1999

Software Evolution Process via a Relational Hypergraph Model

Luqi; Harn, M.; Berzins, V.; Mori, A.

Software Evolution Process via a Relational Hypergraph Model, M. Harn, V. Berzins, A. Mori, Proc. IEEE/IEEJ/JSAI Int. Conf. on Intelligent Transportation Systems, Tokyo, Japan, pp. 599-604.

http://hdl.handle.net/10945/42344

Software Evolution Process via a Relational Hypergraph Model*

M. Harn, V. Berzins, Luqi
Computer Science Department, Naval Postgraduate School, Monterey, CA 93943 USA
{harn, berzins, luqi, mori}@cs.nps.navy.mil

A. Mori
Japan Advanced Institute of Science and Technology
1-1 Asahidai Tatsunokuchi Nomi Ishikawa, 923-1292, JAPAN
amori@jaist.ac.jp

Abstract

The purpose of this paper is to formalize the software evolution process via a relational hypergraph model with primary-input-driven and secondary-input-driven dependency approaches. Software evolution processes are modeled by a multidimensional architecture containing successive software evolution steps and related software evolution components. We analyze a domain-specific software development architecture and give a standard software evolution process in developing a prototype system as well as a software production system. The relational hypergraph model is applied well in several real-time prototyping systems such as Command, Control, Communication and Intelligence (C3I) systems; army transportation systems; rail road signal control systems; and future traffic light systems.

1. Introduction

Our emphasis is a software evolution process based on a relational hypergraph model (RH model). Software evolution is currently not well understood. It is very difficult to completely formalize the software evolution process for a large scale and complex software system, especially if one tries to include social, political and cultural factors [16]. Our intention is to propose this model for automating the software evolution process.

Due to rapid requirement changes in the enterprise environment, a delivered software system seldom includes the new requirement changes requested by the customers [9, 10, 16, 17, 19]. Computer-Aided Software Prototyping System (CAPS) and a hypergraph model have successfully resolved this software evolution requirements instability problem [16]. CAPS is an integrated tool that can be used to rapidly design real-time applications utilizing its prototype system description language (PSDL) editor, reusable software database, program generator, real-time scheduler, and so on [12].

*This research was supported in part by the U. S. Army Research Lab under grant number 7DNAVYR010 and in part by the U. S. Army Research Office under contract/grant number 38690.

There is no an efficient and standard software evolution process to support software system development [8]. This paper proposes that the RH model with primary-input-driven and secondary-input-driven dependency approaches can easily and clearly describe the software evolution process. We analyze a domain-specific software development architecture and give a standard software evolution process in developing a prototype system as well as a software production system. This standard process can be used to a real-time software development environment such as Command, Control, Communication and Intelligence (C3I) systems [15]; army transportation systems; rail road signal control systems; and future traffic light systems.

The RH model is a formal model of the structure of software evolution. The previous studies about software evolution are incomplete. For example: (1) the object-oriented approach to software evolution can be used to describe relationships between classes, but it is still very rigid and hard to evolve because of redundant information about class relationships [10, 20]; (2) the evolution control system (ECS) provides generic automated assistance for the software evolution process [1, 2], but the details of the software evolution process are unknown; (3) the extended graph model [14] is a good way to represent software requirements issues, but efficiently recording and tracing software evolution activities within the software evolution process is unclear; (4) the hypergraph model was introduced to formalize software evolution [16], but it is incomplete to define and classify software evolution objects with their multidimensional dependencies. Therefore, we abstracted and integrated many ideas, such as a graph data model [14], a hypergraph model [6, 7, 16], an IBIS (issue-based information system) model [4, 8], a prototyping method [5, 12, 13], a formal method [15], a reuse architecture [11, 18, 21], and so on, to produce the RH model.

2. Hypergraphs preliminary

Our relational hypergraph model is a refinement and reformulation of the hypergraph model introduced in [6, 7, 16]. The hypergraph model represents the evolution history and future plans for software development as a *hypergraph.*

Hypergraphs generalize the usual notion of a directed graph by allowing *hyperedges*, which may have multiple output nodes and multiple input nodes.

Definition 1. (Hypergraph) A *(directed) hypergraph* is a tuple $H = (N, E, I, O)$ where

1. N is a set of *nodes*,
2. E is a set of *hyperedges* (briefly called *edges*),
3. I: $E \rightarrow 2^N$ is a function giving the set of *inputs* of each hyperedge, and
4. O: $E \rightarrow 2^N$ is a function giving the set of *outputs* of each hyperedge.

This definition describes a bare structure of a hypergraph. The traceability of the software evolution can be presented via the path of a hypergraph.

A *path* in the hypergraph represents an evolution history whose components, including nodes and hyperedges, can be traced.

In the evolutionary hypergraph model [16], the software evolution components and steps have been identified by nodes and edges respectively. The attributes of the components and the steps must be recorded and labeled.

Definition 2. (Evolutionary Hypergraph) An *evolutionary hypergraph* is a labeled, directed, and acyclic hypergraph $H = (N, E, I, O)$ together with label functions $L_N : N \rightarrow C$ and $L_E : E \rightarrow A$ such that the following assumptions are satisfied:

1. The elements of N represent unique identifiers for software evolution components;
2. The elements of E represent unique identifiers for software evolution steps;
3. The functions I and O give the inputs and outputs of each software evolution step, such that $O(e) \cap O(e') \neq \varnothing$ implies $e = e'$;
4. The function L_N labels each node with *component attributes* from the set C, including the corresponding version of the software evolution component;
5. The function L_E labels each edge with *step attributes* from the set A, including the current status of the software evolution step, such that $A = \{s, d\} \cdot A'$ (that is, each element of A has the form (s, a') or (d, a'), where $a' \in A'$). An edge labeled *"s"* is called a *step* and one labeled *"d"* is called a *decomposition.*

According to this definition, both components and steps can be refined into finer components and steps. In particular, the minimal hypergraph whose edge set has only one edge can also be refined into a finer hypergraph [16].

3. RH model

The RH model describes two phenomena of software evolution: primary-input-driven paths and secondary-input-driven paths [6, 7]. The input part to each hyperedge in a path could be a set of multiple input nodes containing many kinds of software evolution components. If there exist an input node and an output node to an evolutionary hyperedge that are different versions of the same component then the path from the input node via the hyperedge to the output node is called a *primary-input-driven path* and the relationship between the input node and the step is called *primary_input.* If there exist an input node and an output node to an evolutionary hyperedge that are different types of components then the path from the input node via the hyperedge to the output node is called a *secondary-input-driven path* and the relationship between the input node and the step is called *secondary_input.* The relational hypergraph can be defined by these concepts.

Definition 3. (Relational Hypergraph) An evolutionary hypergraph $H = (N, E, I, O)$ is called a *relational hypergraph* if and only if for every hyperedge e in H and every input node n in $I(e)$, the relationship between n and e is *primary_input* or *secondary_input.*

Because input nodes to a step come from different entrances, a step in a relational hypergraph can be represented by an arrow with multiple tails or combinational arrows.

The representation of an arrow with multiple tails contains different input entrances that are primary inputs or secondary inputs.

The representation of combinational arrows points out a common output node and different paths (primary-input driven path or secondary-input-driven path) of a step. In order to describe traceability of a software evolution process, we use combinational arrows to illustrate paths of a step.

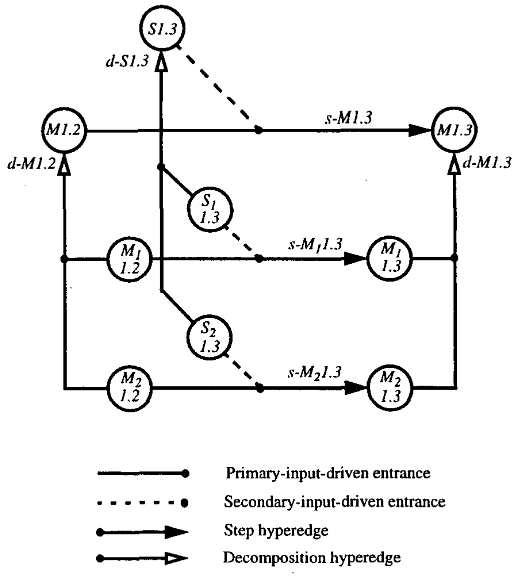

Figure 1: A relational hypergraph

Figure 1 shows a relational hypergraph including a top-level step *s-M1.3* (with input nodes *M1.2, S1.3,* and an output node *M1.3*), and its refined steps *s-*M_1*1.3* (with input nodes M_1*1.2*, S_1*1.3,* and an output node M_1*1.3*) and *s-*M_2*1.3* (with input nodes M_2*1.2*, S_2*1.3,* and an output node M_2*1.3*). The node *M1.2* is a primary input node since the input node *M1.2* and the output node *M1.3* to the hyperedge *s-M1.3* are different versions of the same component *M*. The node *S1.3* is a secondary input node since the input node *S1.3* (representing a specification component) and the output node *M1.3* (representing a module component) to the hyperedge *s-M1.3* are different components. The top-level nodes *M1.2, S1.3,* and *M1.3* can be decomposed into their refined nodes via decomposition hyperedges *d-M1.2, d-S1.3*, and *d-M1.3*, respectively.

Figure 2 represents paths of the same hypergraph as the Figure 1 with two types of combinational arrows *pp* and *sp*. The dark arrow *pp(s-M1.3)* is called the primary-input-driven path of the step *s-M1.3* and the dash arrow *sp(s-M1.3)* is called the secondary-input-driven path of the step *s-M1.3.* It makes a hypergraph easy to read especially for tracing a huge number of steps and their related nodes. Therefore, relationships among nodes, subnodes, a top-level step and refined steps in a relational hypergraph can be established by these two types of representation.

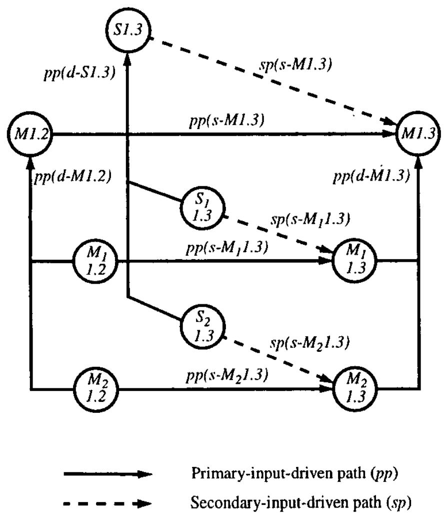

Figure 2: Paths in a relational hypergraph

A *relational hypergraph net* has two relational hypergraphs: a top-level relational hypergraph and a refined relational hypergraph. A complicated relational hypergraph can be transferred into a relational hypergraph net for simplifying the hypergraph structure. The top-level and the refined relational hypergraphs are respectively connected by top-level and refined SPIDERs (Step Processed in Different Entrance Relationships) that are formed by a specified step together with its input components and unique output component [6, 7].

4. Software evolution process

The model of software evolution process is based on the RH model and the IBIS model [4]. The RH model provides a primary and secondary input driven mechanism to drive the software evolution process via a sequence of activities. The IBIS model relates design rationale to the artifacts created during the systems development process [16]. Therefore, the model of software evolution process can describe a secondary input driven mechanism in software evolution process well and provide another aspect from the original software evolution description based on the primary input driven mechanism [2].

A software component has to be modified from an old version to a new version due to social, political, or cultural factors. The software evolution process of a component driven by primary input can be shown as Figure 3.

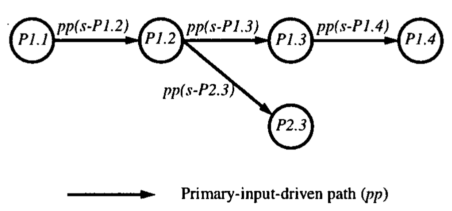

Figure 3: A software evolution process driven by primary input

The objects affected by the software evolution process are called *software evolution objects.* According to the Schematic Model of the Analysis Process modified from the IBIS model in [3, 8], we classify *software evolution objects* (briefly called *objects*) into *software evolution steps* (briefly called *steps*) and *software evolution components* (briefly called *components*) in the software evolution process. Both kinds of objects can be hierarchically refined into finer grain objects. The leaf nodes of the refinement hierarchy are atomic objects.

We have identified eight types of top-level *steps* in the software evolution process: *software prototype demo step, issue analysis step, requirement analysis step, specification design step, module implementation step, program integration step, software product demo step, and software product implementation step.* Each top-level step can be refined into more specific software evolution activities at many levels.

In the software evolution process, there is a top-level component between two adjacent top-level steps that is an input of one and an output of the other. Therefore, we also have identified seven different types of top-level *components* in the software evolution process: *criticisms, issues, requirements, specifications, modules, programs, and optimizations.* Each top-level component can be decomposed into a set of atomic components, either directly or indirectly.

Definition 4. (Software Prototype Evolution Process) Let $H = (N, E, I, O)$ be a relational hypergraph. Let t be the number of evolution times and path p be a sequence $p_1 \dots p_t$ of paths driven by secondary input, where for each $m = 1, \dots, t$ path p_m from a node n of an old version of software prototype component to a node n' of a new version of software prototype component is a sequence $e_1 \dots e_6$ of hyperedges and a sequence $n_0 \dots n_6$ of nodes such that $n_{i-1} \in I(e_i)$ and $n_i \in O(e_i)$ for $i = 1, \dots, 6$, where $n = n_0$ and $n' = n_6$. We say hypergraph H is a *software prototype evolution process* if and only if there exists a path p such that the following assumptions are satisfied:

1. Hyperedges $e_1 \dots e_6$ represent the following kinds of steps: software prototype or product demo, issue analysis, requirement analysis, specification design, module implementation, and program integration, respectively.
2. Nodes $n_0 \dots n_6$ represent the following kinds of components: old version of programs, criticisms, issues, requirements, specifications, modules, and new version of programs, respectively.
3. Let e_i^m be a hyperedge e_i, where $i = 1, \dots, 6$, in the path p of the mth evolution. For $m = 1$ and each $i = 1, \dots, 6$, there exist a hyperedge e_i^m, nodes n_{i-1}^m, n_i^m, and n_i^{m-1}, where $n_{i-1}^m \in I(e_i^m)$ and $n_i^m \in O(e_i^m)$, such that $n_0^m = n_6^{m-1} \in I(e_6^m)$. For each $m = 2, \dots, t$ and $i = 1, \dots, 6$, there exist a hyperedge e_i^m, nodes n_{i-1}^m, n_i^m, and n_i^{m-1}, where $n_{i-1}^m \in I(e_i^m)$ and $n_i^m \in O(e_i^m)$, such that $n_i^{m-1} \in I(e_i^m)$.

Definition 5. (Software Product Generation Process) Let $H = (N, E, I, O)$ be a relational hypergraph. Let t be the number of evolution times and path q be a sequence $q_1 \dots q_t$ of paths driven by secondary input, where for each $m = 1, \dots, t$ path q_m from a node n of a firm software prototype component to a node n' of a software product component is a sequence $e_1 \dots e_2$ of edges and a sequence $n_0 \dots n_2$ of nodes such that $n_{i-1} \in I(e_i)$ and $n_i \in O(e_i)$ for $i = 1, 2$, where $n = n_0$ and $n' = n_2$. We say hypergraph H is a *software product generation process* if and only if there exists a path q such that the following assumptions are satisfied:

1. Hyperedges e_1 and e_2 represent a software prototype or product demo step and a software product implementation step, respectively.
2. Nodes $n_0 \dots n_2$ represent the following kinds of components: new version of software prototype programs or old version of software product programs, optimizations, and new version of software product programs, respectively.
3. Let e_i^m be a hyperedge e_i, where $i = 1, 2$, in the path q of the mth evolution. For $m = 1$ and each $i = 1, 2$, there exist a hyperedge e_i^m, nodes n_{i-1}^m, n_i^m, and n_i^{m-1}, where $n_{i-1}^m \in I(e_i^m)$ and $n_i^m \in O(e_i^m)$, such that $n_0^m = n_2^{m-1} \in I(e_2^m)$. For each $m = 2, \dots, t$ and $i = 1, 2$, there exist a hyperedge e_i^m, nodes n_{i-1}^m, n_i^m, and n_i^{m-1}, where $n_{i-1}^m \in I(e_i^m)$ and $n_i^m \in O(e_i^m)$, such that $n_i^{m-1} \in I(e_i^m)$.

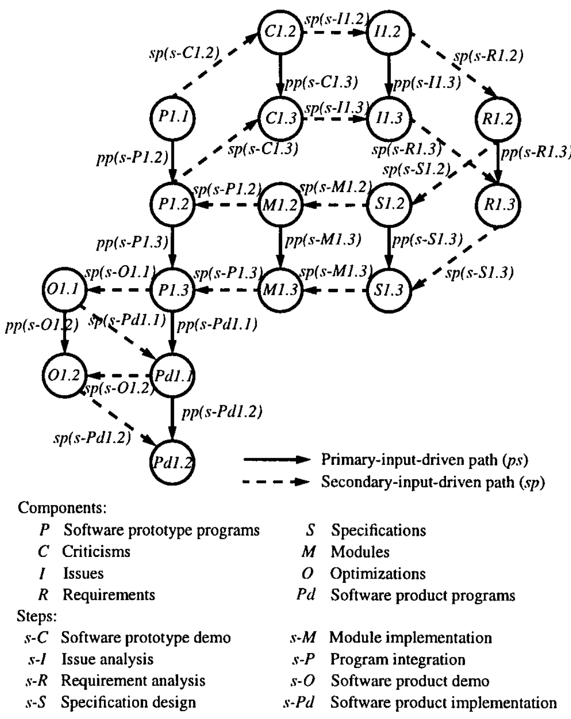

Figure 4: Software evolution process driven by primary input and secondary input

Definition 6. (Software Evolution Process) Let $H = (N, E, I, O)$ be a relational hypergraph. Let path p^k be a sequence $p_1 \dots p_t$ of paths driven by secondary input in software prototype evolution process and path q^k be a sequence $q_1 \dots q_t$ of paths driven by secondary input in software product generation process, where $k = 1, \dots, n$. We say hypergraph H is a *software evolution process* if and only if the following assumptions are satisfied:

1. The structure of hypergraph H is combined by software prototype evolution process and software product generation process.
2. There is a path P such that $P = p^1 q^1 \dots p^n q^n$.

Figure 4 shows an example of software evolution process driven by primary input and secondary input. The first process from node $P1.1$ to node $P1.3$ is a software prototype evolution process (evolution times $t = 2$) and the second process from node $P1.3$ to node $Pd1.2$ is a software product generation process (evolution times $t = 2$).

In the first process, node $P1.2$ is evolved in the first software prototype evolution from node $P1.1$ and node $M1.2$ via step $s\text{-}P1.2$, where node $M1.2$ is the result of a series steps, *s-*

C1.2, s-I1.2, s-R1.2, s-S1.2, s-M1.2. In the second process, node *P1.3* is evolved in the second software prototype evolution from node *P1.2* and node *M1.3* via step *s-P1.3,* where node *M1.3* is the result of a series of steps, *s-C1.3, s-I1.3, s-R1.3, s-S1.3, s-M1.3*. In the third process, the node *Pd1.1* is evolved in the first software product evolution from node *P1.3* and node *O1.1* via step *s-Pd1.1,* where node *O1.1* is the result of a steps *s-O1.1*. In the fourth process, the node *Pd1.2* is evolved in the second software product evolution from node *Pd1.1* and node *O1.2* via step *s-Pd1.2,* where node *O1.2* is the result of a steps *s-O1.2*.

5. An example: TL2000 traffic light system

TL2000 is a cutting-edge traffic light system for the future that will help to improve the traffic problems that many city dwellers are facing. TL2000 is tomorrow's solution for today's congested traffic problems. With TL2000, implemented in the object-oriented Ada'95, we can be assured that traffic congestions will be our least concern as we are heading into the next millennium. TL2000 will not only succeed in meeting its objective of reducing traffic problems; but also will be a model for future software development in that it incorporated a specification language PSDL, the technologically break-through in software prototyping, into the software development process.

We obtain the requirements of TL2000 from customers and design prototypes by RH model and CAPS via the software evolution process. The primitive requirements of TL2000 are as follows:

R1.1: The traffic light system must be able to handle the flow of *"straight"* traffic from all four directions.

R_1*1.1*: The system must signal *"Green"* for one direction and *"Red"* for the other.

R_2*1.1*: Set light to *"Red"* for all *"straight"* traffic and *"Green"* to left-turn traffic.

R1.2: The system must detect and respond accordingly to left-turn traffic.

R_1*1.2*: The system must detect for left-turn traffic.

R_2*1.2*: The system must register accordingly whether or not there is left-turn traffic.

The following new requirements are obtained via a series of software evolution processes: such as software prototype demo step, issue analysis step, and requirement analysis step.

- *Maximum waiting period:* The maximum waiting time for any automobiles at the traffic light must not exceed ninety seconds.
- *Power outage tolerance:* After the main power supply is out and the battery backup kicks in; the system will gracefully shutdown and flashing the red signal at all directions.
- *Straight traffic time:* Once the traffic light has turned to green (go) for any straight traffic, it must remain green for a full 60 seconds.
- *Left-turn lanes:* Cars coming from the same direction that is waiting in left turn lanes shall have priority over those going straight. When traffic light changes before straight traffic can go, left-turn traffic must have a full thirty seconds to maneuver.
- *Shutdown:* The system must gracefully shutdown within three seconds after the turning of the switch.
- *Start-up:* The system must start-up and fully operational within 3 seconds after the turning of the switch. Any deviation from this is deemed as system failure.

After requirements transfer into specifications by PSDL graphic editor (shown as Figure 5), CAPS generates related Ada code for developers. Due to rapid software prototyping, the difference between customers and developers is reduced.

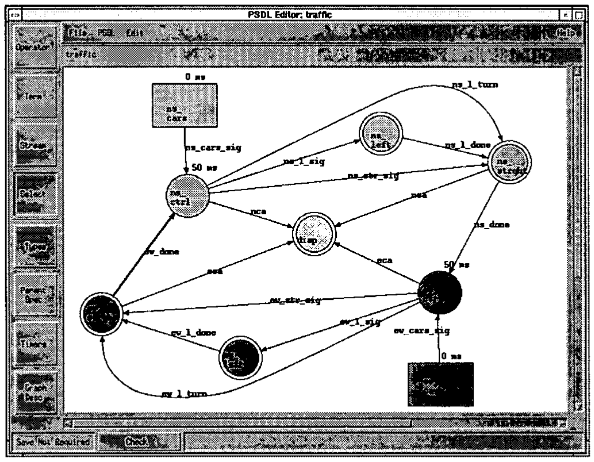

Figure 5: A TL2000 traffic light system designed by PSDL editor

In the software evolution process of TL2000, there are two types of component files: a text file and a software code file. The components of criticisms, issues, requirements and optimizations are described as text and stored in a text component base. The components of specifications, modules, and programs are described by software code and stored in a software component base.

Developed by CAPS, TL2000 is a robust, well-designed, and well-written application that can be integrated into the existing traffic light systems and performed smoothly.

6. Conclusion

The RH model is a formal model for software evolution which can help us develop tools to manage both the activities in a software development project and the products that those activities produce. This model incorporates some features of the previous CAPS models into a more abstract mathematical structure.

This article formalizes a portion of software evolution process that is typical of prototyping. This structure is the basis for developing process dependent inference rules for determining dependencies [7] and performing planning and scheduling tasks.

Acknowledgments

We thank T. Tran and B. Tran for their assistance with the TL2000.

References

[1] S. Badr, A Model and Algorithms for a Software Evolution Control System, *Ph.D. Dissertation*, Computer Science Department, Naval Postgraduate School, Monterey, CA, 1993.

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

[3] V. Berzins, O. Ibrahim, and 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.

[4] J. Conklin and M. Begeman, gIBIS: A Hypertext Tool for Exploratory Policy Discussion, *ACM Transactions on Office Information System*, Vol. 6, October 1988, pp. 303-331.

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

[6] M. Harn, V. Berzins, and Luqi, Software Evolution via Reusable Architecture, *Proceedings of 1999 IEEE Conference and Workshop on Engineering of Computer-Based Systems,* Nashville, Tennessee, March 7-12, 1999, pp. 11-17.

[7] M. Harn, V. Berzins, and Luqi, A Dependency Computing Model for Software Evolution, *Proceeding of the Eleventh International Conference on Software Engineering and Knowledge Engineering,* Kaiserslautern, Germany, June 17-19, 1999, pp. 278-282.

[8] O. M. Ibrahim, A Model and Decision Support Mechanism for Software Requirements Engineering, *Ph.D. Dissertation*, Computer Science Department, Naval Postgraduate School, Monterey, CA, 1996.

[9] T. Leonard, V. Berzins, Luqi, and M. J. Holden, Gathering Requirements from Remote Users, *Proceeding of the ninth International Conference on Tools with Artificial Intelligence*, Newport Beach, California, November 3-8, 1997, pp. 462-471.

[10] K. J. Lieberherr and C. Xiao, Object-Oriented Software Evolution, *IEEE Trans. on Software Engineering*, Vol. 19, No. 4, April 1993, pp. 313-343.

[11] L. Liu, R. Zicari, W. Hursh, and K. Lieberherr, The Role of Polymorphic Reuse Mechanisms in Schema Evolution in an Object-Oriented Database, *IEEE transactions on Knowledge and Data Engineering,* Vol. 9, No. 1, January-February 1997, pp. 50- 67.

[12] Luqi and M. Ketabchi, A Computer-Aided Prototyping System, *IEEE Software*, March 1988, pp. 66-72.

[13] Luqi, Software Evolution Through Rapid Prototyping, *IEEE Computer*, May 1989, pp. 13-25.

[14] Luqi, A Graph Model for Software Evolution, *IEEE Trans. on Software Engineering*, Vol. 16, No. 8, August 1990, pp. 917-927.

[15] Luqi, Computer-Aided Prototyping for a Command-And-Control System Using CAPS, *IEEE Software*, January 1992, pp. 56-67.

[16] Luqi and J. A. Goguen, Formal Methods Promises and Problems, *IEEE Software*, January 1997, pp. 73-85.

[17] N. Madhavji, Environment Evolution: The Prism Model of Changes, *IEEE Trans. on Software Engineering*, Vol. 18, No. 5, May 1992, pp. 380-392.

[18] V. Rajlich, J. Silva, Evolution and Reuse of Orthogonal Architecture, *IEEE Trans. on Software Engineering*, Vol. 22, No. 2, February 1996, pp. 153-157.

[19] B. Ramesh and Luqi, An intelligent Assistant for Requirements Validation for Embedded Systems, *Journal of Systems Integration*, Vol. 5, No. 2, 1995, pp. 157-177.

[20] L. Seiter, J. Palsberg, K. Leiberherr, Evolution of Object Behavior Using Context Relations, *IEEE Trans. on Software Engineering*, Vol. 24, No. 1, January 1998, pp. 79-92.

[21] R. Steigerwald, Luqi, and J. McDowell, CASE Tool for Reusable Software Component Storage and Retrieval in Rapid Prototyping, *Information and Software Technology*, England, Vol. 38, No. 9, November 1991, pp. 698-706.