1998-04

Formal Models and Prototyping

Luqi

Technische Universistät München

Luqi. "Formal models and prototyping." Requirements Targeting Software AND Systems Engineering, RTSE '97 TUM-19807 (1997): 183-194.
https://hdl.handle.net/10945/65146

TUM

INSTITUT FÜR INFORMATIK

RTSE'97 – Workshop on "Requirements Targeting Software and Systems Engineering"

Manfred Broy
Bernhard Rumpe

TUM-I9807
April 98

TECHNISCHE UNIVERSITÄT MÜNCHEN

Formal Models and Prototyping*

Luqi
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Abstract

Rapid prototyping is a promising approach for formulating accurate software requirements, particularly for complex systems with hard real-time constraints. Computer aid is needed for realizing the potential benefits of this approach in practice, because the problems associated with software evolution are greatly amplified in the context of iterative prototyping and exploratory design.

Our computer-aided prototyping system CAPS provides automated support for many aspects of requirements analysis and software prototyping, including: (1) maintaining logical dependencies between assumptions about needs of different groups, software requirements, and design decisions, (2) managing design history, alternatives and dependencies, (3) planning, assigning and scheduling job assignments for teams of designers in the presence of uncertainty, (4) checking and propagating design constraints, (5) maintaining consistency between graphical and text views of a design, (6) constructing real-time schedules, (7) generating control code, and (8) retrieving and instantiating reusable software components.

The principles and methods that make this possible and the practical application of the system are explained via examples.

1 Introduction

The software industry remains far short of the mature engineering disciple needed to meet the demands of our information age society. Symptoms of the problem are large sums spent on cancelled software projects [38], costly delays [19], and software reliability problems [13].

Lack of formalization of rapidly emerging application areas makes software engineering more difficult than other engineering disciplines. Requirements for complex systems are nearly always problematic initially and evolve throughout the life of the systems. Requirements and specification problems have been found to be the dominant cause of faults in the Voyager/Galileo software [34], and we believe this applies to most large and complex systems.

*This research was supported in part by the National Science Foundation under grant number CCR-9058453 and by the Army Research Office under grant number 30989-MA.

Evolutionary prototyping can alleviate this problem by providing an efficient approach to formulating accurate software requirements [27]. Simple models reflecting the main issues associated with the proposed system are constructed and demonstrated, and then reformulated to better match customer concerns, based on specific criticisms and the issues they elicit. This process aids understanding because independent issues are separated and treated in isolation as much as possible, via communication based on the simplest models possible. The models are refined only as needed to resolve open issues, and the issues arising at one level of detail are resolved as much as possible before considering the next level of detail, or the next aspect of the system. This helps to focus the attention of the customers, designers, and analysts because only a few selected aspects of the system are changing at any point in the process.

Automation is necessary to enable the rapid, economical and effective change needed for evolutionary prototyping. Our hypothesis has been that increasing the degree of automation for system development and evolutionary prototyping should improve the quality of the systems produced. A sound basis for the engineering automation is needed to realize evolutionary prototyping for large and complex systems, which typically have real-time constraints. We have explored formal models of various aspects of software development and evolution to achieve reliable and quantifiable automation of subtasks. Formal models have enabled analysis and assessment of the accuracy and efficiency of proposed algorithms and heuristics.

It has been necessary to interleave this theoretical work with experimental validation and adjustment of the models to better fit practical reality. This has been necessary because software development and evolution are extremely complex problem domains, and engineering automation systems have correspondingly complex requirements that strongly manifest all of the difficulties identified above. Thus we have applied the evolutionary prototyping approach to the development of techniques and software for supporting the evolutionary prototyping approach itself. We have found this strategy successful for developing accurate models, effective automation and decision support methods for evolution of software and system requirements. This paper summarizes our experiences and presents some of our recent progress on carrying out the plan outlined above.

The rest of the paper is organized as follows. Section 2 describes our strategies for achieving automation support for evolutionary prototyping and summarizes progress to date. Section 3 discusses a formal model of software evolution and explores some automated processes that can be supported by the model. Section 4 illustrates our ideas with an example. Section 5 contains conclusions.

2 Strategy for Automation Support

The main components of our strategy are developing languages and methods based on formal models of selected aspects of the problem. In each case we sought the simplest models adequate for achieving our purposes, and based the languages and methods on these models. We started with the simplest possible models and refined them only as needed, based on experimental application of the models to assess their adequacy. Our guiding principle was to avoid model features unless we has a convincing practical scenario that required those features. Consequently we were always searching for simplifications and reformulated models whenever we found a way to eliminate a model concept. This was done because we wanted the resulting methods and tools to be easy to use and learn. We expected simpler models to speed up the processes of analysis and design by reducing the number of mandatory choices. This is particularly appropriate in the context of prototyping, where it is important to get the major decisions correct rapidly, without spending effort on fine-tuning. Our experience has confirmed this hypothesis. We have also found that removing concepts from the models and the attention of the designer can introduce stringent requirements for design automation capabilities. Removed design attributes must be derived automatically, accurately, and in a way that provides good designs.

The first area to be modeled was the behavior of real-time systems, because the prototyping approach requires the ability to demonstrate proposed system behavior. The simplest formulation we could find was a refinement of data flow models that incorporates declarative control and timing constraints. The prototype system description language PSDL [25] was developed based on this model. The model was extended to include distributed computation [30] and a formal semantics of the language was developed [22]. The model and language have been found to be adequate for representing a variety of complex systems, including a generic C3I station [29] and a wireless acoustic monitor for preventing sudden infant death syndrome [36].

Real-time scheduling and software integration are other key issues for rapid realization of complex systems. We developed related models in these two areas, based on the model of system behavior.

Real-time scheduling depends on models of the timing requirements and on models of the capabilities of the target hardware. The behavioral model underlying PSDL contains a model of real-time requirements, which we extracted for this purpose [26]. This model was used to develop our initial scheduling methods, and it proved adequate. The initial hardware model was empty, which was adequate for scheduling with respect to fixed, single processor architectures. We realized that scheduling depended on hardware models when we started addressing scheduling methods for more general hardware configurations. We developed a series of more sophisticated hardware models [30], and found that these together with the original model of real-time requirements were adequate for supporting scheduling methods for multi-processor and distributed target hardware configurations [7, 32]. Ongoing work is exploring models and methods that can schedule larger distributed real-time computations within practical resource limits.

Software integration is the process of ensuring that all the parts of a software system work together to achieve their intended purpose. Software integration depends on models of interactions between subsystems and control constraints, including those derived from timing requirements and the schedules used to realize them. We addressed software integration by developing software architectures and methods for architecture-based program generation. Automated program generation is necessary in our context because we had to support rapid, low cost change, and small changes to timing requirements can affect large portions of the code.

The software architecture for prototypes embodies a general structure for realizing interactions and real-time schedules for systems that have a mix of time-critical and non time-critical computations. The structure used to automatically realize connections between subsystems was derived from the system interaction part of the behavioral model underlying PSDL. The structure that realizes the schedules uses a high-priority thread for computations with hard deadlines and a low-priority thread for computations without deadlines.

The software architecture was implicitly defined by a program generator driven by a description of desired system behavior expressed in PSDL and a real-time schedule constructed by a scheduling algorithm. This program generator was itself generated using an attribute grammar processor. This approach works but is not particularly elegant or easy to adapt to other problems.

Our initial capability for generating executable prototypes from simple and quickly constructible models of the problem domain enabled experimental validation of the conjecture that prototyping and demonstrations of systems behavior were valuable aids to requirements determination. The initial experiments supported the validity of this conjecture, which motivated us to put more effort into software reuse and evolution.

Software reuse is a critical part of prototyping for real-time systems because efficiency is of the essence in the time-critical parts of these systems. The highest levels of efficiency can only be achieved by intensive engineering and refinement of sophisticated algorithms and data structures, which usually takes large amounts of time and effort, and produces designs that depend on intricate chains of reasoning. The easiest way to take advantage of such components in a process that must be cheap and rapid is to use a previously constructed library of standard and well-optimized components. Thus we explored formal models of how such libraries could be organized and searched to quickly find the most appropriate components for each particular context [24]. Search methods must trade off precision (retrieving only relevant components) against recall (finding components if they are relevant). We have developed a software component search method that can simultaneously achieve high levels of precision and recall, based on algebraic queries representing symbolic test cases.

Software evolution is a critical aspect of prototyping [27]. In the early stages of requirements formulation the purposes of the proposed system are highly uncertain and major changes are expected. Planning, version control, team coordination, and project management are key issues in this context. Another important issue is how to repeatedly and rapidly change a design without having it degenerate into an unstructured maze that cannot be quickly understood and modified. The next section summarizes our progress on software evolution.

3 Software Evolution

Our initial step towards formalizing software evolution in the large was a graph model of the evolution history [28]. This work led to the insights that the essence of project history lies in dependencies among versions of project documents and the activities that produce them, that the formal structures of project history and project plans are essentially the same, and that integrated modeling and support for software configuration management and project management enables higher automation levels for both [1]. More recent work suggested that hypergraphs may be useful [33], and that integration with personnel models and rationale models enables decision support for the problematic early stages of critique analysis and change planning [8].

To achieve simplicity, we seek to model the products and processes involved in software evolution using a minimal set of general object types, and introduce specialized subclasses only when necessary for accurate modeling. The current version of the model has only three main types: component, step, and person.

The type component represents any kind of versioned software-related object, including critiques, issues, requirements, designs, programs, manuals, test cases, plans, etc. These are the information products produced by software evolution processes.

The type step represents instances of any kind of scheduled software evolution activity, such as analysis, design, implementation, testing, inspection, demonstration, etc. Steps are activities that are usually carried out by people, and may be partially or completely automated. When viewed in the context of evolution history, steps represent dependencies among components. Steps that are not yet completed represent plans. Steps are a subclass of component because they can have versions, to provide a record of how the project plans evolved.

The type person represents the people involved in the software evolution activity, including the stakeholders of the software system, software analysts, designers, project managers, testers, software librarians, system administrators, etc. We need to represent the people involved to be able to trace requirements back to the original raw data, and to link it to the roles the authors of critiques play in the organizational structure. This is a part of the rationale of the system that helps to identify viewpoints and analyze tradeoffs between conflicting requirements. The people in the development team must be modeled because of concerns related to project scheduling and authorization to access project information. Person is also a subclass of component, and therefore versioned, to provide a record of how the roles and qualifications of the people involved in the project change with time.

We have recently developed an improved model of system evolution that better accounts for hierarchical structures of components and steps. The associated refinement concept is useful for helping developers and planners to cope with the complexity of large projects. This model is summarized as follows.

An **evolution record** is a labeled acyclic directed hypergraph $[N, E, I, O, C, S]$ where

1. N is a set of **nodes**, representing unique identifiers for components,

2. E is a set of **edges**, representing unique identifiers for steps,

3. $I : E \to 2^N$ is a function giving the set of **inputs** of each edge,

4. $O : E \to 2^N$ is a function giving the set of **outputs** of each edge, such that $O(e) \cap O(e') \neq \emptyset$ implies $e = e'$,

5. $C : N \to$ *component* is a function giving the **component** associated with each node, and

6. $S : E \to$ *step* is a function giving the **step** associated with each edge.

The hypergraph must be acyclic because its edges represent input/output dependencies for the processes that create components. These dependencies induce precedence constraints for the project schedule, because an activity cannot start until all of its input components are available. The restriction on the outputs says that each component is produced by a unique step. This establishes clear lines of responsibility and produces a record of authorship when each step completes.

Let H denote the set of evolution records.

A **hierarchical evolution record** is an acyclic directed graph $[n, e]$ with label maps h, r and decomposition maps d_n, d_e where

1. n is a set of **nodes** representing unique identifiers for evolution records.

2. e is a set of **edges** representing unique identifiers for evolution record refinements,

3. $h : n \rightarrow H$ is a function giving the **evolution record** associated with each node, such that $(n_1, n_2) \in e$ implies $h(n_1)$ is a subhypergraph of $h(n_2)$. This means that $h(n_1).N \subseteq h(n_2).N$, $h(n_1).E \subseteq h(n_2).E$, $h(n_1).I \subseteq h(n_2).I$, $h(n_1).O \subseteq h(n_2).O$, $h(n_1).C \subseteq h(n_2).C$, and $h(n_1).S \subseteq h(n_2).S$.

4. $r : e \rightarrow step$ is a function giving the **step** that is refined by each edge,

5. $d_n : N \rightarrow 2^N$ is a function giving the set of **subcomponent nodes** of each component node appearing in the evolution record $h(n_i)$ for any node $n_i \in n$, where $N = \bigcup_{n_i \in n} h(n_i).N$.

6. $d_e : E \rightarrow 2^E$ is a function giving the set of **substep edges** of each step edge appearing in the evolution record $h(n_i)$ for any node $n_i \in n$, where $E = \bigcup_{n_i \in n} h(n_i).E$.

7. The graph has a single root (a node with no incoming edges) and a single leaf (a node with no outgoing edges).

8. Any two paths p_1 and p_2 from the root node with the same step label set $\{r(e)|e \in p_1\} = \{r(e)|e \in p_2\}$ end in the same node.

9. If $(n_1, n_2) \in e$, then there is an $E_1 \in h(n_1).E$ with $S(E_1) = r(e)$, $\emptyset \neq d_e(E_1) \subseteq h(n_2).E$, and for each $E_2 \in d_e(E_1)$, $I(E_2) \subseteq \bigcup_{N_1 \in I(E_1)} d_n(N_1) \subseteq h(n_2).N$ and $O(E_2) \subseteq \bigcup_{N_1 \in O(E_1)} d_n(N_1) \subseteq h(n_2).N$.

Each node of a hierarchical evolution record represents a view of the evolution history. The root node is the most abstract view, containing only the top level steps and the top level components those steps produce. The leaf node is the most detailed view, which contains the top level steps and components together with all direct and indirect substeps and subcomponents.

A step is refined by adding all of its substeps to the evolution record, along with the input and output components of the substeps. The last condition in the definition says that the step associated with the link between two views must be decomposed into at least one substep in the detailed view, that the inputs and outputs of the substeps must be subcomponents of the inputs and outputs of the superstep, and that the input and output components of the substeps must appear in the detailed view.

The hierarchical evolution record has a large number of nodes, which are not intended to be stored explicitly in an implementation. The model is intended as a framework for navigation through the possible views of the evolution record at different levels of abstraction. Practical implementations will materialize only those view nodes that are visited.

This model can be used to automatically schedule steps, automatically locate and deliver the proper versions of the input components to the developer assigned to carry out the step, and to automatically check in the new components produced when the step is completed. It can also be used to generate default plans, to maintain the consistency of plans, and to help managers and developers navigate through the plan and document structures of an evolutionary prototyping or development effort.

4 Example

Figure 1 shows an example of a top level evolution record. In this example, the first version of the requirement ($R1$) is used to derive the first version of the prototype ($P1$), which is demonstrated to system stakeholders and elicits the criticism ($C1$). When a step to derive the second version of the requirement ($R2$) from the criticism is proposed, the system automatically proposes a step to create the second version of the prototype ($P2$), because the prototype depends on the requirement and the requirement will be updated. The proposed steps will be scheduled automatically when they are approved by the project management.

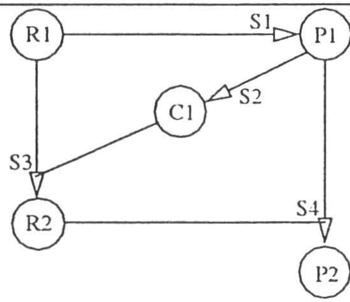

Figure 1: Top Level Evolution Record

Figure 2 shows the refinement of step $S1$ of the top level evolution record shown in Figure 1. Both $S1$ and its substeps $S1.1$ and $S1.2$ are present in the refined evolution record. The top level steps are shown with thicker lines. The component $R1$ is decomposed into the subcomponents $Ra1$ and $Rb1$ because these components are inputs to the substeps, and $P1$ is decomposed into $Pc1$ and $Pd1$ because these are the outputs of the substeps.

Figure 3 shows a further refinement of the evolution record shown in Figure 2 that expands all of the top-level steps. We have left out the top level steps to avoid cluttering the diagram. Note that the subrequirement $Rb1$ is shared by both versions of the requirement R, because it is not affected by the elicited criticism, and that the subsystem $Pd1$ of the prototype that depends only on this subrequirement is also shared by both versions

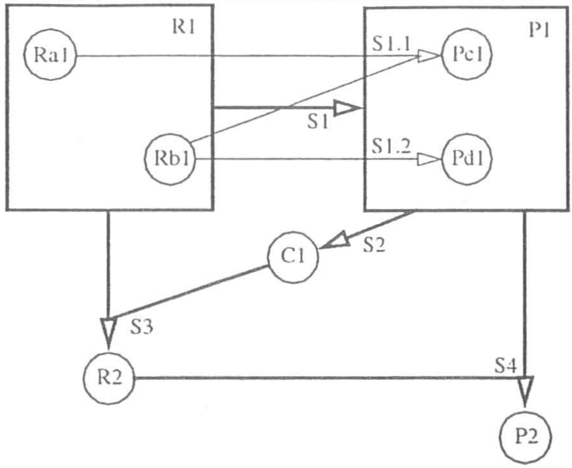

Figure 2: Refinement of Step S1

of the prototype P. Our goal is to provide tools based on this model that will make it easier to discover and manage large scale structures of this variety.

The decomposition mappings for the subcomponents are denoted by geometrical containment in the figures. The decomposition relations for the steps are indicated only via the structure of the step names. Note that the graphical display would get crowded if the decomposition relations were explicitly displayed as hyper-edges, even for this very small example. In realistic situations, there can be many more nodes in the evolution records. We are currently exploring automatic mechanisms for determining and displaying small neighborhoods of these structures that are relevant to particular planning and analysis tasks and are small enough to be understood. Some initial results along these lines can be found in [23].

5 Conclusions

Our previous research has explored formal models of the chronological evolution history [28]. This model has been applied to automate configuration management and a variety of project management functions [1]. The ideas presented in this paper provide a basis for improving these capabilities, particularly in the area of computer aid for understanding the record of the evolution of the system to extract useful information from it. Some recent work on improving the project scheduling algorithms based on these models has enabled scheduling 100,000 tasks in less than a minute [14]. These results suggest that the project scheduling support will scale up to projects of formidable size.

We are currently working on models and notations that support explicit definitions of software architectures for solving given classes of problems independently from the rules

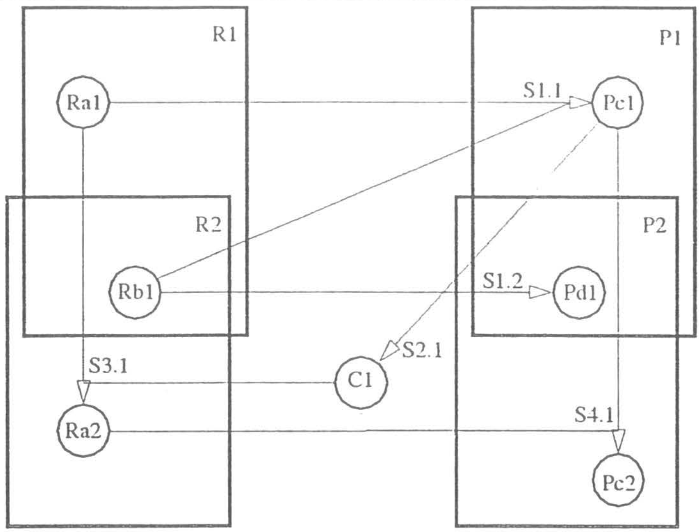

Figure 3: Further Refined Evolution Record

that determine a particular instance of the architecture for solving a given instance of the class of problems. This should make it easier for software architectures and associated program generation capabilities to evolve.

Architecture evolution provides a practical path for quickly obtaining automation capabilities for new problem domains, and to gradually improve those capabilities by adding solution techniques that expand the problem domain and incorporating optimizations for specialized subproblems that improve performance.

Formalizing these aspects of software architectures and developing the corresponding engineering automation methods will eventually enable us to certify that all programs possibly generated from a mature architecture are free from given classes of faults or that they work correctly for all possible inputs. These steps will bring us closer to the point where product-quality software can be economically produced using the same engineering automation technology that enables evolutionary prototyping and helps analysts home in on good requirements models. Our vision is to eliminate the current conflict between rapid development and high software quality.

Our ultimate research goal is to create conceptual models and software tools that allow automatic generation of variations on a software system with human consideration of only the highest-level decisions that must change between one version and the next. Realization of this goal will lead to more flexible software systems and should make prototyping and exploratory design more effective.

References

[1] S. Badr, Luqi, Automation Support for Concurrent Software Engineering. *Proc. of the 6th International Conference Software Engineering and Knowledge Engineering*, Jurmala, Latvia, June 20-23, 1994, 46–53.

[2] F. Bauer et al., *The Munich Project CIP. Volume II: The Program Change System CIP-S*, Lecture Notes in Computer Science 292, Springer 1987.

[3] V. Berzins, On Merging Software Enhancements *Acta Informatica*, Vol. 23 No. 6, Nov 1986, pp. 607–619.

[4] V. Berzins, Luqi, An Introduction to the Specification Language Spec, *IEEE Software*, Vol. 7 No. 2, Mar 1990, pp. 74–84.

[5] V. Berzins, Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada*, Addison-Wesley Publishing Company, 1991, ISBN 0–201–08004–4.

[6] V. Berzins, Software Merge: Models and Methods, *Journal of Systems Integration*, Vol. 1, No. 2, pp. 121–141 Aug 1991.

[7] V. Berzins, Luqi, M. Shing, Real-Time Scheduling for a Prototyping Language, *Journal of Systems Integration*, Vol. 6, No. 1-2, pp. 41–72, 1996.

[8] V. Berzins, O. Ibrahim, Luqi, A Requirements Evolution Model for Computer Aided Prototyping *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, Madrid, Spain, June 17-20, 1997, pp. 38–47.

[9] D. Dampier, Luqi, V. Berzins, Automated Merging of Software Prototypes. *Journal of Systems Integration*, Vol. 4, No. 1, February, 1994, pp. 33–49.

[10] V. Berzins, Software Merge: Semantics of Combining Changes to Programs. *ACM TOPLAS*, Vol. 16, No. 6, Nov. 1994, 1875–1903.

[11] V. Berzins, *Software Merging and Slicing*, IEEE Computer Society Press Tutorial, 1995, ISBN 0-8186-6792-3.

[12] V. Berzins, D. Dampier, Software Merge: Combining Changes to Decompositions. *Journal of Systems Integration*, special issue on CAPS (Vol. 6, No. 1–2, March 1996), pp. 135-150.

[13] M. Dowson, The ARIANE 5 Software Failure, *ACM Software Engineering Notes*, Vol. 22 No. 2, March 1997, p. 84.

[14] J. Evans, Software Project Scheduling Tool, MS Thesis, Computer Science, Naval Postgraduate School, Sep. 1997.

[15] M. Feather, A System for Assisting Program Change. *ACM Transactions on Programming Languages and Systems*, Vol. 4 No. 1, Jan 1982, pp. 1–20.

[16] M. Feather, A Survey and Classification of some Program Change Approaches and Techniques, in *Program Specification and Change (Proceedings of the IFIP TC2/WG 2.1 Working Conference)*, L.G.L.T. Meertens, Ed., North-Holland, 1987, pp. 165–195.

[17] M. Feather, Constructing Specifications by Combining Parallel Elaborations, *IEEE Transactions on Software Engineering*, Vol. 15 No. 2, Feb 1989, pp. 198–208.

[18] S. Fickas, Automating the Transformational Development of Software, *IEEE Transactions on Software Engineering*, Vol. 11 No. 11, Nov 1985, pp. 1268–1277.

[19] W. Gibbs, Software's Chronic Crisis, *Scientific American*, SEP 1994, pp. 86–94.

[20] W. Johnson, M. Feather, Building an Evolution Change Library, *12th International Conference on Software Engineering*, 1990, pp. 238–248.

[21] E. Kant, On the Efficient Synthesis of Efficient Programs, *Artificial Intelligence*, Vol. 20 No. 3, May 1983, pp. 253–36. Also appears in [35], pp. 157–183.

[22] B. Kraemer, Luqi, V. Berzins, Compositional Semantics of a Real-Time Prototyping Language *IEEE Transactions on Software Engineering*, Vol. 19, No. 5, pp. 453–477, May 1993.

[23] D. Lange, Hypermedia Analysis and Navigation of Domains, MS Thesis, Computer Science, Naval Postgraduate School, Sep. 1997.

[24] Luqi, M. Ketabchi, A Computer Aided Prototyping System, *IEEE Software*, Vol. 5 No. 2, Mar 1988, pp. 66–72.

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

[26] Luqi, Handling Timing Constraints in Rapid Prototyping *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, IEEE Computer Society, Jan. 1989, pp. 417–424.

[27] Luqi, Software Evolution via Rapid Prototyping, *IEEE Computer*, Vol. 22, No. 5, May 1989, pp. 13–25.

[28] Luqi, A Graph Model for Software Evolution, *IEEE Transactions on Software Engineering*, Vol. 16. No. 8, pp. 917–927, Aug. 1990.

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

[30] Luqi, Real-Time Constraints in a Rapid Prototyping Language, *Journal of Computer Languages*, Vol. 18, No. 2, pp. 77–103, Spring 1993.

[31] Luqi, Specifications in Software Prototyping, *Proc. SEKE 96*, Lake Tahoe, NV, June 10-12, 1996, pp. 189-197.

[32] Luqi, Scheduling Real-Time Software Prototypes, *Proceedings of the 2nd International Symposium on Operations Research and its Applications*, Guilin, China, December 11-13, 1996, pp. 614-623.

[33] Luqi, J. Goguen, Formal Methods: Promises and Problems, *IEEE Software*, Vol. 14, No. 1, Jan. 1997, pp. 73-85.

[34] R. Lutz, Analyzing Software Requirements: Errors in Safety-Critical Embedded Systems, TR 92-27, Iowa State University, AUG 1992.

[35] C. Rich, R. Waters, Eds., *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[36] D. Rusin, Luqi, M. Scanlon, SIDS Wireless Acoustic Monitor (SWAM), *Proc. 21st Int. Conf. on Lung Sounds*, Chester, England, International Lung Sounds Association, Sep. 4-6, 1996.

[37] D. Smith, G. Kotik, S. Westfold, Research on Knowledge-Based Software Environments at Kestrel Institute, *IEEE Transactions on Software Engineering*, Vol. 11 No. 11, Nov 1985, pp. 1278-1295.

[38] Chaos, Technical Report, The Standish Group, Dennis, MA, 1995, http://www.standishgroup.com/chaos.html.

[39] W. Swartout, R. Balzer, On the Inevitable intertwining of Specification and implementation, *Communication of the ACM*, Vol. 25 No. 7, July 1982, pp. 438-440. Also appears in *Software Specification techniques*, N. Gehani, A.D. McGettrick, Eds., 1986, pp. 41-45.