification libraries to increase the productivity and quality of formal methods;
- ♦ integrating formal methods with the entire software-development process;
- ♦ demonstrating that existing techniques scale up to handle real-world problems and scaling up the techniques themselves; and
- ♦ training more people in the use of formal methods.

A unified framework (long-term). Currently no formal foundation or model for nonfunctional requirements has been found. Hence, problems in functional requirements and nonfunctional requirements have to be dealt with separately. However, piecemeal solutions do not necessarily work well overall. Therefore, we must establish a unified framework to integrate requirements-engineering principles and concepts and to avoid fragmentation.

Research in improving natural-language specification, and rapid prototyping should provide quick results and be readily transferable. However, research in knowledge engineering and formal methods requires longer term commitments and substantial effort to bring about practical results — and they may not be readily transferable to practice.

We must learn from experience that technology transfer in requirements engineering is exceptionally slow. Two avenues may be the best routes to successful technology transfer: *Incremental* changes in techniques, which provide gradual improvement and *radical* changes in techniques, which imply revolution and thus require significant evidence of worth, utility, and applicability because the risk is much higher. The requirements-engineering community has yet to be exposed to the latter case in its short 25-year life. ♦



Pei Hsia is a professor of computer-science engineering at the University of Texas at Arlington, and an associate editor-in-

chief of *IEEE Software*. His research interests include requirements engineering, software-development paradigms, rapid prototyping, languages, and computation complexity.

Hsia received a PhD in computer science from the University of Texas at Austin. He is a member of the ACM, Sigma Xi, Scientific Research Society, and Phi Beta Delta Honor Society for International Scholars, and a senior member of the IEEE.

His address is Computer Science Engineering Dept., University of Texas, PO Box 19015, Arlington, TX 76019-0015; CSnet hsia@evax.arl.utexas.edu.



Alan M. Davis is a professor of computer science and El Pomar chair of software engineering at the University of

Colorado at Colorado Springs and at the Colorado Institute for Technology Transfer and Implementation, and an associate editor of *IEEE Software*. He is the author of *Software Requirements: Objects, Foundations, and States* (Prentice-Hall, 1993) and the author or coauthor of more than 40 papers on software and requirements engineering. He is also associate editor of *Journal of Systems and Software*.

Davis received a BS in mathematics from the State University of New York at Albany, and an MS and a PhD in computer science from the University of Illinois at Urbana-Champaign. He is a senior member of the IEEE.



David C. Kung is an associate professor of computer science and engineering at the University of Texas at Arlington. His

research interests are object-oriented design and object-oriented software testing.

Kung received a BSc in mathematics from Peking University and an MS and a PhD in computer science from the Norwegian Institute of Technology. He is a member of the IFIP Working Group on Information Systems Design and Evaluation.

Address questions about this article to Hsia at University of Texas, Computer Science Dept., Box 19015, Arlington, TX 76019-0015; hsia@cse.uta.edu or Davis at University of Colorado at Colorado Springs, 1867 Austin Bluffs Pkwy, Ste. 200, PO Box 7150, Colorado Springs, CO, 80933-7150; adavis@vivaldi.uccs.edu.

ACKNOWLEDGMENTS

This report was developed out of workshops in 1991 and 1992 whose participants were members of the *IEEE Software* editorial and industry advisory boards. The requirements-engineering subgroup included the authors plus Art Hersh, Software Productivity Consortium; Gunter Koch, 2i Consult; and Shari Pfeleger, City University of London.

REFERENCES

1. R. Balzer, et al., "Informality in Program," *IEEE Trans. Software Eng.*, Mar. 1978, pp. 94-103.
2. B. Boehm, "Verifying and Validating Software Requirements and Design Specifications," *IEEE Software*, Jan. 1984, pp. 75-88.
3. A. Davis et al., "Identifying and Measuring Quality in Software Requirements Specification," *Proc. Software Metrics Symp.*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 141-152.
4. P. Hsia and A. Gupta, "Incremental Delivery Using Abstract Data Types and Requirements Clustering," *Proc. Int'l Conf. Systems Integration*, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 137-50.
5. Luqi and W. Royce, "Status Report: Computer-Aided Prototyping," *IEEE Software*, Nov. 1992, pp. 77-81.
6. W. Wang et al., "Scenario-Driven Requirements-Analysis Method," *Proc. Int'l Conf. Systems Integration*, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 127-136.
7. A. Davis, K. Jordan, and T. Nakajima, "A Canonical Representation for Requirements," tech. report, Computer Science Dept., Univ. of Colorado, Colorado Springs, 1993.
8. H. Rubenstein and R. Waters, "The Requirements Apprentice," *Proc. Int'l Conf. Systems Integration*, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 127-136.
9. V. Berzins and Luqi, *Software Engineering with Abstractions*, Addison-Wesley, Reading, Mass., 1991.
10. J. Wing, "A Specifier's Introduction to Formal Methods," *Computer*, Sept. 1990, pp. 8-24.

The First IEEE International Conference on Software Engineering is being held April 18-21, 1994, in Colorado Springs, Colorado. Address questions about the conference to Hsia or Davis.