Software Prototyping and Requirements Engineering

June 1992

Prepared for:
Rome Laboratory
RL/C3CB
525 Brooks Road
Griffiss AFB, NY 13441-4505

Prepared by:
Joseph E. Urban
Arizona State University
College of Engineering and Applied Sciences
Department of Computer Science and Engineering
Tempe, AZ 85287-5406

Distributed by:
ITT Systems & Sciences Corporation
P.O. Box 120
Rome, NY 13503-0120

Data & Analysis Center for Software
P.O. Box 120
Rome, NY*13503-0120*

DACS

The Data & Analysis Center for Software (DACS) is a Department of Defense (DoD) Information Analysis Center (IAC), administratively managed by the Defense Technical Information Center (DTIC) under the DoD IAC Program. The DACS is technically managed by Rome Laboratory (RL). Itt Systems & Sciences Corporation manages and operates the DACS, serving as a source for current, readily available data and information concerning software engineering and software technology.

Executive Summary

The field of software engineering has yet to achieve the productivity and quality gains that have been seen in the hardware counterpart field. Prototyping in more traditional engineering disciplines is the common approach for demonstrating feasibility of the functionality of a system early in the life cycle. Prototyping is also useful for risk assessment and as a means for validation of end user requirements. Software engineers began to formally recognize the benefits of prototyping software systems in the early 1980's. A decade of experience and research gives the opportunity for an assessment of the area.

Similarly, software requirements and specification techniques provide a basis for establishing early in the software life cycle the functionality of a system. The benefits of establishing a "standard" against which the end user needs are validated can have a significant impact on minimizing problems. These problems arise downstream in the life cycle when there is not a clear understanding of what is needed. In addition, the "standard" serves as a means in which a design can be validated. Executable requirements and specification techniques allow for the dynamic demonstration of functionality of a software system. Requirements engineering has a similar decade of experience and research. One area of particular interest to the software engineering community is in coupling prototyping and requirements engineering.

This report includes the motivation for using software prototyping in general and specifically in the context of requirements engineering. An overview of software prototyping covers life cycle models, approaches, pitfalls, and opportunities. The section on software requirements and specification establishes a basis for investigating techniques. The summary analyses of software requirements and specification techniques and tools for prototyping address twenty techniques across a variety of language models. Each technique summary analysis was developed to include the history, technique overview, method, supporting tools, language features, and strengths/weaknesses. The description of needed detailed analyses includes a summary of common aspects among the techniques to be developed in a repository. Software technology transfer is addressed in this report from the standpoint of past problems, avenues of opportunity, and actual experience in this area. The report ends with potential areas of future research and a summary.

TABLE OF CONTENTS

1. INTRODUCTION

This report is intended for software managers and software engineers responsible for the development of large-scale software systems. These managers and engineers have a need for information on software prototyping and requirements engineering. In order to satisfy the need for knowledge on this advanced technology, this report begins with background information regarding the importance of these topics to software development, maintenance, and management. The critical issue throughout this report is the concern for software techniques (methods) and tools (automated support) that assist in understanding "what" the software is supposed to do rather than "how" the software is implemented.

As motivation for the emergence of the report topic, in June 1990, the First International Workshop on Rapid System Prototyping was held in Research Triangle Park. North Carolina. The subtitle of the workshop, "Shortening the Path from Specification to Prototype." represents a major advantage that can be gained from using this technology in the development and maintenance of complex, large-scale software systems. Another major advantage not explicitly mentioned in the subtitle is the potential reliability gains that can come from a prototyping approach to software development. There are many approaches to software prototyping, as will be reviewed in this report. Software requirements engineering techniques and tools represent one such approach to prototyping of software systems. The technology associated with software requirements engineering can potentially provide, if used effectively, the most significant impact on improving the software development process, as well as software quality.

In addition to the Rapid System Prototyping Workshop, the IEEE International Symposium on Requirements Engineering will be held January 4-6, 1993 at Coronado Island, San Diego, California. These two technical meetings represent the maturation of the first generation of this technology and the emergence of the next generation of techniques and tools. In order to review the front-end of the software life cycle, this report addresses initially, some of the critical issues related to software development and management. This initial discussion will provide the foundation for a more detailed treatment of the area.

1.1 Problems With the Software Life Cycle

The conventional waterfall software life cycle model (or software process) is used to characterize the phased approach for software development and maintenance. Software life cycle phase names differ from organization to organization. For this report, the software process includes the following phases: requirements formulation and analysis, specification. design, coding, testing, and maintenance.

Alternative software life cycle models have been proposed as a means to address the problems that are associated with the waterfall model. One alternative

software life cycle model uses prototyping as a means for providing early feedback to the end user and developer. Definitions of prototyping vary among organizations [Carey90]. Section 2 will establish a framework for discussion.

The waterfall model allows for a changing set of means for representing an evolving software system. These documents then provide a basis for introducing errors during the software life cycle. The user often begins to receive information concerning the actual execution of the system after the system is developed. During the development of large-scale software systems, the end user, developer, and manager can become frustrated with ambiguous, missing, or changing software requirements.

1.2 Changing Software Requirements

Software techniques and tools exist for identifying ambiguous and missing software requirements. These problems are important factors in the development of any software system. However, the problems are further complicated with changing software requirements. The development length of large-scale software systems is such that changing requirements are a significant problem that leads to increased development costs. Software requirements formulation and analysis is even more difficult in complex application domains. Bera [Bera90] describes the problems with requirements engineering in the context of future fighter aircraft.

1.2.1 Management Implications

The problems that changing requirements introduce into the software life cycle are reflected in schedule slip pages and cost overruns. One argument is that more time spent upstream in the software life cycle results in less turmoil downstream in the life cycle. The more time argument is typically false when the software requirements and specification technique is a natural language. A discussion of the management implications of deriving requirements was provided in [Yeh82] and then new directions provided in [Yeh84].

1.2.2 Product Implications

The product implications are quality based aspects of the system during software development and maintenance. The requirements problems listed above have a ripple effect throughout the development of a software system. Even with this advanced technology, changing requirements are to be expected, but there would be the environment for control and discipline with the changes.

1.3 Software Developer, End User, and Project Manager Needs

Software engineers need the capability to provide early software life cycle feedback to the end user. In addition, this early feedback serves as a "standard" of the software functionality for the entire development team and management to review.

1.3.1 Rationale for Prototyping Requirements and Specifications

Software engineers need the capability to represent software requirements and specifications in a formal notation that will support analysis and execution. The notation must be amenable to transformation downstream in the software life cycle.

1.3.2 Translation of Needs Into Techniques and Tools

Given a notation with enough rigor, software engineers benefit from automated support for prototyping support to include construction, analysis for inconsistencies and incompleteness, execution for dynamic analysis, and management of complexity.

1.4 Process and Product Quality

Catchpole provided strong motivation for the development of a successful methodology or process [Catchpole86]. There has been a recent focus in the software engineering community on process and product quality. The SEI assessment methodology, Rome Laboratory quality assessment, and 6-Sigma software reliability goals are three examples that work in conjunction with prototyping requirements and specifications.

1.4.1 SEI Assessment Methodology

The Software Engineering Institute assessment methodology addresses the process aspect of software development and maintenance. Use of the technology for prototyping and requirements engineering sharpens the focus on the process. A change in the user needs can be reflected as a similar change in the formal notation used to represent the requirements and specifications.

1.4.2 Rome Laboratory Quality Assessment

This report does not address the efforts on non-functional specifications, e.g., the Rome Laboratory effort on software quality specifications. However, the natural language functional requirements of the Assistant for Software Quality developed by Rome Laboratory could be used in conjunction with a formal specification language.

1.4.3 6-Sigma Software

Reliability requirements are becoming more important and must be addressed early in the software life cycle; not deferred to a later stage of the life cycle for attention and correction. Reliability models should be coupled with the technology of prototyping and requirements engineering for assessment early on and throughout development and maintenance.

1.5 Report Overview

Section 2 is devoted to an overview on prototyping software systems. The section addresses all possible forms of software prototyping as a means for communicating with end users, developers, and managers.

Software requirements engineering techniques and tools represent one potential form of software prototyping and are described in Section 3. This approach is one that has the potential for providing the most benefits during the software life cycle. These benefits include early identification of problems in understanding user needs, developer understanding of the software system under development, and a standard against which management can track performance.

A summary analysis of twenty software requirements and specification techniques and tools is presented in Section 4. This summary analysis is intended to also serve as a survey of a broad cross-section of the technology. A template was used to describe each technique in an overview form. The section concludes with a summary section and comparisons of the techniques and tools. Problems in software technology transfer have impeded progress in the area of software requirements engineering. These problems are addressed in Section 5. The report concludes in Section 6 with the state-of-the-practice in industry and recommendations for future research.

2. PROTOTYPING SOFTWARE SYSTEMS

Ince and Hekmatpour provide excellent motivation for prototyping and raise some additional research questions [Ince87]. The field of prototyping software systems has many approaches. Rather than pass judgement on any one approach, this section overviews the approaches in an effort to set the stage for the remainder of the report which singles out the requirements engineering approach.

2.1 Software Life Cycle Models

Prototyping has been introduced throughout the conventional, waterfall software life cycle model. Two forms of life cycle models, rapid prototyping and evolutionary prototyping, have emerged around prototyping technology.

2.1.1 Conventional Model and Related Variations

The conventional life cycle model can allow for prototyping within any of the phases. Unfortunately, this approach can be difficult to control when, for example, coding of a user interface takes place during requirements formulation and analysis. Hoffman provided a methodology to aid in this area [Hoffman88].

2.1.2 Rapid Prototyping

The rapid prototyping model strives for demonstrating functionality early on the development of a system. This approach has been proposed for use in conjunction with risk management [Clapp87] under a study sponsored by the US

Air Force Electronic Systems Division. Examples of rapid prototyping were reported [Luqi88b, Luqi88c, Lea90] with notations that supported the rapid deployment of functionality.

2.1.3 Evolutionary Prototyping

In evolutionary prototyping the focus is on achieving functionality for demonstrating a portion of the system to the end user for feedback and system growth. The prototype emerges as the actual system downstream in the life cycle. As with each iteration in development, functionality is added and then translated to an efficient implementation. Also of interest is functional programming [Henderson86] and relational programming [Ceri88] as a means for accomplishing evolutionary prototyping. Methods have been described for controlling prototyping from development through maintenance which are beneficial for incorporating this new technology [Lui89a. Mayhew89].

2.2 Software Prototyping Taxonomy

A range of possibilities exists for prototyping software systems. Any form of prototyping is perceived better than not prototyping at all. Several taxonomies have been proposed and served as the basis for this discussion. The early, middle, and late prototyping taxonomy [Ratcliff88] was based on the software life cycle. Another software life cycle based prototyping taxonomy [Hooper89] included execution support as a criterion.

One form of prototyping occurs with the natural language approaches. The manual approach uses a variety of non-standard representation mechanisms and lacks, of course, automated support. Another approach is based on software requirements and specification techniques, which will be the focus of a major portion of this report. Later in the software life cycle, but still of benefit, would be prototyping based on software design techniques and high level languages. Recently, advances in knowledge based approaches to software development have emerged as a means for capturing domain analysis.

In addition to the four major approaches to prototyping, a hybrid approach looks at bridging several techniques, e.g., 'me too' [Alexander89].

2.2.1 Natural Language Approaches

Natural language serves as the primary manual approach. The ANSI/IEEE Std 830-1984 [ANSI84] provides some structure to essentially a natural language approach. The Arts [Dorfman84] approach includes software support for hierarchically structured natural language requirements. Experience with capturing requirements for six systems (five of which were process control systems) [Zucconi89] was reported from a software process standpoint with primarily a natural language approach.

2.2.2 Software Requirements and Specifications

The software requirements and specifications approach to prototyping is the basis for Sections 3 and 4. Davis provided one of the early arguments for rapid prototyping with executable software requirements and specifications [DavisA82]. Example techniques that aim at the front-end of the software life cycle though execution of functionality are 001 [Harnilton90] and Descartes [Urban90].

2.2.3 Software Design and High Level Languages

Design and coding languages have been used for prototyping. The utility of showing the end user the notation with these techniques is not a viable possibility. Example specification languages that were influenced by high level languages are Anna/TSL [Helmbold88. Luckham85. Luckham87], Gypsy [Ambler77], Larch [Guttag85] and SUSL [Belkhouche86]. Also, languages out of the mainstream general purpose programming languages have been used for prototyping, e.g., the declarative language. Lucid [Skillicorn89] and the logic programming language. Prolog [Brvant89]. Recently, HCLIE [Tsai91] was developed as a superset of Prolog for use as a requirements language. Similarly, RSF uses transition rules that map into logic programming [Degl'Innocenti90].

2.2.4 Knowledge Based Approach

The knowledge based approach has been used successfully during requirements elicitation. Luqi provides a discussion of the benefits to rapid prototyping with this approach [Luqi88d]. The key to this approach is in building sufficient application domain knowledge. An example of this approach can be found in Express [Topping87], which was developed for embedded system applications. An example of a knowledge-based approach for dataintensive applications was developed by Chen and Chou [ChenP-M88]. A software storming method combined with a knowledge-based approach was proposed in [Jordan89]. Additional knowledge-based approaches addressed in this study were [Tsai88, Tsai89, Loucopoulos89, Lee85, TsaiS-T90, Reubenstein91]. The Knowledge-Based Requirements Assistant is discussed in Section 5 with regards to domain analysis research.

2.3 Prototyping Pitfalls

Prototyping has not been as successful as anticipated in some organizations for a variety of reasons [Tozer87]. Training, efficiency, applicability, and behavior can each have a negative impact on using software prototyping techniques.

2.3.1 Learning Curve

A common problem with adopting prototyping technology is high expectations for productivity with insufficient effort behind the learning curve.

In addition to training for the use of a prototyping technique, there is an often overlooked need for developing corporate and project specific underlying structure to support the technology. When this underlying structure is omitted, lower productivity can often result.

2.3.2 Tool Efficiency

Prototyping techniques outside the domain of conventional programming languages can have execution inefficiencies with the associated tools. One prominent prototyping technique was the basis for the ESPRIT funded SETL to Ada project [Doberkat87] as an effort to provide efficient transformations. The efficiency question was argued as a negative aspect of prototyping [Prizant86].

2.3.3 Applicability

Application domain has an impact on selecting a prototyping technique. There would be limited benefit to using a technique not supporting real-time features in a process control system. The control room user interface could be described, but not integrated with sensor monitoring deadlines under this approach. Goldsack and Finkelstein provide a discussion on various unique aspects of real-time requirements [Goldsack91].

There are techniques that have been developed for specific application domains. Several applications domains for which prototyping has been used throughout the life cycle include: business [Misra88, Knoll89], computer-aided design [Gupta89. Lor91], distributed [Cheng82, Budkowski87, Cieslak89, Berzins90, Putilo91], flight control [Duke89], module interface [Archibald83, Hoffman89], operating systems [Archer90, Mosse90, Zhou90], programming languages [Herndon88. Michaelson88. Bryant89, Cordy91], software engineering environments [Kuo87], and user interface [Easterby87, Lewis89]. This report is more concerned with wide spectrum techniques. However, there is interest in the techniques listed above, even though these techniques may have narrow applicability.

2.3.4 Undefined Role Models for Personnel

This new approach of providing feedback early to the end user has resulted in a problem related to the behavior of the end user and developers. An end user with a previously unfortunate system development effort can be biased in future interactions with development teams. Hemenway and McCusker provide empirical experience with an actual prototype and users feedback [Hemenway82]. Alavi describes in [Alavi91] a university experiment on software prototyping. Boehm also describes in [Boehrn84] a university experiment on software prototyping. A change classification scheme was proposed in [Mayhew89] as a means for control based software prototyping activities.

2.4 Prototyping Opportunities

Not to prototype at all should simply not be an option in software development. The benefits of software prototyping are elaborated in [Herndon88]. The end user can not throw the software needs (stated in natural language) over the transom and expect the development team to return the finished software system after some period of time with no problems in the deliverables. Given that there must be interaction between the end user and development team, even the manual approaches described in Section 2.2.1 are the bare minimum that would be needed.

2.4.1 Existing Investment in Maintained Systems

One of the major problems with incorporating this technology is the large investment that exists in software systems currently in maintenance. The idea of completely reengineering an existing software system with current technology is not feasible. There is, however, a threshold that exists where the expected life span of a software system justifies that the system would be better maintained after being reengineered in this technology. Total reengineering of a software system should be a planned for effort rather than as a reaction to a crisis situation. At a minimum, prototyping technology could be used on critical portions of an existing software system. This minimal approach could be used as a means to transition an organization to total reengineering.

2.4.2 Adding Investment in Fully Exploiting the Technology

In many cases, an organization will decide to incorporate this advanced software prototyping technology, but the range of support for the concept varies widely. Software prototyping, as a development technique, must be integrated within an organization through training, case studies, and library development. In situations where this full range of commitment to the technology is lacking, e.g., only developer training provided, when problems begin to arise in using the technology a normal reaction of management is to revert back to what has worked in the past.

2.4.3 Developer to End User Pass Off

Finally, the end user involvement becomes enhanced when changes in requirements can first be prototyped and agreed to before any development proceeds. Similarly, during development of the actual system or even later out into maintenance should the requirements change, the prototype is enhanced and agreed to before the actual changes become confirmed. A schema was presented for validating an embedded system description from initial description to running target system [Goedicke86].

This section established prototyping as a means for improving software development and maintenance. The next section is based on selection of one of the major approaches to prototyping. An overview of software requirements and

specification technology is addressed in the next section. Also included is the description of an evaluation method for front-end software life cycle techniques and tools.

3. SOFTWARE REQUIREMENTS AND SPECIFICATIONS

The previous sections have provided a background for the problems in software engineering and software prototyping as a potential aid to these problems. Four approaches to software prototyping were discussed and the software requirements and specification approach has been singled out for further exploration in this section. Rzepka and Ohno [Rzepka85] provide strong motivation for the held of requirements engineering in a special issue of Computer which they guest edited. The motivation for using formal methods has been addressed in terms of successes in the European software engineering community [Babcock89]. Thayer and Dorfman [Thayer90] have edited an excellent tutorial of selected readings in the area. Included in the tutorial is a survey and comparison of five techniques [DavisA90].

Current software requirements and specification techniques have been designed with the goal of providing general facilities that permit usage across different domains or application areas. This approach seems to be logical, and in fact resembles the approach taken in traditional high level programming languages. However, certain application areas, such as real-time systems, have characteristics that are unique and are difficult to represent using traditional approaches. However, given this technology base, industry practice has been poor to non-existent. The role of the government can benefit this area by taking a leadership position on the technology as described in Section 5.

Six analysis methods have emerged as front runners in terms of commercially available techniques and tools. These methods and several sample techniques are: process, data, control, object, natural language, knowledge base, and assertional. The structured analysis approach has gained a large following, e.g., [Ross77a, Ross77b, Krista89, Lea90. Lintulampi90]. The object oriented method has been used extensively, e.g., Object-Oriented Requirements Specification Method [Bailin89], Object-Oriented Analysis [Coad90], Object-Oriented Specification [Jarvinen90], Object Oriented Transformation Schemas [Coomber90], and Ada Object-Based Analysis [Walers91b]. The state machine and Petri net approaches have been used primarily for real-time systems specification [Wang88, Baldassari91, Jaffe91, Schmidt91]. RT-ASLAN [Auernheimer86] and OSDN/SBOM [ChenJ89] are techniques that combine the process and object methods. There are approximately fifty techniques available on the market based on a recent Software magazine product review. NASA empirical work led to development of the HOS methodology and USE.IT software engineering environment which is now being extended as 001. RAPID/USE [Wasserman86] and TAGS [Sievert85] are other examples.

Therefore, an appropriate approach is to provide the requirements

engineer with tools that embody a conceptual model of the area of application described. The use of a conceptual model will be of benefit for both users and developers, since it will increase the communication level between participants in the software life cycle by providing concepts and mechanisms that resemble the characteristics and behavior of real world objects.

3.1 Drawing Requirements, Specifications, and Design Fencelines

The intent of this report is not to draw strict fencelines separating requirements, specification, and design. One uniform notation that could transition from end user needs to production software would be ideal. The reality is such that some techniques state only the "what" while others state the "what" along with the "how". Software quality can be improved by the use of software requirements and specification technology, which will improve the developer's ability to satisfy customer needs and create a reliable system. Early work in this area did cross fencelines, e.g. SSAGS [Payton82] and Event-Driven Methodology [Rolland83].

The primary goal of advances on the upstream effort is to reduce the cognitive gap that exists between current requirements engineering techniques and the users (whether end user, software engineer, or project manager) of such techniques. The primary goal of advances in the downstream effort is to smooth the transition to efficient, production quality software systems.

3.1.1 Evolution of the Area

The field of software requirements and specifications was primarily initiated as a research area during the 1970's. One major research funding program [DavisC77] in this area was reported in a special collection on requirements analysis that appeared in the *IEEE Transactions on Software Engineering.*

CASE products for software requirements and specification began to emerge in the 1980's. These early CASE products achieved limited success due to some of the problems identified elsewhere throughout this report. The technology has matured and become a viable CASE market. However, more research is needed to build on the current technology and provide greater capabilities in the next generation of languages.

3.1.2 Policies and Procedures

One problem with this technology is the lack of a strong set of policies and procedures endorsing the use of the technology. The ANSI/IEEE Std 830-1984 is one of the most significant efforts in the area of policies. However, 830-1984 falls somewhat short, only serving as a guide. Further work is ongoing to revise 830-1984 into a complete standard.

3.1.3 Taxonomies

Several taxonomies have been proposed for classifying software requirements and specification techniques. Roman provided a taxonomy based on the underlying model supporting a technique [Roman85]. Davis and Freeman provided a multi-faceted taxonomy that includes the underlying model [DavisA91]. This report extends this earlier work on classifying techniques with a template-based overview for each technique and identification of summary detailed analysis.

3.2 Representation Forms

In terms of representation forms software requirements and specification techniques appear in three major forms: textual, boxology, and graphical. In the textual form, formal grammars are used to define the syntax of the language. The boxology form has definitions for various symbols in the language. Finally, in the graphical form the actual objects of a system are used in the technique. All three forms have advantages and disadvantages. In some cases, multiple representation forms are used with one underlying representation form for storage and retrieval. For example, a technique could allow for textual or graphical representation, but store the requirements and specifications in textual form.

3.2.1 Textual

The textual approach uses characters based on a formally defined language to describe the requirements and specifications. This approach is an outgrowth of conventional programming languages.

3.2.2 Boxology

Two approaches have been used for the graphical form of representation. One form, referred to here as a boxology, is to define geometric symbols that have syntactic and semantic context. This approach has significance to the software engineers trained in the technology and lesser impact with the end user when viewed statically. Graphical support for the specification of complex systems has been provided in the past by several specification techniques. However, the graphics support provided by existing specification techniques is generally restricted to those boxologies that do not convey any implicit meaning for the application described. Animation of a boxology has been explored as a means to support specification understanding [Stasko90].

3.2.3 Graphical

A more recent graphical form has been driven by object oriented techniques. This form involves representing the actual objects within a system. Two graphical forms have emerged, one statically represents the objects within a system and the other form animates the objects to demonstrate the functionality. New approaches, based on the availability of improved graphics capabilities and

object-oriented programming techniques, should be incorporated in the support software for a new generation of specification techniques.

3.3 Tool Support

Early software requirements and specification techniques that appeared in the open literature did not include tool support. However, with the advent of executable specifications the tool support increased. The tool support provided for software requirements and specification techniques covers the range of a single tool to an integrated software environment. In addition, tool to tool communication with tools in other phases of the software life cycle is also an important part of the total environment. Slusky provides a classification scheme for CASE tools that support prototyping [Slusky87]. The remainder of this section is an overview of software requirements and specification tool support for: construction, analysis, execution, test data generation, documentation generation, and management.

3.3.1 Construction

The construction of requirements and specifications in a minimal setting would be through the use of a general-purpose text editor. Context-sensitive or language-sensitive editors are another means for specification construction which aid in removing syntactic errors. With the recent advancement and cost reduction of bit mapped displays, editors have become more readily available for the graphic representation forms.

3.3.2 Analysis

Analysis of software requirements and specifications typically involves syntactic, internal consistency, and logical completeness checking. Empirical studies [Hamilton90] have reported that over 75% of the errors introduced during software development can be attributed to consistency and completeness errors. Tool support through analysis of requirements and specifications can have a significant impact when compared to the implications of removing errors later in the software life cycle.

3.3.3 Execution

Not all techniques support execution of requirements and specifications. Some early techniques, which were manual, now have execution support e.g., Structured Analysis [Lea90] and Jackson System Development [Kato87]. Those techniques that have execution support, typically, use one of three forms: simulation, interpretation, or transformation. None of these approaches to execution has proven to be as efficient as hand development of code. However, the advantages gained through early life cycle feedback with the end user and developers is more significant than the execution efficiency when put in the context of large-scale systems. Also, as advances in compiler and interpreter technology occur, similar advances are expected to occur in the execution of

requirements and specifications.

The execution capability has an even more significant impact when the language and tool support execution of partial specifications. A long term goal would be for execution of requirements and specifications to be as efficient as hand developed code, thus eliminating the need for the traditional design and coding phases of the software life cycle.

3.3.4 Test Data Generation

Analysis tools for requirements and specification provide static syntactic analysis. Executable specifications provide a means to demonstrate dynamically the functionality of the specifications based on input data provided by the user. However, executable specifications provide the opportunity for the development of techniques for white box testing of the specifications based on automatic test data generation of the expected input data. The benefits of this capability have been described in the context of prototyping [Staknis90].

3.3.5 Documentation Generation

Some techniques provide the capability to automatically generate documentation from the requirements and specifications. This documentation can be in the form of user manuals, internal project documentation, and management oversight reports.

3.3.6 Management

The management part of tool support reflects the same needs for configuration management as with the design and coding products of the software life cycle. Database support is critical for the scaling up of the technology. Evaluation tools can benefit management in determining the quality of software life cycle products [Cardenas- Garcia91]. Additionally, the technology provides the basis for more effective life cycle cost estimation and scheduling.

3.4 Evaluating Software Requirements and Specification Techniques

Edmonds and Urban [Edmonds84] developed an evaluation method for software requirements and specification techniques and tools. This method is based on evaluation criteria that appeared in the open literature, Gilb's MECCA method, and McClure’s rating scheme. The evaluation method was applied to Higher Order Software's USE.IT and TRW's SREM. This method was not applied directly in this report as the method is tailored to meet the needs of an organization and due to the dynamic aspects of the evaluation method. However, many of the principles of the evaluation method were applied in developing this report and could serve as the basis for applying the evaluation method.

3.4.1 Qualitative Criteria

The criteria selected for use in the Edmonds and Urban evaluation method were divided into qualitative and quantitative criteria. The qualitative criteria were subjective in nature and so there could be variability among a group of evaluators.

As experimental techniques and tools, the language definitions and prototyping tools will undergo a number of modifications and enhancements. The modifications to techniques and tools, however, should not only be the outcome of language theory and language designer preferences. Human factors studies of end user and developer interactions is therefore required to pinpoint and evaluate the most important modifications.

The information obtained through the human factors studies could have a number of other uses. For example, the syntactic and semantic representation mechanisms could be analyzed for optimizing the understanding of the software functionality. To make the process of specifying easier for the developer, additional tool support may be provided that could be helpful in guiding the developer, and also with the functional description of the specified system. The results obtained from the human factors studies play an important role in the development of the capabilities described above, since the results of the studies will be used to validate the knowledge to be embedded within the language definition and tool support.

3.4.2 Quantitative Criteria

The quantitative criteria were objective in nature and could be measured. and measurements could be repeated consistently. In developing this report, an effort was made to strive for quantitative criteria.

3.4.3 Static Aspects

There were static ways in which criteria could be measured in the Edmonds and Urban method. The static aspects of the evaluation method meant that the criteria could be measured by reviewing the language definition or support tool documentation. This report was based on static analysis of the material available in the open literature.

3.4.4 Dynamic Aspects

The dynamic aspects of the Edmonds and Urban evaluation method are criteria that require the development of a specification and/or use of the support tools. This report did not depend on measuring the dynamic aspects of the techniques and tools.

This section addressed software requirements and specification technology from the standpoint of models. representation forms. tool support and

evaluations. The next section is a survey of software requirements and specification techniques. This survey provides summary analyses for a broad spectrum of techniques and tools.

4. REQUIREMENTS AND SPECIFICATION TECHNIQUES AND TOOLS FOR PROTOTYPING

This section is a summary description and analysis of twenty software requirements and specification techniques and associated support software. The first section is an introduction to the template that was used to overview each technique. The next twenty sections after the introduction are the overviews of the twenty selected techniques.

The techniques were selected for description and analysis in order to provide a representative cross-section of software requirements and specification technology. One criterion for selection was to include wide spectrum languages in the analysis. Another criterion was to ensure coverage of the various models used for requirements and specification language design.

This section concludes with Section 4.22 which is a description of the detailed analysis that the software engineering field needs developed and centered in a repository of information. An overview of detailed evaluation criteria is provided for application on all the available techniques and support software.

Language and tool use should be part of the evaluation process. A candidate set of problems is provided later in Section 5 for exercising the technology and providing a base of case studies. The problems have been used for demonstrating techniques that were described in the open literature.

4.1 Analysis Introduction

A template was created from the review of existing taxonomies of the field. Each technique is described using the following template: technique overview, method, supporting tools, language features, and strengths/weaknesses. The template was designed to provide an approximate one page summary section of each technique.

Each technique summary section begins with a brief history of the technique development and, if appropriate, tool support development. Development efforts cover the range from individual researcher through multi-organization. The technique overview subsection provides a one paragraph summary of the models upon which the techniques were developed. There are several techniques included in this subsection that were based on multiple models, e.g., object-oriented and data flow.

The method subsection describes the process by which a software developer produces requirements and specifications with a technique and associated support software. The support tools subsection provides a description of the software

tools used with a technique. The minimal tools for a technique would include text editors and formatters. In some cases language processors exist for providing executable prototypes of requirements. Analysis tools are available for many of the techniques described below. A few of the techniques have available a fully integrated environment of CASE tools.

The language features subsection addresses the textual, boxology, and/or graphical forms of representing syntax and semantics. The range of textual languages includes fixed format natural language through design-like and programming-like languages. A technique summary concludes with a strengths and weaknesses subsection that covers the method, tool support, and language features in terms of executable prototyping.

The following techniques were selected for analysis based on material available in the open literature: 1) 001 [Hamilton92], 2) Anna/TSL [Helmbold88, Luckham85, Luckham87], 3) ANSI/IEEE Standard 830-1984 [ANSI84], 4) Arts [Dorfman84], 5) DARTS [Gomaa84], 6) Descartes [Urban77, Urban90], 7) ENVISAGER [Diaz-Gonzalez87a, Diaz-Gonzalez87b, Diaz-Gonzalez91], 8) GSS [Harbert90], 9) Larch [Guttag85], 10) PAISLey [Zave91], 11) PLEASE [Terwilliger89], 12) PROSPER [Leszczylowski89], 13) PROTOB [Baldassari91], 14) PSDL/CAPS [Luqi88a, Luqi88c, Luqi89b, Luqi90, Luqi92], 15) SBRE [Holbrook90], 16) SLAN4 [Beichter84], 17) SPADES [Ludewig85], 18) SREM [Alford77, Alford85], 19) STATEMATE [Harel90, i-Logix, Inc.90], and 20) SUSL [Belkhouche86].

Further need for detailed analysis of all software requirements and specifications techniques is described in Section 4.22. The section includes a description of the summary and detailed analyses needed, as well as application development with published problems.

4.2 Technique 1 - 001

001 [Hamilton90] is based on empirical research for NASA on removing software errors from large-scale, real-time avionics software. This empirical research led to the commercial development of the Higher Order Software methodology and associated support software, USE.lT. 001 is an example of second generation specification technology.

Technique Overview

001 is based on the functional model and allows for the top-down development of a specification. 001 is applicable to a wide variety of application domains as the, underlying theory is based on three application independent primitive control structures: join, include, and or. An analysis capability can be applied to a specification for detection and removal of syntactic and semantic errors. A code generation capability allows for the execution of specifications. including partial specifications.

Method

The control structures were developed from a set of axioms in order to analyze specifications for internal consistency and logical completeness. The control structures are used to develop specifications. Analyzed specifications that do not contain language errors can be subjected to transformation into high level language statements.

Supporting Tools

An extensive toolset is provided that includes editor support for specification construction and modification, analysis for consistency and completeness, and code transformation through resource allocation tools. Data definition facilities and primitive library support capabilities support the development of specifications.

Language Features

The three primitive control structures are developed in a top-down, tree-structured manner to support subfunction decomposition. The control structures, join, include, and or, are used to represent one-way communication between subfunctions, independent subfunctions, and a choice in subfunctions, respectively. These control structures use rigid variable tracing rules to state function responsibilities, order, rejection, and input/output access rights. A set of four non-primitive co-control structures allows for more flexibility in the variable tracing rules than the three primitive control structures.

Strengths and Weaknesses

The earlier empirical work in support of the theory provides a basis for quantifying the impact of error prevention that can be provided with this technique. The analysis and code generation capabilities are the primary automated support strengths for validating syntactic and semantic aspects of a functional specification. The notation is easy to learn based on experience in graduate software engineering courses. Breadth and depth of minimal support libraries can have an impact on achieving rapid productivity gains.

4.3 Technique 2 - Anna/TSL

The Anna (ANNotated Ada) and TSL (Task Sequencing Language) specification languages were developed by the Program Analysis and Verification Group at Stanford University [Helmbold88, Luckham85, Luckham87]. The research effort has been supported by the Defense Advanced Research Projects Agency. Anna and TSL were developed as extensions to the Ada Programming language with tool support for an Ada software development environment.

Technique Overview

Anna and TSL specifications are developed as formal comments that appear

as regular comments in Ada programs. The Anna and TSL formal comments are in support of an assertional approach to Ada software development. The tool support allows for analysis of Anna and TSL specifications. In addition, Ada code generation from Anna and TSL specifications provide the ability for runtime checking of Ada programs against the specifications.

Method

Anna and TSL formal comments are assertions that are inserted in Ada programs for verification during software development. These formal comments are processed by Anna and TSL tools, but would be ignored by validated Ada compilers and other Ada tools which do not analyze comments. A result of this approach is that the annotations serve as documentation for Ada programs. The verification process is applied during software development to partial through complete specifications.

Supporting Tools

The support tools are part of a prototype environment for wide-spectrum languages. An extensive set of tools were developed [Luckham87] in the categories: components, integration mechanisms, support tools, and application tools. These tools cover the spectrum of specification analysis, code generation, and run-time checking. The environment is integrated with Ada through extensions to the intermediate representation mechanism of DIANA Abstract Syntax Trees.

Language Features

Anna supports the development of virtual Ada text as comments and annotations as formal comments. The annotations are assertions developed on logical and program variables and can handle quantified expressions at the level of first order predicate calculus. TSL supports the specification of interactions between Ada tasks. These task specifications are developed for detecting task deadlock and blocking.

Strengths and Weaknesses

The Anna and TSL specification languages are relatively easy to learn for Ada developers who are familiar with assertions in program verification. These specification languages can have a significant impact on the development of reliable Ada software. The resulting documentation from using the specification languages will support ease of maintenance. This research project is a continuing effort and the more stable prototyped tools should be migrated to a production environment.

4.4 Technique 3 - ANSI/IEEE Standard 830-1984

ANSI/IEEE Standard 830-1984 is an initial effort by an internationally recognized standards organization in attempting to develop a generalized

software specification standard. Initially the working group intended to develop a standard, but the result was a guide.

Technique Overview

ANSI/IEEE Standard 830-1984 is a guide to bring order on the use of natural language for software requirements specification. As a guide, this document does not constitute a mandated standard.

Method

ANSI/IEEE Standard 830-1984 prescribes a fixed outline approach to the development of software requirements specification. The outline consists of four parts: introduction, general description, specific requirements, support information. The introduction, general, and support sections can be considered as a generic part that applies to all software systems development. The specific requirements part provides four different outlines for the detailed development of a specification. These four outlines use the same components in different arrangements depending upon project organizational issues and problem application domain.

Supporting Tools

The supporting tools for ANSI/IEEE Standard 830-1984 include general purpose text editors and documentation formatters. There is limited opportunity for automated analysis of a specification as this approach is primarily natural language. This technique could be integrated within a more sophisticated natural language approach, such as ARTS [Dorfman84].

Language Features

ANSI/IEEE Standard 830-1984 uses natural language in a fixed format. Beyond the fixed outline there is a wide spectrum of natural and formal approaches that can be incorporated with the technique. This flexibility would defeat the standard goals, unless an organization mandates the approach to the non-fixed part.

Strengths and Weaknesses

This approach alone has limited applicability to prototyping software systems. However, the technique can bring control to an organization that is using ad hoc techniques. A new software requirements specification standard is expected to capitalize on the advances made in formal languages during the 1980s.

4.5 Technique 4 - ARTS

The Automated Requirements Traceability System (ARTS) was developed in the late 1970's by the Lockheed Missiles & Space Company, Inc. [Dorfman84].

The system is an early effort in automating software requirements bookkeeping. The system manages natural language software requirements for traceability.

Technique Overview

The technique uses a hierarchical decomposition of natural language software requirements in a database. Specified formats are used for stating requirements. Software functionality is described in natural language. More rigid formats are used for input, output, and data structures.

Method

A specification developer builds a database of hierarchically decomposed software requirements. The constructed requirements can be retrieved, modified, and formatted in report form. Traceability is obtained through the tree structured storage of the requirements. The requirements reports support technical program reviews and test plan development.

Supporting Tools

In the mid-1980s, ARTS was implemented on UNIVAC, DEC, and IBM machines. A powerful user interface supports requirements formulation and analysis. A relational database system, RIMS, provides the requirements storage and retrieval capability.

Language Features

Requirements formulation is performed according to specified formats. Allocation links ensure that higher level requirements flow down to lower level requirements for maintaining traceability. ARTS could be adapted to handle software requirements developed under ANSI/IEEE Standard 830-1984.

Strengths and Weaknesses

ARTS represents mature first generation technology to capture software requirements. There is limited prototyping support for other than input and output formats. A simulation facility could walk the user through the hierarchical structure as an enhancement to the support software. Development of explicit links to design, code, and test documents would provide traceability throughout the software life cycle.

4.6 Technique 5 - DARTS

The Design Approach for Real-Time Systems (DARTS) was developed by Gomaa at the General Electric Industrial Electronics Development Laboratory [Gomaa84]. Although described as a design method, DARTS bridges the software requirements specification and design phases. The DARTS development was influenced by techniques developed during the late 1970.

Technique Overview

DARTS is built upon concepts from functional, data flow, and object-oriented techniques. The real-time application domain required the development of language features to support task communication and synchronization. DARTS uses a graphical approach to represent a software system.

Method

DARTS uses data flow diagrams as the first level of decomposition of system requirements. This approach allows for interface definition of subsystems early in the software life cycle. The first level decomposition leads to a software system structuring that exploits tasks in parallel and sequential execution. The task identification then allows for a redefinition of the initial decomposition, followed by design and implementation.

Supporting Tools

DARTS was reported [Gomaa84] as a manual technique with no supporting tools. The advent of relatively low cost bit mapped displays provide a practical outlet for this technique. Experience gained since the mid-1980s on developing support tools for late flow methods would indicate that a DARTS implementation would be low risk.

Language Features

DARTS uses language features from structured analysis and structured design. DARTS employs information hiding principles through task interfaces. A task synchronization module and task communication module implement the information hiding.

Strengths and Weaknesses

DARTS is a process model approach that is real-time language independent. A major benefit of DARTS is decomposition and partitioning into concurrent tasks. The prototyping capabilities are limited to manual analysis. However, automated support would enhance DARTS based development and simulation of requirements.

4.7 Technique 6 - Descartes

Descartes is one of the earliest executable specification languages having been developed in 1977 as part of Ph.D. dissertation research [Urban77, Urban90]. The language has been limited to laboratory use as a research tool and in graduate software engineering courses.

Technique Overview

Descartes is based on data structuring methods proposed by Hoare. A tree structure notation is used to perform analysis and synthesis of data. Most of the research effort associated with Descartes has been on providing extensions to the original tool support. However, real-time extensions to the language definition have been recently under development, as well as computer-aided transformation support for mapping Descartes specifications into object oriented designs. Descartes specification technology has matured to the point that a commercially viable product could emerge in the mid 1990's.

Method

Descartes specifications are developed in a top-down modular fashion. Partial specifications are supported through partial refinement of the tree structure notation. The specifications are executed through interpretation. Once the specifications are completed, any design technique could be used as the next step in the software process.

Supporting Tools

An interpreter exists for executing Descartes specifications on abstract and concrete data. This abstract execution feature allows for dynamic analysis of partial specifications. Initially developed in PL/I on Multics, the interpreter currently is written in C on UNIX and executes on a Sun workstation. An automatic test data generation capability was developed to generate input data based on path analysis. Recent work on the support software has focused on extending the visual representation mechanisms for the specifications.

Language Features

The three data structuring methods of Hoare used in Descartes are direct product, discriminated union, and recursion, which are the same as those used in the Jackson Design Method and Jackson Structured Programming. However, in a Descartes specification the data structuring methods are used in tree structures to define the input and output data. where the analysis of the input data is such that output data can be synthesized as a function of the input data.

Strengths and Weaknesses

The training time for learning Descartes is little, based on over a decade of graduate course experience. The language is simple to use, but powerful for demonstrating software functionality early on in the software life cycle. One aspect of using Descartes is in run-time overhead of the language processor due to the matching process applied to the tree structures.

4.8 Technique 7 - ENVISAGER

The ENvironment for the VIsual Specification And Graphical Execution of Requirements (ENVISAGER) [Diaz-Gonzalez87a, Diaz-Gonzalez87b, DiazGonzalez91] is a visual requirements engineering environment for the specification of real-time systems. The development of the environment has been partially supported by Bell-Northern Research. The ENVISAGER environment is a prototype and has been limited to laboratory use as a research tool.

Technique Overview

ENVISAGER is based on a conceptual model using an object-oriented approach, with the interaction between objects being specified by Interval Temporal Logic formulas. The environment utilizes high-level graphical representations of the actual objects in the application being specified and animates the objects to represent the operations.

Method

The technique is supported by a high-level graphical interface that incorporates notions which are common in real-time systems, such as messages, processes, and timers. An extension to traditional first order logic that provides mechanisms for specifying time varying properties of systems was used as the underlying formalism of the technique.

Supporting Tools

By using the graphical interface, a software engineer is able to specify the dynamic behavior of a system. In addition, the graphical specifications are reused by a simulator providing animation facilities. By using the animation of specifications, the execution of concurrent and time-constrained activities can be examined in detail by the software engineer, and in this form determine the validity of the requirements early in the development stage. The animation facility currently executes a fixed set of test cases.

Language Features

The underlying representation mechanism within ENVISAGER is Interval Temporal Logic which provides a facility for the description of constraints on a system with respect to the execution state of abstract operations. Namely, the logic provides facilities for specifying that the system is at A (just before the beginning of execution of A), in A (within the execution of A), and after A (just after finishing the execution of A), where A is an abstract operation. The recent incorporation into the environment of an abstract data type definition facility improves the reusability of objects, and through the inheritance of object behavior, also facilitates the description of new objects. For example, a datavoice terminal could be defined through the inheritance of operations provided by data terminals and voice terminals, therefore only the constraints that control the

contention for shared resources in the object need to be developed in the specifications.

Strengths and Weaknesses

A primary strength of ENVISAGER is in the significantly different means for demonstrating functionality to the end user. Demonstrating functionality through the animation of the actual objects that are part of the end user's system provides a realistic representation mechanism. Software developers of the Interval Temporal Logic representation of the objects need a mathematical background through first order logic. However, the software engineers who are the users of the objects need to tailor the object definitions to meet the needs of the application. Additional tool support is needed to assist in the tailoring of objects.

4.9 Technique 8 - GSS

The development of the Graphical Specification System (GSS) is a joint software environment effort between Texas A&M University and Lockheed Software Technology Center [Harbert90]. GSS represents second generation specification technology. GSS is part of the Express [Topping87] project (not the proposed national standard, Express [van Delft89]), which is a knowledge based approach to prototyping embedded systems.

Technique Overview

The application domain for GSS is the user interface specification for realtime embedded systems. The technique uses object-oriented, data flow, and functional techniques to rapidly prototype a user interface. A GSS user interface specification can be linked to specifications of the remainder of an application to complete the prototyping process. This syngerism of user interface and application specifications enhances the significant benefits that can be obtained from having the user approve only the interface.

Method

GSS uses a library of graphical display units with default interactions as the basis for defining a user interface. The interactions of graphical display units can be modified to meet the application needs. A separate back-end specification technique defines the interface between the user interface and application software.

Supporting Tools

GSS includes supporting tools for icon generation/retrieval, display generation, and linkage to application software. The library is extensible for tailored/existing library units, as well as new application domain icon development.

Language Features

The graphical display units allow a software developer to visually specify the objects that an end user will need in a system. The underlying representation mechanism for end user interactions with objects is state transition diagrams. These state transition diagrams exist for library display units and may be modified to meet application needs. Data flow diagrams are used as the backend specification link with an application.
on.

Strengths and Weaknesses

This technique and tool support represent critical technology for currently meeting the needs of icon-based software development. An icon-based approach to prototyping user interface development should significantly reduce end user and developer misunderstandings. Extensive library development is needed to significantly improve productivity, which will come with further experience.

4.10 Technique 9 - Larch

The Larch specification languages were developed as part of a project at MIT's Laboratory for Computer Science and DEC's Systems Research Center [Guttag85]. The Larch project builds on the researchers' foundation in algebraic specification languages, since the mid-1970. This project has been supported by the Defense Advanced Research Projects Agency (DARPA), Digital Equipment Corporation. Xerox, and the National Science Foundation (NSF).

Technique Overview

A two-tiered approach to software specification is used with the Larch family of languages. The first language, Larch Shared Language, is an algebraic specification language that is used for all programming languages. The second language, Larch interface language, is one of many languages tailored to individual programming languages.

Method

Developing a Larch Shared Language specification involves constructing a theory for an abstract data type. The theories are referred to as traits, which are the Larch Shared Language units for software reuse. Larch/Pascal and Larch/CLU interface languages have been developed as second-tier languages. These Larch interface languages provide the means for specification of data and procedural abstraction.

Supporting Tools

A set of tools has been under development in the Larch project.

Specification construction is through syntax-directed editing to allow for incremental checking during development. A theorem prover supports semantic checking. Tools have also been proposed for specification management and browsing. Tool support for the AFFIRM abstract data type language [Musser80] has influenced the developers of the Larch tool support.

Language Features

A Larch Shared Language trait can consist of seven clauses: assumes, includes, imports, introduces, constrains, converts, and exempts. The constrains clause includes the axioms regarding the operations on an abstract data type. The Larch interface languages have three parts for specifying data abstraction: header, trait/type mapping, and operations interface. In addition, the Larch interface languages have three parts for specifying procedural abstraction: header, traits with theory of operators, and a body of routine effects.

Strengths and Weaknesses

The Larch family of languages is a powerful set of specification languages for software reuse of abstract data types. The syntactic and semantic checking capabilities will benefit software productivity and reliability. Automated transformation between the shared, interface, and programming languages would further benefit this approach.

4.11 Technique 10 - PAISLey

PAISLey was originally developed as part of Ph.D. dissertation research in the late 1970's [Zave91]. The early 1980's witnessed the application of the technology on a variety of applications. Software tools in a UNIX-based environment to support the analysis and execution of specifications began to emerge in the mid-1980s while the language developer was at AT&T Bell Laboratories.

Technique Overview

PAISLey is a Process-oriented, Applicative, and Interpretable Specification Language based on the process model and therefore well-suited for the real-time application domain. This model evolved as a result of merging asynchronous processes and functional programming with exchange functions.

Method

The PAISLey approach involves decomposing a system into a set of asynchronous processes that are analyzed and executed. Thus, the end user and software developer must address event and timing issues early on in the software life cycle. The notation is such that the software engineer develops the specification and the end user reviews the executable functional specification. The tool support aids both developer and user with identifying consistency and completeness errors.

Supporting Tools

The set of software tools supporting PAISLey include syntax and semantic checking through a cross referencer, static analysis through a consistency checker, and execution of specifications through an interactive interpreter. The interpreter provides dynamic analysis in process and time dimensions.

Language Features

A PAISLey specification for a software system consists of a set of asynchronous processes which are described in a graphical (boxology) notation based on finite state machines. Exchange functions provide support for stating synchronization and communication among the asynchronous processes. In addition, the language provides features for timing constraints at behavioral, architectural, and atomic levels.

Strengths and Weaknesses

A major strength of PAISLey is ease of use for both the language (provided there is not an explosion of processes) and support software. Early life cycle feedback to the end user is another benefit attained with PAISLey. However, performance testing is not currently supported in the toolset.

4.12 Technique 11- PLEASE

The PLEASE specification language was developed by the Software Automation, Generation, and Administration (SAGA) project as part of the ENCOMPASS software engineering environment [Terwilliger89]. PLEASE is an executable specification language developed as part of Ph.D. dissertation research. The research was supported under a NASA grant in the mid-1980s.

Technique Overview

PLEASE is an Ada-based specification language which uses annotations that are converted to Prolog for execution. The language design was influenced by Anna and VDM. The specification language and tools are used for prototyping through incremental development.

Method

A multi-level approach is used by a developer for software development with PLEASE, which is based on the assertional model. The first level is an abstract specification of end user needs. The abstract specification is then validated against the end user needs. The second level is a design transformation that is verified with a theorem prover. This refinement proceeds to the lowest level or decomposition.

Supporting Tools

Development or PLEASE specifications is supported through a language oriented editor that performs syntactic and semantic checking. A proof management system, TED, is linked to theorem provers for verification. For execution, there is a transformation tool that generates Prolog code and the necessary Ada code to invoke Prolog procedures.

Language Features

PLEASE has pre- and post-conditions for the specification of assertions that are used for validation and transformation. The assertions extend Ada in a manner similar to Anna [Luckham85, Luckham87]. PLEASE predicates are similar to Prolog predicates.

Strengths and Weaknesses

PLEASE is a unique integration of language designs, which includes Ada, Prolog, and VDM. The language is part of the ENCOMPASS environment, combining executable specifications with programming-in-the-large. There is a run-time overhead associated with the use of Prolog. This overhead will be reduced as logic programming technology advances. However, the overhead is minor compared to the benefits of verification during incremental development of software.

4.13 Technique 12 - PROSPER

The PROtotypes and SPEcifications with Relative types (PROSPER) specification language was developed by researchers at the Polish Academy of Sciences and Colorado State University [Leszczylowski89]. The PROSPER language definition was influenced by Vienna Development Method (VDM) [Jackson85], Z [Norris90], and the abstract data type model. Although PROSPER is based on non-executable specification languages, the result is a fully typed (not strongly typed) language suitable for the execution of prototypes.

Technique Overview

The PROSPER specification language is based on the process model. Abstract data types are developed as fully typed objects and associated functions. Polymorphism and dependent types are used to parameterize type expressions. The language is similar in nature to the Larch Shared Language [Guttag85] approach to building reusable software components.

Method

The PROSPER specification language is a kernel based on primitive language constructs. Higher level abstract specifications are built in PROSPER on the primitive constructs. This approach is closer to detailed specification than requirements engineering. However, the introduction of VDM and Z language

concepts is innovative.

Supporting Tools

A prototype PROSPER interpreter was reported as under development [Leszczylowski89]. The language structure is such that a syntax-directed editor would benefit construction productivity and consistency checking.

Language Features

PROSPER differentiates between a basic world for the object level logic and a super world for the meta level logic. The two worlds are part of an infinite hierarchy of universes. The language features include: parameterized type expressions, value/module declarations, and special type operators.

Strengths and Weaknesses

PROSPER forms the basis for building libraries of reusable components. Linking PROSPER with a higher level specification language supporting synchronization and communication constructs would facilitate real-time system development. Executable prototypes, although at the abstract data type level, will benefit reuse component retrieval.

4.14 Technique 13 - PROTOB

PROTOB was developed at the Politecnico di Torino under partial funding through the Italian C.N.R. Project, Progretto Calcolo Parallelo obiettivo SPECTER [Baldassari91]. PROTOB represents a second generation executable specification language for prototyping real-time systems. The technique combines the graphical and textual approach.

Technique Overview

PROTOB uses an object-oriented method that is based on PROT nets, which are high-level Petri nets [Bruno86]. This technique is targeted for event-driven systems. The functionality of a system is described in terms of states and state transitions. An extensive environment was developed to support the technique.

Method

There are three major phases in the development of PROTOB specifications, which are modelling, emulation and application generation. A hierarchical architecture of graphical objects is developed in PROT nets. The objects are represented in a boxology which may be animated graphically in a discrete-event simulation. The emulation phase is a refinement of a PROT net model to handle implementation details. Finally, the application generation phase establishes links to the hardware and software environment.

Supporting Tools

The PROTOB environment includes the following kernel tools: editor/animator, translator, simulator/emulator, report generator, and script generator. The environment executes in a distributed architecture of VAX/VMS and UNIX machines. The toolset supports consistency and completeness checking of specifications.

Language Features

The use of high-level Petri nets does not result in a large number of objects as in other process models. Open and closed objects differentiate externally communicating and internal objects, respectively. External objects are referred to as software chips.

Strengths and Weaknesses

PROTOB builds on advantageous features of the object-oriented, data flow, and Petri net approaches to specification development. The PROTOB environment is an integrated set of tools for comprehensive requirements engineering support. An automated test data generation capability would enhance this prototyping technology.

4.15 Technique 14 - PSDL/CAPS

The Prototype System Description Language (PSDL) and Computer-Aided Prototyping System (CAPS) [Luqi88a, Luqi88c, Luqi89b, Luqi90, Luqi92] is a software specification language and set of support software for rapidly prototyping large-scale, real-time, embedded software systems. The research effort for the development of PSDL/CAPS was conducted primarily at the Naval Postgraduate School with partial support through the National Science Foundation.

Technique Overview

This approach is an operator, data, and control abstraction technique that supports retrieval of software components through a software base. PSDL supports the development of prototypes that demonstrate functionality through execution. There is a well-defined hand-off to the Ada programming language at the detailed design level.

Method

The method supports iterative development of software requirements based on end user needs. The executable prototyping capability provides the end user with rapid feedback, especially during feasibility studies. The method maps to the Ada programming language as the basis for implementation through reusable components.

Supporting Tools

The tool support for PSDL includes a static scheduler, translator, dynamic scheduler, software base management system, syntax directed editor, and paraphraser. The execution aspects of the language appear to be best suited for translation into Ada.

Language Features

The computational model uses data streams to communicate with operators. Control constraints allow for the specification of real-time operators, triggers, timers, and conditionals. Timing and hierarchical constraints are also supported in the language definition.

Strengths and Weaknesses

PSDL has a set of abstractions suitable for developing large-scale, complex, real-time software systems [Luqi92]. The language and system serve as an executable prototyping approach for specification or design. However, further work on PSDL has been proposed for handling tight real-time constraints in a distributed computing environment. The software base could be the driving factor on achieving significant productivity gains.

4.16 Technique 15 - SBRE

The Scenario Based Requirements Elicitation (SBRE) technique was developed by Captain Hilliard Holbrook III at the US Air Force Academy [Holbrook90]. This technique addresses a formalization for improving initial end user and developer communications. This research effort is continuing at the University of Florida.

Technique Overview

SBRE is intended to serve as a bridge between the concept in the end user's mind and the developer's elaboration of the concept. There is a capability to concurrently develop the requirements with the high level design. Software system scenarios are the primary means for communicating end user requirements. This approach is similar to other scenario-based techniques [Hsia86, Hallman88].

Method

A conceptual architecture is developed to describe the user's world and the designer's world. The conceptual architecture consists of four sets of information: goal, scenario, issue, and design. The scenario and issue sets bridge the two worlds, while the goal and design sets are part of the user and designer worlds, respectively. The requirements elicitation process has three phases of goal elaboration, scenario elaboration, and scenario evaluation.

Supporting Tools

Hypertext tool support is envisioned for software requirements/design construction and management. Holbrook described a proposed Apple Hypercard implementation. An SBRE architecture would be maintained on five stacks for - the four information sets, as well as constraint information.

Language Features

A user's goal set can include requirements, constraints, standards, and available resources. The design set of the designer's world is a base of high-level designs and design decisions. An SBRE scenario set is a behavioral specification of the designer's interpretation of the goal set. The issue set can include assumptions, responses, and implications.

Strengths and Weaknesses

This proposed technique addresses a means for providing structure on end user and developer interactions. Further work is needed and planned for implementing this approach to reducing misunderstandings early in the software life cycle. Linking the high-level designs to a software reuse library would provide support for executable prototypes.

4.17 Technique 16 - SLAN-4

Software Language-4 (SLAN-4) is a specification and design language that was developed at IBM Laboratories, Boeblingen, West Germany, beginning back in 1978 [Beichter84]. The language focus is on specifying and designing abstract data types and modules through four approaches.

Technique Overview

The technique is used after software requirements formulation and analysis as a means to capture functional specification through detailed design, as the early real-time technique, EPOS [Biewald79]. The functional model is the basis from which SLAN-4 was developed with data and functional abstraction. The explicit linkage between specification and design provides a means for minimizing phase to phase transition problems.

Method

The four approaches to SLAN-4 specifications are: abstract data types, algebraic specifications, axiomatic specifications, and pseudocode design. The first approach uses a class construct as a means to achieve abstract data type definitions. The second approach uses algebraic specifications to define the relations between abstract data types and modules. The third approach uses pre- and post-conditions to axiomatically define a specification module. The fourth approach allows for the detailed design of the remainder of the system.

Supporting Tools

There was no tool support for SLAN-4 in 1984 [Beichter84]. However, the developers suggested the following tool support: syntax-driven editor, syntactical/semantical checking, and a database. However, the developers also noted that SLAN-4 specifications "cannot necessarily be translated into machine-code by a compiler."

Language Features

The class construct in SLAN-4 allows for the top-down development of data abstractions. A class is composed of an interface declaration, class specification, class and module declarations, and instantiation statements. A module is composed of an interface declaration, result type, module specification, class and module declarations, and statements. A module specification allows for the axiomatic definition of module pre-expression and post-expression, optionally intermediate expressions, and expressions for exceptions. The statements part of the specification is stated in pseudocode.

Strengths and Weaknesses

SLAN-4 was reported as being applied to an industrial software product that resulted in about 18,000 lines of high-level language source code. The experience base available on algebraic and axiomatic specifications is such that software engineers with sufficient formal notation background should have minimal difficulty with the syntax and semantics of SLAN-4. However. difficulties with developing the axioms for new applications are a drawback of using this technique. In addition, the lack of tool support will impact productivity and reliability.

4.18 Technique 17 - SPADES

The SPecification And DEsign System (SPADES) was developed at Brown Boveri Research Center in Switzerland [Ludewig85]. The system is a direct evolution of the developer's earlier work on PCSL, which was followed by ESPRESO [Ludewig83]. ESPRESO was developed for real-time system specification at the Nuclear Research Centerin Karlsruhe, Federal Republic of Germany. The researchers were influenced by PSL/PSA [Teichroew77], SREM [Alford85], and the Entity-Relationship model [ChenP90].

Technique Overview

SPADES consists of a specification method (SPADES-M), language (SPADES-L), and set of tools (SPADES-T). The specification language is a textual approach (with some boxology capability) that has comprehension comparable to programming languages. The initial toolset, although small, has tools for specification construction, storage, retrieval, and report generation.

Method

Specifications developed in SPADES-L are analyzed and stored in a database by the SPADES-T set of tools. Access control mechanisms in SPADES-T ensure the integrity of a specification under development. The entity-relationship concepts in SPADES guide a specifier in identifying objects and links.

Supporting Tools

The SPADES-T toolset is implemented in Modula-2 in VAX/VMS. A SPADES-L analysis tool provides consistency and completeness checking capabilities. A conversion and deconversion tool translates specifications for storage and retrieval in the database. The report generation tool can provide six reports: content, hierarchy of modules and informal objects, call structure, data flow, range check. and completeness.

Language Features

SPADES-L can describe eight kinds of objects: modules, actors, parameters, media, types, intervals, constants, and informal objects. In addition. SPADES-L can describe six kinds of relations which are used to link objects. The SPADES-L relations include: hierarchies (and other structures), communication coordination, execution schedules, restrictions, and general references. The language structure has been referred to as Pascal-like.

Strengths and Weaknesses

SPADES was developed on solid software requirements specification technology that emerged in the late-1970. The user base of real-time system developers influenced enhancements to SPADES. A syntax-directed editor would aid the development of SPADES-L specifications. A code transformation tool is needed to provide an executable prototype capability.

4.19 Technique 18 - SREM

The Software Requirements Engineering Methodology (SREM) was initially developed by TRW in Huntsville, Alabama [Alford77, Alford85] under the Software Development System project of the US Army Ballistic Missile Defense Advanced Technology Center (BMDATC) [DavisC77]. SREM later evolved to the Distributed Computing Design System through additional BMDATC support and through support from the C3I program at the US Air Force Rome Air Development Center (now Rome Laboratory). More recently, SREM and DCDS further evolved into a commercial product.

Technique Overview

SREM is based on the functional model where software requirements are developed in the Requirements Specification Language (RSL) in either textual or

graphical (boxology) form. The Requirements Engineering Validation System (REVS) supports simulation of RSL specifications that are maintained in a database.

Method

Software requirements are formulated in RSL, stored in the REVS database, analyzed for consistency and completeness errors, and simulated to demonstrate software functionality. Performance constraints, such as events and timing, are identified during requirements formulation and analysis for validation against the end user needs. In addition, automatic documentation generation is provided as a benefit of using SREM.

Supporting Tools

The REVS support software includes tools for both textual and graphical development, extraction, modification, and documentation of RSL specifications through the editor and database tools. Static and data flow analysis is provided through the RADX tool. Dynamic analysis is provided through simulation tools that support both functional and performance aspects of the specifications.

Language Features

The language, RSL, has language primitives for element, attribute, and relationship primitives which are closely analogous to the English language equivalent of noun, adjective, and verb, respectively [Bell77]. The language definition contains twenty-one elements to define features, twenty-one attributes to describe elements, and twenty-three relationships to describe element connections. Structures, the fourth primitive mechanism in RSL, are used to describe processing sequences. The language includes a facility for user extension of the primitives.

Strengths and Weaknesses

SREM is proven technology for software requirements engineering based on longevity and success in a wide variety of real-time applications. One of the most extensive applications of SREM technology [Scheffer85] was for a highly complex C3I application. One SREM drawback is in the size of both the RSL specifications and the REVS support software, which impacts the learning curve and run-time efficiency aspects of using the technology.

4.20 Technique 19 - STATEMATE

The STATEMATE system was developed in the mid-1980s by i-Logix Inc., Burlington, Massachusetts and Ad Cad, Rehovot, Israel [Harel90, i-Logix, Inc.90]. The system is commercially available through i-Logix, Inc. in the United States. The STATEMATE tools are used for specification, analysis, design, and documentation.

Technique Overview

STATEMATE supports the specification and design of a software system through multiple viewpoints of objects: structural, functional, and behavioral. Three types of charts are used to obtain the multiple viewpoints. These charts are used to represent modules, activities, and states in a graphical representation form.

Method

In STATEMATE, the structure of a software system is developed through physical decomposition and information flow analysis. The functionality of a software system is obtained by functional decomposition and information flow analysis. The behavior of a software system is derived from control mechanisms and temporal relations. The order of developing the viewpoints is as described in this paragraph. However, there appears to be no restrictions on the ordering on the decomposition and control, which can be viewed as providing flexibility.

Supporting Tools

The STATEMATE system is implemented on Apollo, Digital Equipment Corporation, and Sun Microsystems workstations. The system executes in the UNIX and VMS operating systems with several windowing systems. The toolset is extensive with graphics and forms editors, analysis support, simulation/code generation capabilities, report generators, management functions' document generation, and a database.

Language Features

A module-chart describes the structural view of a system with external and internal modules. An activity-chart combines data and control flow to form the structural viewpoint. Statecharts are an extension to finite state machines and state transition diagrams for developing the control in the behavioral viewpoint.

Strengths and Weaknesses

The STATEMATE system has been successfully applied to the development of real-time systems. Multiple viewpoints are viewed as an advantage, but that has not been reported as having been measured empirically. The simulation facilities provide for a rapid prototyping capability. Code generation is supported for the Ada and C programming languages.

4.21 Technique 20 - SUSL

SUSL is an abstract data type specification language that was developed as part of Ph.D. dissertation research [Belkhouche86]. The SUSL language design was influenced by the algebraic specification approach and OBJ [Goguen79], as well as later influence from OBJ2 [Futatsugi85]. SUSL was developed as a formal specification language to support transformation to high-level language

constructs.

Technique Overview

SUSL is based on the process model for the definition of abstract data types and their associated operations. SUSL has the wide range of applicability that is associated with algebraic specification techniques. Assertions are used to describe the semantics of the operations on the abstract data types. The code transformation capability allows for rapid prototyping of abstract data types.

Method

A specification developer uses the object-oriented approach to identify datatypes and operations. Input and output assertions on the operations describe the semantics. An input assertion is a relation or invariant of components that must be true upon entry to an operation. An output assertion is similar to an input assertion, where the condition applies upon exit.

Supporting Tools

The SUSL support software was developed in PL/I on the Multics operating system. An analysis capability provides, syntactic, consistency, and completeness checking. A synthesis capability provides for the automatic generation of PL/I or Pascal code from error-free analyzed specifications.

Language Features

Language defined abstract data structures include: sets, sequences, cartesian products, and discriminated unions. The syntax of a SUSL specification is centered around the abstract data type. Each abstract data type is described with the following sections: header, interface, abstract representation, initialization, operations, restrictions, and tail. The language supports generic abstract data types through parameterization in the header. The operation section is used for describing operation behavior through the input and output assertions.

Strengths and Weaknesses

SUSL provides a powerful specification technique in that reliability can be improved through the requirement for input and output assertions on operations. The effort to develop the assertions is comparable to other algebraic and axiomatic approaches. Productivity gains can be achieved through the transformational approach to code generation. The tool support would benefit from the development of a syntax-directed editing capability. Porting the SUSL tool support to an Ada-based environment and providing an Ada code generation capability would complement SUSL and Ada.

4.22 Further Technique and Tool Analyses

This section includes the basis for further detailed analysis of the software

requirements and specification techniques. The detailed analysis described in this section was based on criteria that have appeared as important to requirements and specification technique developers, software engineers who use the techniques, and software project managers.

4.22.1 Software Requirements and Specification Repository

This section has provided summary descriptions and analyses for a wide cross-section of software requirements and specification techniques. A comprehensive repository for software requirements and specification technology would benefit the field of software engineering. Software application developers, project managers, language designers, and tool builders would have access to the repository. The repository could be an integrated version of the CASE tools activities performed at the US Air Force Software Technology Support Center (Ogden ALC/TISAC, Hill AFB, Utah) and the process assessment activities at the Software Engineering Institute (Carnegie Mellon University, Pittsburgh, Pennsylvania).

The repository activities should include developing and maintaining information on non-proprietary software requirements and specification technology. This information should consist of summary and detailed analyses of the techniques and tools. The summary analyses could be at the level of detail and similar format that was used for this report. The summary analyses would be useful for initial study by browsing as with a bibliographic database of publication abstracts.

4.22.2 Detailed Analyses

The detailed analyses should be developed consistently across all the techniques. The detailed analyses would be used as the basis for technique selection by a software development group. An evaluation method similar to the Edmonds and Urban approach [Edmonds84], described in Section 3, should guide the development of the detailed analyses. The selection of techniques by software development groups, under an approach like the Edmonds and Urban method, would involve establishing the weighting factors for each group. The detailed analyses should address at least the following sixteen categories with regards to software development: conceptual development support, processing, formal foundation, analysis method, basic structures, graphical support, interfaces, abstractions, proofs of correctness, control flow, comments, software life cycle model, training, applicability, management support, and availability.

In terms of conceptual development support the techniques should be assessed with regards to the level of abstraction, software reusability, user defined types, and reverse engineering. The executable processing of requirements and specifications would be described for code generation. executable specification. pre-defined routines, and knowledge base capabilities.

The formal foundations of the techniques should address in the analyses

whether finite state machine, data flow, stimulus-response path, communicating processes, function composition, and/or data oriented foundations were the basis of the technique design. The analysis method used by the techniques can be decomposed into the following models: process, data, control, object, natural language, knowledge base, and assertional. The syntactic and semantic basic structures can be categorized according to which of the following forms are used: high level language constructs, set expressions, tree structures, objects, and textual form. When graphical support is provided with a technique there is typically data-flow diagrams, control-flow diagrams, tabular forms, object-oriented diagrams, and actual objects.

The language features part of the analyses should include interfaces. These interfaces include internal language interfaces for procedures, data, and tasks. A broader set of external interfaces include phase-to-phase transition across the software life cycle, tool integration, and platforms. More specifically in terms of language features, there is a need to determine the support for abstractions in terms of data, procedural, classification hierarchy, and inheritance mechanisms. Techniques that support proofs of correctness concepts will include features for pre- and post-conditions, rewrite rules, and/or Boolean values. Control flow would be assessed in terms of features for specifying sequential processing, parallel processing, and timing constraints. Lastly in language features, comments may not be part of a technique, but when these are included there can be formal and informal approaches used.

Software project management issues should be addressed in the detailed analyses. The different software life cycle models in which a technique can be used should be one or more possibilities of waterfall, prototype, spiral, and transformational models. The training support can take the forms of manuals, formal training/education programs, and on-line support through help files, as well as computer-based instruction. The applicability of a technique should be described in terms of the categorization used by Boehrn, which is embedded, organic, and semi-detached types of applications. The management support available should measure tool support for configuration management, static analysis, and dynamic analysis. Finally, the availability of technique and tool support should describe how the technology can be acquired and the availability of user groups.

4.22.3 Applications Development

This repository will need to develop a variety of applications in order to support the detailed analyses. A set of candidate problems is needed to accommodate the dynamic analysis part of an evaluation technique like the Edmonds and Urban method. Managing the development of the applications would be necessary to ensure consistency. This management is also needed to obtain the measures that demonstrate success or failure with a technique.

This repository will also benefit the developers of software engineering standards. The field has matured to the point that the International Standards

Organization standard specification languages, SDL, Estelle [Budkowski87], and LOTOS [Biemans86], exist for protocols. The extensions to ANSI/IEEE Standard 830-1984 and other wide spectrum specification language standards would benefit from access to information in the repository. In addition, the level of field maturity suggests that a theory of specification languages would abstract the research efforts over the past twenty years. The repository would serve as basis for applying a specification language theory. Finally, the repository would benefit from the lessons learned in establishing the Requirements Engineering Testbed at Rome Laboratory.

This section has provided a survey of software requirements and specification technique with a view towards prototyping. A template was used for developing each summary description in order to have as uniform descriptions as possible. The section concluded with a description of detailed analyses that should be developed to assist the software engineering community. The next section addresses software technology transfer and more specifically the problems and a potential solution for software requirements and specification technology.

5. SOFTWARE ENGINEERING TECHNOLOGY TRANSFER

One mayor problem in software engineering that has had an impact on this new technology is the slow rate of technology transfer. This section addresses problems of technology transfer in software engineering. A potential solution involves building a base of components through the coupling of forward and reverse engineering.

5.1 Overview of the Problem

Software technology transfer is typically recognized as an average of seven years. Note that the lifespan of the MCC Software Technology Program [Belady92] was seven years. This average technology transfer time has been longer for requirements and specification technology. This problem is addressed below in terms of time, behavior, management, and quick-fix solutions.

5.1.1 Technology Transfer Time

The technology transfer time for requirements engineering and prototyping technology has been longer than other software engineering techniques and tools. This time length has been primarily due to the significant investment in large-scale software systems. This investment means that organizations have been reluctant to integrate this technology unless the technology is to be used for new systems development. Organizations have also been reluctant with regards to reengineering of existing systems. This reluctance is waning with the emergence of reverse engineering techniques and tools.

5.1.2 Behavioral Process

Another technology transfer problem relates to an inability to understand the behavioral processes involved in integrating this new technology. There are multiple aspects of behavior that impact technology transfer. One aspect is the ability of software developers to accept change in the methods that have worked successfully in the past. In many cases, if a formal notation is part of a new technique, there is a need for mathematical skills to be revisited or attempted for the first time. Learning these new techniques involves work, but there should be support for overcoming the learning curve through formal training. In addition, there should be a period of time for productivity improvement to be demonstrated in a non-production environment.

More importantly, end user and developer interactions have not been formalized much beyond the traditional waterfall life cycle model. Stephens and Bates [Stephens90] described the end user and developer interactions involving the use of interface and functional prototypes. These types of investigations need to be integrated within process models in order tO provide an explicit statement of expectations.

5.1.3. Middle Level Management

Middle level management has sometimes had this new technology thrust onto a project. Unfortunately, when faced with schedule slippage, the project manager is more likely to scrap the advanced technology, in favor of what has worked well in the past. Mayhew and Dearnley [Mayhew90] discuss the need for educating all participants and controlling the process.

5.1.4 Quick-Fix Approach

Finally, the technology transfer problem is compounded by a lack of resources and total commitment by upper level management to invest in the technology transfer process at a sufficient level to ensure success. Assuming a quick fix solution to the software problem has not been the answer with this technology.

5.2 Approaches to Technology Transfer

There are several approaches to technology transfer, five of which are addressed below before providing a proposed approach. These approaches include the market, standards, edict, guerrillas, and education. Each of these approaches works with varying degrees of success at technology transfer.

5.2.1 Market Driven

The market driven approach is one of the most visible and proven approaches to technology transfer. The competitive nature of software development implies that word spreads in the human networking of software developers. This approach is typically rapid for technology transfer. However, the approach can be hampered if the learning curve is not addressed adequately

by technique developers and those acquiring the techniques.

5.2.2 Government/Corporate Standards

Government and corporate standards are also one of the most visible and proven approaches to technology transfer. However, the development of standards are lengthy and lock onto a technology after maturity. Perhaps the most significant gain in requirements and specification technology transfer can occur in much the same manner that the Ada programming and environments was introduced to the software engineering community. A good example in place now is the requirements management aspect of the SEI process assessment for level 2. Some problems have been identified with requirements specification under DOD-STD-2167A, which should be explored further [Walter91a].

5.2.3 Edict/Fiat Directed

In lieu of more formal government and corporate standards, the edict or fiat driven approach is on a smaller scale, but of course could also be driven by the market or standards. The edict approach requires much less time than the market or standard driven approaches. However, the problems associated with the market driven approach can also occur in the edict directed approach.

5.2.4 Guerrilla Warfare

One approach to technology transfer that is more sporadic than the previous approaches is when new technology is learned by an individual (or individuals on a project) and then applied as a means to improve productivity and reliability. This approach is not desirable due to the potential for a lack of a management perspective on the introduction of the technology. However, when there is encouragement and a cooperative approach to introducing new technology, this approach can be quite effective.

5.2.5 Software Engineering Education

Formalizing the previous approach will occur through incorporation of software engineering education model curricula in graduate software engineering programs. Recently, DARPA announced plans to support the development of pilot undergraduate software engineering programs. These programs will create an opportunity to migrate some of the current graduate level software engineering concepts into the undergraduate curriculum. An opportunity would also be created for advancement of a theoretical foundation for the graduate level courses. Software requirements and specification concepts incorporated within undergraduate software engineering degree programs would soon be reflected in the industry. This approach is in much the same manner that UNIX was introduced within undergraduate computer science degree programs and later within industry.

5.3 Reverse Engineering

As mentioned earlier, the significant investment in large-scale software systems has been an impediment to the introduction of requirement engineering techniques. Reverse engineering is one area in which cooperation can be achieved to introduce requirements engineering. The remainder of this section addresses the problem and provides a solution.

5.3.1 Maintainers Hampering Developers and Vice Versa

In many cases over the system lifetime, there is the myth of maintainers using the developer techniques and tools. This situation is further complicated where there is a separation of maintenance and development from the standpoint of research. Lack of an apprenticeship approach with senior software engineers for Junior software engineers results in maintenance in isolation as the case for many new personnel. Finally, a lack of tool integration further complicates the situation.

5.3.2 Feedback and Feedforward

Early software reuse [Karakostas89] is viewed as one of the most important areas for increasing productivity. A comprehensive approach is needed where maintainers and developers work together. This approach needs forward engineering and reverse engineering to cooperate.

Forward engineering is the traditional process of moving from high level abstractions to physical implementation. Reverse engineering is a process of analyzing a subject system to identify and create. The identification part of reverse engineering involves identifying system components and their relationships. Based on the identification work, representations of the system are created in another form or at a higher level of abstraction.

5.3.3 Advanced Library Systems

Advanced library systems of reusable components and the guidance on how to apply reusability can facilitate software development and maintenance. In many cases, the lack of these library systems has been the source of developer frustration when advertised productivity gains are not achieved relatively soon. These concepts can be applied now with existing technology. An area of future research described later expands this theme across application domains.

5.4 Case Studies

A major part of the detailed analyses in the repository described in Section 4 should include the application of the techniques on a wide variety of problems that have appeared in the open literature. Case studies are used extensively in other engineering, scientific, and business settings. The field of software engineering does not have a sufficient base of case studies that includes both successes and failures. The software engineering field does not have a standard

for reporting case studies. One problem for the software engineering field is that in many case studies, the technology developers report only on successful applications of the technology.

Four problem statements were a part of the Call for Papers for the Fourth International Workshop on Software Specification and Design (IWSSD) held in l987. These problems were from different application domains. Authors submitting papers for this workshop were required to provide a solution to one of the four problem statements using the technique described in the manuscript. The set of four problems included a home heating system, library database, text reformatter, and a lift (elevator) controller.

Jeannette Wing noted that twelve papers in the Fourth IWSSD proceedings had developed solutions to the library problem [Wing99]. She pursued detailed analyses of the twelve solutions under support from IBM, DARPA, and NSF. Another information system problem [Urban85] was derived from an existing system to analyze the Descartes specification language. For another existing system, a rapid prototyping technique was used to develop the undo support in GNU Emacs [Yang90]. On a smaller scale, the telegram problem has been used extensively for specification, design, and programming techniques [Urban90]. A payroll problem was described [Fraser91] to demonstrate the use of structured analysis and VDM. An estimating system for the metal-finishing industry was used for demonstrating the advantages of prototyping and identification of additional research questions [Stephens90]. The Sorcerer's Apprentice Problem, which has impossible semantic rules, was used to promote the techniques that use state machines as the underlying model [Jorgensen86].

There are several non-trivial real-time applications that have appeared in the open literature. The cruise control problem has become a commonly used technique for the demonstration of real-time features in requirements and specification techniques [Smith88]. A workcell architecture for computer integrated manufacturing [Biemans86] was formulated to demonstrate the applicability of LOTOS. AFFIRM was applied to an alternating bit protocol problem [Sunshine82]. The Gist operational specification was applied in the development of a package router for a network [Balzer83]. A non-trivial Navy communications system, SCP, was used for an abstract requirements specification technique [Heitmeyer83].

The command, control, communications, and intelligence (C3I) application domain was addressed in two case studies for demonstrating two different techniques. One of the C3I case studies presented was an example of technique developer reported success. The recent case study of a C3I prototype was reported [Luqi92] by the developer. The C3I prototype was developed with the use of the Prototype System Description Language (PSDL) [Luqi88a] and Computer-Aided Prototyping System (CAPS) [Luqi88b]. The second case study represents a report of efforts by a group other than the original developers. The second case study [Scheffer85] is also a C3I system and uses some mature technology, SREM. The repository should maintain solutions produced by both

technique developers and others.

These problems should form the basis for developing a matrix of requirements and specification solutions in all the techniques, wherever possible. The application development should be coupled with the detailed analyses such that domain issues are taken into consideration during technique selection. The applications developed to fill the matrix of solutions should follow the same case study format. This application development with a fixed reporting format will provide a significant base of case studies.

This section addressed software engineering technology transfer problems and an approach to speeding up the process for requirements and specification technology. The next section includes proposed future research and a report summary.

6. RECOMMENDATIONS AND SUMMARY

Four areas for future research are outlined to extend the prototyping technology. These research areas were identified as a result of technique analysis and other emerging technologies. The technique analysis identified the incorporation of multiple models within recently developed techniques. Two emerging technologies from fields other than software engineering were identified as appearing to have potential for incorporation within prototyping technology. The report concludes with a summary section of the research findings.

6.1 Future Research

As mentioned earlier, software prototyping and requirements engineering are entering a new generation of techniques and tools. The remainder of this section identifies four primary areas for future research in support of software prototyping and requirements engineering. Two of the research areas are directly related to technique and tool advancement. Two of the research areas depend on advances in other areas.

The first area of future research involves a recognition of the maturity of existing techniques to warrant technique unification. The second area is based again on the maturity of existing techniques to incorporate the advances in computer-supported cooperative work. The third area addresses adding further dimension to the understanding of end user needs through multimedia and scientific visualization. The last area draws on integrating existing work to perform domain analysis

6.1.1 Technique Unification

There are two directions with regards to the next generation of software requirements and specification techniques. One direction is towards the unification of techniques into a single technique that will essentially be a "do all"

for everybody. The other direction of research is towards continued development of new techniques, but the introduction of a backplane or spine through which a variety of techniques could be used to develop a software system. One common theme of both research directions is in efficiency. This efficiency theme cuts across optimizing development effort, ease of user understanding, and effective support software. Either approach can lead toward the development of a theory of software requirements and specifications. This theory of requirements and specification will be developed as was the case for theories within data engineering, operating systems, and program testing.

6.1.2 Computer-Supported Cooperative Work

The interactions between end user, development team. and management could be improved by capitalizing on advances that have been made in the area of computer-supported cooperative work. This area has two directions that need to be addressed with regards to software engineering. The first direction is the behavioral aspects of human interactions during the software life cycle. A user interface to foster cooperation among all participants in software development and maintenance is needed. The second direction must address the underlying support system issues for the cooperative interface.

6.1.3 Multimedia and Scientific Visualization

Another area of research involves tapping the potential for emerging technologies that are being used in improving understanding of other areas of complexity. The introduction of audio and video technologies will improve the interaction between end users and developers. One aspect of the interaction involves providing the end users with a more realistic representation of the objects that are part of the software systems. Another aspect of the interaction involves exploiting the technologies in ways to assist the software developer and manager in managing the complexity of software systems.

6.1.4 Domain Analysis

A final area of future research that has significant cost and reliability implications is domain analysis. Other engineering disciplines work with standard components that can be pulled off the shelf. Software engineering is still in an infancy period with regard to standard components, other than at a gross system level.

Research in domain analysis, such as the Rome Laboratory-supported Knowledge-Based Requirements Assistant, should focus the capability to allow for identification and development of standard components within application areas. In addition to component identification/development, the means for component selection and integration would need to be developed. These component aspects will have significant impacts on software productivity. However, more importantly the software components would be validated before becoming available for general use and reliability thresholds used as a means for

determining availability. Applying the integration techniques then becomes the critical issue with regards to reliability. As already demonstrated, software prototyping and requirements engineering are crucial technologies for success in this area.

6.2 Report Summary

Software prototyping and requirements engineering were addressed in this report as a means for decreasing software cost and improving software quality. The report included a discussion on several approaches **to** prototyping as a means for improving software development. Software requirements engineering was addressed as an approach to prototyping that can provide the most significant improvement in software development. Summary analysis of twenty requirements engineering techniques and tools were included to provide a broad perspective on the field.

Unfortunately, this technology has not had a significant impact on industry due to the infancy of the area as a research topic. However, the technology has matured to the point that the first generation technology has made a foothold in the state-of-the-practice. Further, the next generation of research efforts in requirements engineering show promise for significant impact on the software engineering discipline.

The goal of eliminating the design and coding phases of the software life cycle is feasible now, but at the price of execution efficiency. This elimination goal will become closer to being achieved through the remainder of this century.
.

7. REFERENCES

[Alavi91] M. Alavi and J C. Wetherbe, "Mixing Prototyping and Data Modeling for Information- System Design," *IEEE Software*, Vol. 8. No. 4, May 1991, pp 86-92.

[Alexander89] H. Alexander and B. Potter, "Case Study: The Use of Formal Specification and Rapid Prototyping To Establish Product Feasibility," *Information and Software Technology*, Vol. 29, No. 7, September 1989, pp. 388-394.

[Alford77] M Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January 1977, pp. 60-69.

[Alford85] M. Alford, "SREM at the Age of Eight; The Distributed Computing Design System," *Computer*, Vol. 18, No. 4, April 1985, pp. 36-46.

[Ambler77] A. Ambler, D. I. Good. J. C. Browne, W. F. Burger, R. M. Cohen,

C. G. Hoch, and R. E. Wells, "GYPSY: A Language for Specification and Implementation of Verifiable Programs." *ACM SIGPLAN Notices*, Vol. 12, No. 3, 1977, pp. 1-10.

[ANSI84] ANSI/IEEE Std. 830-1984: IEEE Guide to Software Requirements Specifications, in *System and Software Requirements Engineering*, Edited by M. Dorfman and R. Thayer, IEEE Computer Society Press Tutorial, Order No 1921, 1990, pp. 170-192,

[Archer90] M. Archer, D. Fincke, and K. Levitt, "A Template for Rapid Prototyping of Operating Systems," *Proceedings of the First lnternational Workshop on Rapid System Prototyping*, Research Triangle Park, North Carolina, June 4-7, 1990, pp. 119-127.

[Archibald83] J. L. Archibald, B. M. Leavenworth, and L. R. Power, "Abstract Design and Program Translator: New Tools for Software Design," *IBM Systems Journal*, Vol. 22, No. 3, 1983, pp. 170-187.

[Auernheimer86] B. Auernheimer and R. A. Kemmerer, "RT-ASLAN: A Specification Language for Real-Time Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 9, September 1986, pp. 879-889.

[Babcock89] J. Babcock, S. Gerhart, K. Greene, and T. Ralston, SpecTra: *A Formal Methods Environment*, MCC Technical Report Number ACI-ILO-STP-324-89, August 1989, 15 pp.

[Bailin89] S. C. Bailin, "An Object-Oriented Requirements Specification Method," *Communications of the ACM*, Vol. 32, No. 5, May 1989, pp. 608-623.

[Baldassari91] M. Baldassari, G. Bruno, and A. Castella "PROTOB: An Object-Or,iented CASE Tool for Modelling and Prototyping Distributed Systems," *Software Practice and Experience*, Vol. 21, No. 8, August 1991, pp. 823-844

[Balzer83] R. M. Balzer, D. Cohen, M. S. Feather, N. M. Goldman, W. Swartout, and D. S. Wile, "Operational Specification as the Basis for Specification Validation," in *Theory and Practice of Software Technology,* North-Holland Publishing Company, 1983, pp. 21-49.

[Beichter84] F. W. Beichter. O. Herzog, and H. Petzsch. "SIAN-4 - A Software Specification and Design Language,"*IEEE Transactions on Software Engineering,* Vol. SE-10, No. 2, March 1984, pp. 155-162.

[Belady92] L. A. Belady, "The 7 Years of MCC's Innovative Software Technology Program," *American Programmer*, Vol. 15, No. 1, January 1992, pp. 10-15.

[Belkhouche86] B. Belkhouche and J. E. Urban. "Direct Implementation of Abstract Data Types from Abstract Specifications", *IEEE Transactions on*

Software Engineering,Vol. 12, No. 5, May 1986, pp. 649-661.

[Bell77] T. E. Bell, D. C. Bixler, and M. E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January 1977, pp. 49-60.

[Bera90] R. K. Bera "Setting Software Requirements: Scenario for Future Fighters," *Information and Software Technology*, Vol. 32, No. 9. October 1990, pp. 253-257.

[Berzins90] V. Berzins and Luqi, "An Introduction to the Specification Language Spec," *IEEE Software*, Vol. 7, No 2, March 1990, pp. 74-84.

[Biemans86] F. Biemans and P. Blonk, "On the Formal Specification and Verification of CIM Architectures Using LOTOS," *Computers in Industry*, Vol. 7, pp. 491-504.

[Biewald79] J. Biewald, P. Goehner, R. Lauber, and H. Schelling, "EPOS--A Specification and Design Technique for Computer Controlled Real-Time Automation Systems," *Proceedings of the 4th International Conference on Software Engineering*, Munich, Germany, September 1979, pp. 245-250.

[Boehm84] B. W. Boehm, T. E. Gray, and T. Seewaldt, "Prototyping Versus Specifying: A Multiproject Experiment," *IEEE Transactions on Software Engineering*, Vol. 10, No. 3, May 1984, pp. 290-302.

[Bruno86] G. Bruno and G. Marchetto, "Process-Translatable Petri Nets for the Rapid Prototyping of Process Control Systems," *IEEE Transactions on Software Engineering*, Vol. 12, No. 2, February 1986, pp. 346 357.

[Bryant89] B. R. Bryant and A. Pan, "Rapid Prototyping of Programming Language Semantics Using Prolog," *Proceedings of the 13th International Computer Software & Applications Conference*, Orlando, Florida, IEEE Computer Society Press, September 1989, pp. 439-446.

[Budkowski87] S. Budkowski and P. Dembinski, "An Introduction to Estelle: A Specification Language for Distributed Systems," *Computer Networks and ISDN Systems*, Vol. 14, 1987, pp. 3-23.

[Cardenas-Garcia91] S. Cardenas-Garcia and M. V Zelkowitz. "A Management Tool for Evaluanon of Software Designs," *IEEE Transactions on Software Engineering*, Vol. 17, No. 9, September 1991, pp. 961-971.

[Carey90] J. M. Carey, Prototyping: Alternative Systems Development Methodology," *Information and Software Technology*, Vol. 32. No. 2, March 1990, pp. 119-126.

[Catchpole86] P. Catchpole. "Requirements for a Successful Methodology in

Information Systems Design," *Data Processing,* Vol. 28, No. 4, May 1986, pp. 207-210.

[Ceri88] S. Ceri, S. Crespi-Reghizzi, A. Di Maio, and L. A. Lavazza, "Software Prototyping by Relational Techniques: Expenences with Program Construction Systems," *IEEE Transactions on Software Engineering*, Vol. 14, No. 11, November 1988, pp. 1597-1609.

[ChenJ89] J. Chen, J. Wang, and J. Kuo, "An Integrated Framework for Software Prototyping," *Proceedings of the 13th International Computer Software & Applications Conference*, Orlando. Flonda. IEEE Computer Society Press, September 1989, pp. 463-470.

[ChenP90] P. Chen. "Entity-Relationship Approach to Data Modeling," in *System and Software Requirements Engineering*, Edited by M. Dorfman and R. Thayer, IEEE Computer Society Press Tutorial, Order No. 1921, 1990, pp. 238-243.

[Chen88P-M] P.-M. Chen, and C.-R. Chou, "The Requirement Model in a Knowledge-Based Rapid Prototyping System," *Proceedings of the 12th Annual International Computer Software & Applications Conference*, Chicago, Illinois, IEEE Computer Society Press, October 1988, pp. 418-426.

[Cheng82] L. L. Cheng, M. L. Soffa, and Y. H. Yang, "Simulation of an I/O Driven Requirements Language," *Proceedings of the 6th International Computer Software & Applications Conference*, Chicago, Illinois, November 1982, pp. 433-441.

[Cieslak89] R. Cieslak, A. Fawaz, S. Sachs. P. Varaiya, J. Walrand, and A. Li, "The Programmable Network Prototyping System," *Computer*, Vol. 22, No. 5, May 1989, pp. 6776.

[Clapp87] J. A. Clapp, "Rapid Prototyping for Risk Management," *Proceedings of the 11th International Computer Software & Applications Conference*, Tokyo, Japan, IEEE Computer Society Press, October 1987, pp. 17-22.

[Coad90] P. Coad and E. Yourdon, " Object-Oriented Analysis," in *System and Software Requirements Engineering*, Edited by M. Dorfman and R. Thayer, IEEE Computer Society Press Tutorial, Order No 1921, 1990, pp. 272-289.

[Coomber90] C. J. Coomber and R. E. Childs, "A Graphical Tool for the Prototyping of Real- Time Systems," *ACM SIGSOFT Software Engineering Notes*, Vol. 15, No. 2, April 1990, pp. 70- 82.

[Cordy91] J. R. Cordy, E. Promislow, and C. D. Halpern-Hamu, "TXL: A Rapid Prototyping System for Programming Language Dialects,"*Computer Languages,* Vol. 16, No. 1, January 1991, pp. 97-107.

[DavisA82], A. M. Davis. "Rapid Prototyping Using Executable Requirements

Specifications." *ACM SIGSOFT Software Engineering Notes*, Vol. 7, No. 5. December 1982, pp. 39-42.

[DavisA90] A. M. Davis. "The Analysis and Specification of Systems and Software Requirements," in *System and Software Requirements Engineering*, Edited by M. Dorfman and R. Thayer, IEEE Computer Society Press Tutorial, Order No 1921, 1990, pp. 119-144.

[DavisA91] A. M. Davis and P. A. Freeman, "Guest Editor's Introduction-Requirements Engineering," *IEEE Transactions on Software Engineering*, Vol. 17, No. 3, March 1991, pp. 210- 211.

[DavisC77] C. G. Davis and C. R. Vick, "The Software Development System," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January 1977, pp. 69-84.

[Degl'lnnocenti90] M. Degl'Innocenti, G. L. Ferrari, G. Pacini, and F. Turini, "RSF: A Formalism for Executable Requirements Specifications," *IEEE Transactions on Software Engineering*, Vol. 16, No. 11, November 1990, pp. 1235-1245.

[Diaz-Gonalez87a] J. P. Diaz-Gonzalez and J. E. Urban, "ENVISAGER: A Visual, Object-oriented Specification Environment for Real-Time Systems," *Proceedings of the 4th International Workshop on Software Specification and Design*, Monterey, California, EKE Computer Society Press, April 1987, pp. 13-20.

[Diaz-Gonzalez87b] J. P. Diaz-Gonzalez, *The Requirements Engineering of Real-Time Systems: A Temporal Logic Approach*, Ph.D. Dissertation, The Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, Louisiana, December 1987, 166 pp.

[Diaz-Gonzalez91] J. P. Diaz-Gonzalez and J. E. Urban, "Language Aspects of Envisager: An Object-Oriented Environment for the Specification of Real-Time Systems," *Computer Languages*, Vol. 16, No. 1, January 1991, pp. 19-37.

[Doberkat87] E. E. Doberkat and U. Gutenbeil, "SETL To Ada - Tree Transformations Applied," *Information and Software Technology*, Vol. 29, No. 10, December 1987, pp. 548557.

[Dorfman84] M. Dorfman and R. F. Flynn, "Arts - An Automated Requirements Traceability System," *Journal of Systems and Software*, Vol. 4, No. 1, April 1984, pp. 63-74.

[Duke89] E. L. Duke, R. W. Brumbaugh, and J. D. Disbrow, "A Rapid Prototyping Facility for Flight Research in Advanced Systems Concepts," *Computer*, Vol. 22, No. 5, May 1989, pp. 61-66.

[Easterby87] R. Easterby, "Trillium: An Interface Design Prototyping Tool," *Information and Software Technology*, Vol. 29, No. 4, May 1987, pp. 207-213.

[Edmonds81] L. S. Edmonds and J. E. Uban. "A Method for Evaluating Front-End Life Cycle Tools," *Proceedings of the First International Conference on Computers and Applications,* Beijing, China, June 20-22, 1984, pp. 324-331.

[Fraser91] M. D. Fraser. K. Kumar. and V. K. Vaishnavi, "Informal and Formal Requirements Specification Languages: Bridging the Gap," *IEEE Transactions on Software Engineering,* Vol. 17, No. 5, May 1991. pp. 454-466.

[Futatsugi85] K. Futatsugi, J. A. Goguen, I. P. Jouannaud, and J. Meseguer, "Principles of OBJ2," *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages,* New Orleans, Louisiana. January 1985, pp. 52-66.

[Goedicke86] M. Goedicke. "The Use of Formal Requirements Specifications in EDE in a Software Development Environment," *Proceedings of the 10th International Computer Software & Applications Conference,* Chicago, Illinois, IEEE Computer Society Press, October 1986, pp. 190-196.

[GoguenJ9] J. A. Goguen and I. J. Tardo, "An introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications," *Proceedings of the Conference on Specifications of Reliable Software, 1979, pp. 170-189.*

[Goldsack91] S. J. Goldsack and A. C. W. Finkelstein, "Requirements Engineering for RealTime Systems," *Software Engineering Journal*, Vol. 6, No. 3, May 1991, pp. 101-115.

[Gomaa84] H. Gomaa, "A Software Design Method for Real-Time Systems," *Communications of the ACM*, Vol. 27, No. 9, September 1984, pp. 938-949.

[Gupta89] R. Gupta, W. H. Cheng, R. Gupta, I. Hardonag, and M. A. Breuer, "An Object-oriented VLSI CAD Framework," *Computer*, Vol. 22, No. 5, May 1989, pp. 28-37.

[Guttag85] J. V. Guttag, J. J. Horning, and J. M. Wing, "The Larch Family of Languages," *IEEE Software*, Vol. 2, No. 5, September 1985, pp. 24-26.

[Hallmann88] M. Hallmann, "Incorporating Transactions in a Requirement Engineering Method," *Proceedings of the 12th Annual International Computer Software & Applications Conference*, Chicago, Illinois, IEEE Computer Society, October 1988, pp. 121-126.

[Hamilton90] M. H. Hamilton and W. R. Hackler, "001: A Rapid Development Approach for Rapid Prototyping Based on a System That Supports Its Own Life Cycle," *Proceedings of the First International Workshop on Rapid System Prototyping*, Research Triangle Park, North Carolina, June 1990, pp. 46-62.

[Harbert90] A. Harbert, W. Lively, and S. Sheppard, "A Graphical Specification System for User-Interface Design," *IEEE Software*, Vol. 7, No. 4, July 1990, pp. 12-20.

[Harel90] D. Harel, H. Lachover, A. Naarnad, A. Pnueli, M. Poti, R. Sherman, A. ShtullTaunng, and M. Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems." *IEEE Transactions on Software Engineering*, Vol. 16, No. 4, April 1990, pp. 403-414.

[Heitmeyer83] C. L. Heitmeyer and J. D. McLean. "Abstract Requirements Specification: New Approach and Its Application." *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 5, September 1983, pp. 580-589.

[Helmbold88] D. P. Helmbold. *The Meaning of TSL: An Abstract Implementation of TSL-1*, Stanford University Computer Systems Laboratory Technical Report No. CSL-TR-88-353, March 1988, 37 pp.

[Hemenway82] K. Hemenway and L. X. McCusker, "Prototyping and Evaluating a User Interface," *Proceedings of the 6th International Computer Software & Applications Conference*, Chicago, Illinois, November 1982, pp. 175-180.

[Henderson86] P. Henderson, "Functional Programming, Formal Specification, and Rapid Prototyping," *IEEE Transactions on Software Engineering*, Vol. 12. No. 2, February 1986, pp. 241-250.

[Herndon88] R. M. Herndon and V. A. Berzms, "The Realizable Benefits of a Language Prototyping Language," *IEEE Transactions on Software Engineering*, Vol. 14, No. 6, June 1988, pp. 803-809.

[Hoffman88] D. Hoffman and R. Snodgrass, "Trace Specifications: Methodology and Models," *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, September 1988, pp. 1243-1252.

[Hoffman89] D. Hoffman, "Practical Interface Specification," Software Practice and Experience, Vol. 19, No. 2, February 1989, pp. 127-148.

[Holbrook90] Capt. H. Holbrook, "A Scenario-Based Methodology for Conducting Requirements Elicitation," *ACM SIGSOFT Software Engineering Notes*, Vol. 15, No. 1, January 1990, pp. 95- 104.

[Hooper89] J. W. Hooper, "Language Features for Prototyping and Simulation Support of the Software Life Cycle," *Computer Languages*, Vol. 14, No. 2, February 1989, pp. 83-92.

[Hsia86] P. Hsia, A. T. Yaung, and S. H. Jiam, "Requirements Clustering for Incremental Construction of Software Systems," *Proceedings of the 10th International Computer Software & Applications Conference*, Chicago, Illinois,

IEEE Computer Society Press, October 1986, pp. 204- 211.

[i-Logix, Inc.90] i-Logix, Inc., *The STATEMATE Approach to Complex Systems*, i-Logix Report, Burlington, Massachusetts, 1990.

[Ince87] D. C. Ince and S. Hekmatpour, "Software Prototyping - Progress and Prospects," *Information and Software Technology*, Vol. 29, No. l, January/February 1987, pp. 8-14.

[Jackson85] M. I. Jackson, "Developing Ada Programs Using the Vienna Development Method (VDM)," *Software Practice and Experience*, Vol. 15, No. 3, March 1985, pp. 305-318.

[Jaffe91] M. S. Jaffe, N. G. Leveson, M. P. E. Heimdahl, and B. E. Melhart, "Software Requirements Analysis for Real-Time Process-Control Systems," *IEEE Transactions on Software Engineering*, Vol. 17, No. 3. March 1991. pp. 241-258.

[Jarvinen90] H. M. Jarvinen, R. Kurki-Suonio. M. Sakkinen, and K. Systa, "Object-Oriented Specification of Reactive Systems," *Proceedings of the 12th International Conference on Software Engineering*, Nice, France, March 1990, IEEE Computer Society Press, pp. 63-71.

[Jordan89] P. W. Jordan. K. S. Keller, R. W. Tucker, and D. Vogel, "Software Storming: Combining Rapid Prototyping and Knowledge Engineering," *Computer*, Vol. 22, No. 5, May 1989, pp. 39-48.

[Jorgensen86] P. Jorgensen, "Complete Specifications and the Sorcerer's Apprentice Problem," *Proceedings of the 10th International Computer Software & Applications Conference*, Chicago, Illinois, IEEE Computer Society Press, October 1986, pp. 197-204.

[Karakostas89] V. Karakostas, "Requirements for CASE Tools in Early Software Reuse," *ACM SIGSOFT Software Engineering*, Vol. 14, No. 2, April 1989, pp. 39-41.

[Kato87] J. Kato and Y. Morisawa, "Direct Execution of a JSD Specification," *Proceedings of the 11th Computer Software & Applications Conference*, Tokyo, Japan, IEEE Computer Society Press, October 1987, pp. 30-37.

[Knoll89] H. D. Knoll and W. Suk, "A Graphic Language for Business Application Systems to Improve Communication Concerning Requirements Specification with the User," *ACM SIGSOFT Software Engineering Notes*, Vol. 14, No. 6, October 1989, pp. 58-60.

[Krista89] R. Krista and I. Rozman, "A Computer Aided Prototyping Methodology," *ACM SIGSOFT Software Engineering Notes*, Vol. 14, No. 6, October 1989, pp. 68-72.

[Kuo87] J. H. Kuo and H.-C. Tu, "Prototyping a Software Information Base for Software Engineering Environments," *Proceedings of the 11th International Computer Software & Applications Conference*, Tokyo, Japan, IEEE Computer Society Press, October 1987, pp. 3844.

[Lea90] R.-J. Lea and C.-G. Chung, "Rapid Prototyping from Structured Analysis: Executable Specification Approach," *Information and Software Technology*, Vol. 32, No. 9, November 1990, pp. 589-597.

[Lee85] S. Lee, "On Executable Models for Rule-Based Prototyping," *Proceedings of the 8th International Conference on Software Engineering*, 1985, London, England. pp. 210-215.

[Leszczylowski89] J. Leszczylowski and J. M. Bieman, "Prosper: A Language for Specification by Prototyping," *Computer Languages*, Vol. 14, No. 3, Apnl 1989, pp. 165-180.

[Lewis89] T. G. Lewis, F. Handloser III, S. Bose, and S. Yang, "Prototypes From Standard User interface Management Systems," *Computer,* Vol. 22, No. 5, May 1989, pp. 51-60.

[Lintulampi90] R. Lintulampi and P Pulli, Graphics Based Prototyping of Real-Time Systems." *Proceedings of the First International Workshop on Rapid System Prototyping*, Research Triangle Park, North Carolina, June 1990, pp. 128-137

[Lor91] K.-W E. Lor, "Operational Definitions for System Requirements as the Basis of Design Automation." *Software Practice and Experience*, Vol. 21, No. 10, October 1991, pp. 1103-1124.

[Loucopoulos89] P. Loucopoulos and R. E. M. Champion, "Knowledge-Based Support for Requirements Engineering," *Information and Software Technology*, Vol. 31, No. 3, April 1989, pp. 124-135.

[Luckham85] D. C. Luckham and F. W. von Henke, " An Overview of Anna, A Specification Language for Ada," *IEEE Software.* Vol. 2, No. 2, March 1985, pp. 9-22.

[Luckham87] D. C. Luckham. R. Neff, D. S. Rosenblum. "An Environment for Ada Software Development Based on Formal Specification: Status and Development Plan," *ACM SIGADA Ada Letters*, Vol. ii, No. 3, March 1987, pp. 94-106.

[Ludewig83] J. Ludewig, "ESPRESO - A System for Process Control Software Specification," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 4, July 1983, pp. 427-436.

[Ludewig85] J. Ludewig, M. Glinz, H. Huser, G. Matheis, H. Matheis, and M. F. Schmidt, "SPADES - A Specification and Design System and Its Graphical Interface," *Proceedings of the 8th International Conference on Software Engineering*, August 1985, pp. 83-89.

[Luqi88a] Luqi and M. Ketabchi. "A Computer-Aided Prototyping System," *IEEE Software,* Vol. 5, No. 2, March 1988, pp. 66-72.

[Luqi88b] Luqi and V. Berzins, "Rapidly Prototyping Real-Time Systems," *IEEE Software*, Vol. 2, No. 5, September 1988, pp. 25-36.

[Luqi88c] Luqi, V. Berzins, and R. Yeh, "A Prototyping Language for Real-Time Software," *IEEE Transactions on Software Engineering*, Vol. 14, No 10, October 1988, pp. 1409-1423.

[Luqi88d] Luqi, "Knowledge-Based Support for Rapid Software Prototyping," *IEEE Expert*, Vol. 3, No. 4, December 1988, pp. 9-18.

[Luqi89a] Luqi, "Software Evolution Through Rapid Prototyping," *Computer*, Vol. 22, No. 5, May 1989, pp. 13-25.

[Luqi89b] Luqi and Y. J. Lee, "Interactive Control of Prototyping Process," *Proceedings of the 13th International Computer Software & Applications Conference*, Orlando, Florida, IEEE Computer Society Press, September 1989, pp. 447-454.

[Luqi90] Luqi, P. D. Barnes, and M. Zyda, "Graphical Tool for Computer-Aided Prototyping," *Information and Software Technology,* Vol. 32, No. 3, April 1990, pp. 199-206.

[Luqi92] Luqi, "Computer-Aided Prototyping for a Command-and-Control System Using CAPS," *IEEE Software,* Vol. 9, No. 1, January 1992, pp. 56-67.

[Mayhew89] P. I. Mayhew, C. I. Worsley, and P. A. Dearnley, "Control of Software Prototyping Process: Change Classification Approach," *Information and Software Technology,* Vol. 31, No. 2, March 1989, pp. 59-67.

[Mayhew90] P. J. Mayhew and P. A. Dearnley, "Organization and Management of Systems Prototyping," *Information and Software Technology,* Vol. 32, No. 4, May 1990, pp. 245-252.

[Michaelson88] G. Michaelson, "Interpreter Prototypes From Language Definition Style Specifications," *Information and Software Technology,* Vol. 30, No. 1, January/February 1988, pp. 23-31.

[Misra88] S. K. Misra and P. J. Jalics, "Third-Generation Versus Fourth-Generation Software Development," *IEEE Software,* Vol. 5, No. 4, July 1988, pp. 8-14.

[Mosse90] D. Mosse, O. Gudmundsson, and A. K. Agrawala, "Prototyping Real Time Operating Systems: A Case Study," *Proceedings of the First International Workshop on Rapid System Prototyping*, Research Triangle Park, North Carolina, June 1990, pp. 144-154.

[Musser80] D. R. Musser, "Abstract Data Type Specification in the AFFIRM System," *IEEE Transactions on Software Engineering*, SE-6, No. 1, January 1980, pp. 24-32.

[Norris90] M. Norris, "Z (A Formal Specification Method)," in *System and Software Requirements Engineering*, Edited by M. Dorfman and R. Thayer, IEEE Computer Society Press Tutorial, Order No 1921, 1990, pp. 345-369.

[Payton82] T. Payton, S. Keller, J. Perkins, S. Rowan, and S. Mardinly, "SSAGS: A Syntax and Semantics Analysis and Generation System," *Proceedings of the 6th International Computer Software & Applications Conference*, Chicago, Illinois, November 1982, pp. 424-432.

[Prizant86] A. Prizant, "Prototyping Counterproductive?," *DataProcessing*, Vol. 28, No. 7, September 1986, pp. 379.

[Purtilo91] J. M. Purtilo and P. Jalote, "An Environment for Prototyping Distributed Applications," *Computer Languages*, Vol. 16, No. 3/4, 1991, pp. 197-207.

[Ratcliff88] B. Ratcliff, "Early and Not-So-Early Prototyping - Rationale and Tool Support," *Proceedings of the 12th Annual International Computer Software & Applications Conference*, Chicago, Illinois, IEEE Computer Society Press, October 1988, pp. 127-134.

[Reubenstein91] H. B. Reubenstein and R. C. Waters, "The Requirements Apprentice: Automated Assistance for Requirements Acquisition," *IEEE Transactions on Software Engineering*, Vol. 17, No. 3, March 1991, pp. 226-240.

[Rolland83] C. Rolland and G. Benci," An Event-Driven Methodology for Technical Software Design," Proceedings of Real-Time Systems Symposium, December 1983, pp. 81-87.

[Roman85] G.-C. Roman, "A Taxonomy of Current Issues in Requirements Engineering," *Computer*, Vol. 18, No. 4, April 1985, pp. 14-21.

[Ross77a] D. T. Ross, "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January 1977, pp. 16-34.

[Ross77b] D. T. Ross and K. E. Schoman, Jr., "Structured Analysis for

Requirements Definition," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January 1977, pp. 6- 15.

[Rzepka85] W. Rzepka and Y. Ohno, "Guest Editor's Introduction - Requirements Environments: Software Tools for Modeling User Needs," *Computer*, Vol. 18, No. 4, April 1985, pp. 9-12.

[Scheffer85] P. A. Scheffer, A. H. Stone III, and W. E. Rzepka, "A Case Study of SREM," *Computer*, Vol. 18, No. 4, April, 1985, pp. 47-54.

[Schmidt91] H. W. Schmidt, "Prototyping and Analysis of Non-Sequential Systems Using Predicate-Event Nets," *Journal of Systems and Software*, Vol. 15, No. 1, 1991, pp. 43-62.

[Sievert85] G. E. Sievert and T. A. Mizell, "Specification-Based Software Engineering With TAGS," *Computer*, Vol. 18, No. 4, April 1985, pp. 56-65.

[Skillicorn89] D. B. Skillicorn and J. I. Glasgow, "Real-Time Specification Using Lucid," *IEEE Transactions on Software Engineering*, Vol. 15, No. 2, February 1989, pp. 221-229.

[Slusky87] L. Slusky, "Integrating Software Modelling and Prototyping Tools," *Information and Software Technology*, Vol. 27, No. 7, September 1987, pp. 79-87.

[Smith88] S. L. Smith and S. L. Gerhart, "STATEMATE and Cruise Control: A Case Study," *Proceedings of the 12th International Computer Software & Applications Conference*, Chicago, Illinois, IEEE Computer Society Press, October 1988, pp. 49-56.

[Staknis90] M. E. Staknis, "Software Quality Assurance Through Prototyping and Automated Testing," *Information and Software Technology*, Vol. 32, No. 1, January/February 1990, pp. 26- 33.

[Stasko90] J. T. Stasko, "A Practical Animation Language for Software Development," *Proceedings International Conference on Computer Languages*, New Orleans, Louisiana, IEEE Computer Society Press, March 1990, pp. 1-10.

[Stephens90] M. A. Stephens and P. E. Bates, "Requirements Engineering by Prototyping: Experiences in Development of Estimating System," *Information and Software Technology*, Vol. 32, No. 4, May 1990, pp. 253-257.

[Sunshine82] C. A. Sunshine, D. H. Thompson, R. W. Erickson, S. L. Gerhart, and D. Schwabe, "Specification and Verification or Communication Protocols in AFFIRM Using State Transition Models," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 5, September 1982, pp. 460-489.

[Tate90a] G. Tate and J. Verner, "Case Study of Risk Management, Incremental

Development, and Evolutionary Prototyping," *Information and Software Technology*, Vol. 32, No. 3, April 1990, pp. 207-214.

[Tate90b] G. Tate, "Prototyping: Helping to Build the Right Software," *Information and Software Technology*, Vol. 32, No. 4, May 1990, pp. 237-244.

[Teichroew77] D. Teichroew and E. A. Hershey III, "PSLJPSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January 1977, pp. 41 48.

[Terwilliger89] R. B. Terwilliger and R. H. Campbell. " PLEASE: Executable Specifications for Incremental Software Development." *Journal of Systems and Software*, Vol. 10, No. 2, 1989, pp. 97-112.

[Thayer90] R. H. Thayer and M. Dorfman (Eds.), *System and Software Requirements Engineering*, IEEE Computer Society Press Tutorial, Order No. 1921, 1990.

[Topping87] P. Topping, J. McInroy, W. Lively, and S. Sheppard, "Express - Rapid Prototyping and Product Development via Integrated, Knowledge-Based Executable Specifications," *Proceedings of the 1987 Fall Joint Computer Conference*, Dallas, Texas, IEEE Computer Society Press, pp. 3-9.

[Tozer87] J. E. Tozer, "Prototyping as a System Development Methodology: Opportunities and Pitfalls," *Information and Software Technology*, Vol. 29, No. 5, June 1987, pp. 265-269.

[Tsai88] J. J. P. Tsai, M. Aoyama, and Y. L. Chang, "Rapid Prototyping Using FRORL Language," *Proceedings of the 12th International Computer Software & Applications Conference*, Chicago, Illinois, IEEE Computer Society Press, October 1988, pp. 410-417.

[Tsai89] J. J. P. Tsai and T. Weigert, "Exploratory Prototyping Through the Use of Frames and Production Rules," *Proceedings of the 13th International Computer Software & Applications Conference*, Orlando, Florida, IEEE Computer Society Press, September 1989, PP. 455-462.

[Tsai91] J. J. P. Tsai and T. Weigert, "HCLIE: A Logic-based Requirement Language for New Software Engineering Paradigms," *Software Engineering Journal*, Vol. 6, No. 4, July 1991, pp. 137-151.

[TsaiS-T90] S.-T. Tsai, C.-C. Yang, and C.-C. Lien, "Automated Retrieval of Consistent Documentation for Rapid Prototyping Systems and Software Maintenance," *Information and Software Technology*, Vol. 32, No. 8, October 1990, pp. 521-530.

[Urban77] J. E. Urban. “A Specification Language and Its Processor,” Ph. D.

Dissertation. University or Southwestern Louisiana, December 1977, 179 pp.

[Urban85] S. D. Urban, I. E. Urban, and W. D. Dominick, "Utilizing an Executable Specification Language for an Information System," *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, July 1985, pp. 598-605.

[Urban90] J. E. Urban, "The Descartes Specification Language," in *System and Software Requirements Engineering*, Edited by M. Dorfman and R. Thayer, IEEE Computer Society Press Tutorial. Order No 1921, 1990, pp. 331-344.

[van Delft89] A. I. E. van Delft. "Express: Proposal for Uniform Notations," *Information and Software Technology*, Vol. 31, No. 3, April 1989, pp. 143-159.

[Walters91a] N. Walters, "Requirements Specification for Ada Software Under DoD- STD2167A," *Journal of Systems and Software*, Vol. 15, No. 2, May 1991, pp. 173-183.

[Walters91b] N. Walters. "An Ada Object-Based Analysis and Design Approach," *ACM SIGADA Ada Letters*, Vol. XI, No. 5, July 1991, pp. 62-78.

[Wang88] Y. Wang, "A Distributed Specification Model and Its Prototyping," *IEEE Transactions on Software Engineering*, Vol. 14, No. 8, August 1988, pp. 1090-1097.

[Wasserman86] A. I. Wasserman, P. A. Pircher, D. T. Shewmake, and M. L. Kersten, "Developing Interactive Information Systems with the User Software Engineering Methodology," *IEEE Transactions on Software Engineering*, Vol. 12, No. 2, February 1986, pp. 326-345.

[Wing88] J. M. Wing, "A Study of 12 Specifications of the Library Problem," *IEEE Software*, Vol. 5, No. 4, July 1988, pp. 66-76.

[Yang90] Y. Yang, "Experimental Rapid Prototype of undo Support," *Information and Software Technology*, Vol. 32, No. 9, November 1990, pp. 625-635.

[Yeh82] R. T. Yeh, "Requirements Analysis - A Management Perspective," *Proceedings of the 6th International Computer Software & Applications Conference*, Chicago, Illinois, IEEE Computer Society, November 1982, pp. 410-416.

[Yeh84] R. T. Yeh, P. Zave, A. P. Conn, and G. E. Cole, Jr., "Software Requirements: New Directions and Perspectives," *Handbook of Software Engineering*, C. R. Vick and C. V. Ramamoorthy (Eds.), Van Noserand Reinhold Company, New York, 1984, pp. 519-543.

[Zave91] P. Zave. "An Insider's Evaluation of Paisley," *IEEE Transactions on Software Engineering*, Vol. 17, No. 3, March 1991, pp. 212-225.

[Zhou90] W. Zhou, "PM: A System for Prototyping and Monitoring Remote Procedure Call Programs," *ACM SIGSOFT Software Engineering Notes*, Vol. 15, No. 1, January 1990, pp. 59-63.

[Zucconi89] L. Zucconi, "Techniques and Experiences Capturing Requirements for Several Real-Time Applications," *ACM SIGSOFT Software Engineering Notes, Vol*. 14, No. 6, October 1989, pp. 51-55.

Appendix B.
LIST OF ACRONYMS

Anna	ANNotated Ada
ARTS	Automated Requirements Traceability System
BMDATC	US Army Ballistic Missile Defense Advanced Technology Center
C3I	Command, Control, Communications, and Intelligence
CAPS	Computer-Aided Prototyping System
DARPA	Defense Advanced Research Projects Agency
DARTS	Design Approach for Real-Time Systems
DEC	Digital Equipment Corporation
ENVISAGER	ENvironment for the VIsual Specification And Graphical Execution of Requirements
GSS	Graphical Specification System
IWSSD	International Workshop on Software Specification and Design
NSF	National Science Foundation
PROSPER	PROtotypes and SPEcifications with Relative types
PSDL	Prototype System Description Language
REVS	Requirements Engineering Validation System
RSL	Requirements Specification Language
SA	Structured Analysis
SBRE	Scenario Based Requirements Elicitation
SLAN-4	Software Language-4
SPADES	SPecification And DEsign System
SREM	Software Requirements Engineering Methodology
TSL	Task Sequencing Language
VDM	Vienna Development Method

About the Author

Joseph Urban is a professor of computer science at Arizona State University, responsible for the Software Engineering Research Group. He has worked at the University of Miami, the University of Southwestern Louisiana, and part-time at the University of South Carolina while with the US Army Signal Center. He has published over 50 technical papers. His research areas include software engineering, computer languages, data engineering, and distributed computing. His research efforts have been supported primarily through industry. He has conducted research for Bell-Northem Research, Florida High Technology

and Industry Council, Lockheed, and the National Science Foundation, and served on the Air Force Studies Board of the National Research Council.

He recently served as a co-conference general chair for the 11th Annual International Phoenix Conference on Computers and Communications, April 1992, Scottsdale, Arizona and as a program co-chair for the 12th International Conference on Distributed Computing Systems, June 1992, Yokohama, Japan. He was a member of the IEEE Computer Society Board of Governors, chair of the Awards Committee, and a Computer Society representative on the IEEE Publications Board. He has served as the Computer Society's elected first and second vice president, responsible for conferences and tutorials, and as treasurer and Finance Committee chair. He initiated and chaired the IEEE Computer Society's Technical Committee on Computer Languages and chaired the Publications Planning Committee. He chaired and lectured in the Chapter Tutorials and Distinguished Visitors Programs. In addition, he has served as general chair for the following conferences: IEEE Computer Society 6th (1990) International Conference on Data Engineering, Los Angeles, California; IEEE Computer Society / ACM 1988 International Symposium on Databases in Parallel and Distributed Systems, Austin, Texas; IEEE Computer Society 1986 International Conference on Computer Languages, Miami Beach, Florida; IEEE Computer Society 1984 Ada Applications and Environments Conference; and IEEE Computer Society Symposium on Logic Programming, Atlantic City, New Jersey. He has served on the Editorial Boards of IEEE Transactions on Software Engineering and IEEE Expert.

He earned a BS degree from the Florida Institute of Technology, an MS degree from the University of Iowa, and a PhD degree from the University of Southwestern Louisiana, all in computer science. He has received the Computer Society's Meritorious and Distinguished Service Awards, a Distinguished Professor Award while at the University Southwestern Louisiana, and an Association for Computing Machinery Doctoral Forum Award for one of the four best Ph.D. theses produced during the 1977-1978 academic year.