Using Non-Functional Requirements to Systematically Select Among Alternatives in Architectural Design*

Lawrence Chung

Computer Science Program
The University of Texas at Dallas

Brian A. Nixon & Eric Yu

Department of Computer Science
University of Toronto

Abstract

Non-functional requirements, such as modifiability, performance, reusability, comprehensibility and security, are often crucial to a software system. As such, these non-functional requirements (or NFRs) should be addressed as early as possible in a software lifecycle and properly reflected in a software architecture before committing to a detailed design. The purpose of this paper is to discuss how the treatment of NFRs as goals (which may be synergistic or conflicting) serves to systematically guide selection among architectural design alternatives. During the architectural design process, goals are decomposed, design alternatives are analysed with respect to their tradeoffs, design decisions are made rationalised, and goal achievement is evaluated. This process can be supported by by a body of organised knowledge. This paper outlines an approach by which such knowledge can be organized. This approach is illustrated by a preliminary study of architectural design for a KWIC (Key Word in Context) system.

Keywords: non-functional requirements, selection among design alternatives.

1 Introduction

Non-functional requirements, such as modifiability, performance, reusability comprehensibility and security, are often crucial to a software system. As such, these non-functional requirements (or NFRs) should be addressed as early as possible in the software lifecycle and properly built into a software architecture before a detailed design proceeds on an otherwise undesirable path.

As pointed out by Garlan and Perry [18], architectural design has traditionally been largely informal and *ad hoc.* The manifested symptoms include difficulties in communication, analysis, and comparison of architectural designs and principles. A more disciplined approach to architectural design is needed to improve our ability to understand the interacting high-level system constraints and the rationale behind architectural choices, to reuse architectural knowledge concerning NFRs, to make the system more evolvable, and to analyse the design with respect to NFR-related concerns.

Suppose we are developing an architecture for a KWIC (Keyword in Context) system. We want to meet non-functional requirements for modifiability, performance, etc. But if we optimize performance too early, we may well hinder future modifiablity. How do we keep track of requirements and their interactions, while selecting among design alternatives to meet the requirements?

The purpose of this paper is to discuss how the treatment of NFRs as potentially synergistic or conflicting goals serves to systematically guide selection among architectural design alternatives. This treatment uses a framework,

*Authors' addresses: L. Chung, Computer Science Program, The University of Texas at Dallas, P. O. Box 830688, Richardson, TX 75083–0688, U.S.A., chung@utdallas.edu; B. Nixon and E. Yu, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A4, {nixon,eric}@ai.toronto.edu.

the NFR-Framework (Chung [7], Mylopoulos, Chung, and Nixon [30]), in which NFRs are represented as goals to be achieved *during the process* of software development.

During the design process, goals are decomposed, design alternatives are analysed with respect to their tradeoffs, design decisions are rationalised, goal achievement is evaluated, and a selection is made. In this approach, NFR-related knowledge is codified into *methods* and *correlation rules.* Methods are used to facilitate decomposition and achievement of goals, and argumentation of design decisions. Correlation rules are used to help analyse tradeoffs among design alternatives, to guide selection among alternatives, to help detect goal conflicts, synergy, and omissions, and to visually aid the representation of goal interactions. This way, a body of codified NFR-related knowledge offers the needed subject matter and vocabulary, and is made available and reusable throughout the process.

This process is intended not only as a means for supporting the design of software architecture but also as a resultant history record for later review, justification and evolution.

Our proposal has emphasis in representing and using NFRs during the design of software architectures. Boehm [3], Perry and Wolf [38], and Kazman, Bass, Abowd, and Webb [24] have argued convincingly for the importance of addressing non-functional concerns in software architectures. Our explicit representation of NFRs was motivated by Boehm's [2] insightful observation that when developers are made aware of quality concerns, that by itself helps improve the overall software quality.

This approach is illustrated by a preliminary study of architectural design for a KWIC system. This study examines primarily the requirements level, and is intended as a basis for systematic selection among architectural design alternatives.

Section 2 describes the process-oriented NFR-Framework and its treatment of NFRs as (potentially) conflicting or synergistic goals in the context of software architectural design. Section 3 illustrates the use of the NFR-Framework by a study of architectural design for a KWIC system. Section 4 discuss related work. The contribution of the current paper, along with future directions, is summarised in Section 5.

2 Goal-driven, Process-Oriented Architectural Design

To remedy the problems inherent in *ad hoc* architectural design, a more disciplined approach is needed to improving our ability to understand the high level system constraints and the rationale behind architectural choices, to reuse architectural design knowledge, to make the system more evolvable, to analyse the design with respect to design criteria.

For this kind of a more disciplined approach, we adopt the NFR-Framework ([7] [30]), a framework for dealing with NFRs *during the process* of software development, to offer:

1. explicit representation of non-functional requirements: NFRs are represented as (potentially) conflicting or synergistic *goals* to be addressed during the process of architectural design, and used to rationalise the overall architectural design and selection process;

2. systematic use of architectural design knowledge: *Methods* are used to organize NFR-related knowledge and experience and made available during the process;

3. management of tradeoffs among architectural design alternatives: *Correlation rules* are used to organise knowledge and experience about design tradeoffs, arising from goal conflict and synergy, and made available during the process;

4. evaluation of goal achievement with a particular choice of architectural design: Throughout the process, the *evaluation procedure* propagates, via labels, the effect of each design decision in order to help select among architectural design alternatives.

2.1 Treating Non-Functional Requirements as Goals

In our process-oriented approach, non-functional requirements, such as "very modifiable system" and "good system performance", are explicitly represented as goals to be addressed and achieved during the process of architectural design. Each goal (e.g., `Modifiability [system; critical]`) is associated with a sort or type (e.g., `Modifiability`), a parameter list (e.g., `system`), and importance (e.g., `critical`).

One fundamental premise of our approach is that NFR goals have the property of potentially interacting with each other, in conflict or in synergy. This property is used to systematically guide selection among architectural design alternatives, and to rationalise the overall architectural design process.

In the design process, goals and goal relationships (*links*) can also represent design alternatives, decisions, and rationale; they are recorded and structured in a *goal graph* (See Figure 5). Link types, such as *AND* and *OR*, are similar in spirit to Nilsson's AND/OR trees [32]. It incorporates Simon's notion of goal *satisficing* [40][1] which reflects partial contributions by a design decision towards (or against) a particular goal. Consequently a particular architectural design is expected to satisfy NFRs within acceptable limits, rather than absolutely.

This process is intended not only as a means for supporting design but also as a resultant history record for later review, justification and evolution.

2.2 Methods

Architectural design knowledge and experience about specific NFRs can be organized into methods and made available to the software architect through systematic search. An important first step in using the NFR framework is to obtain and organise knowledge of a specific NFR, e.g., performance, from both academic research and industrial experience. This paper illustrates the use of the framework, while presenting just an initial classification of architectural knowledge.

In our approach, there are three types of methods: *decomposition methods* for refining or clarifying NFRs, *satisficing methods* for achieving NFRs, and *argumentation methods* for making design rationale in support or denial of design decisions.

Figure 1 shows both sort- and parameter- decomposition methods.

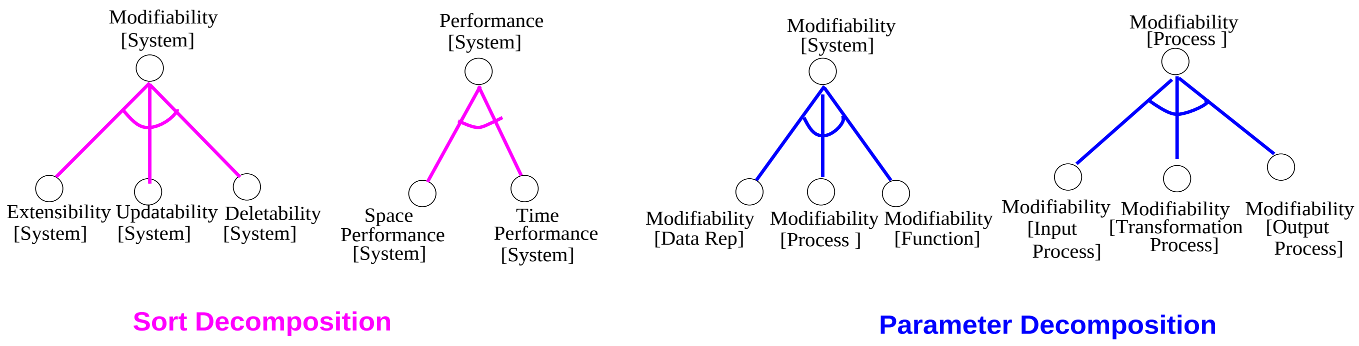

Figure 1: Sample decomposition methods.

The first sort decomposition method, which codifies our general knowledge about changes, states:

> *In order to achieve "modifiability", one needs to achieve "extensibility", "updatability", and "deletability".*

Such a method takes a parent goal, and produces offspring goals, shown as top-down decompositions in Figure 1. The final method is a parameter decomposition on process, including algorithms as used in [17]. The use of methods will be illustrated in Section 3.

Satisficing methods are used to codify knowledge about achieving NFR goals, and embedded in architectural designs when selected. For example, an *implicit function invocation regime* (Figure 4 (based on [17]), architecture 3) can be used to hide implementation details in order to make an architectural design more *extensible,* thus contributing to one of the goals in the above decomposition.

Argumentation methods are used to codify principles and guidelines for making design rationale for or against design decisions. An example of argumentation method is a codification of the *vital few, trivial many* prioritisation

[1] Simon actually uses the term to refer to decision methods that look for satisfactory solutions rather than optimal ones.

principle (the "80–20" rule) from industry [27, 23] and the system performance area [41]: given a set of goals, focus on meeting the few (20%) of them which are vital, rather than the remaining 80% of them. This method can be used in determining which goals are most important to satisfice and in selecting among alternatives to satisfice NFR goals, especially in the presence of time and manpower limitations.

2.3 Correlation Rules

Knowledge and experience about tradeoffs among architectural design alternatives can be codified into correlation rules and made available to the software architect through systematic search.

Once codified, correlation rules can be browsed by the software architect in selecting among architectural alternatives. Correlations can then be instantiated to record goal conflict and synergy, omissions and redundancies.

Table 1 shows a synopsis of correlation rules which are based on the presentation by Garlan and Shaw [17] which compared the extent to which alternative solutions address design considerations. In our adaptation of the notion of "satisficing", entries in Table 1 reflect weak or strong contributions by architectural design alternatives for (+) or against (−) NFRs. An empty entry indicates a lack of significant contribution. (See legend of Figure 2.) The table can be extended to incorporate more tradeoff knowledge or tailored to the needs of the intended application domain. An entry with +− means either a positive or negative contribution, and requires the software architect to consider the characteristics of the intended application domain.

	Shared Data	Abstract Data Type	Implicit Invocation	Pipe & Filter
Modifiability [Process]	− −	−	+	+
Modifiability [Data Rep]	− −	+	−	− −
Space Performance	++		−	− −*
Time Performance		−	− −	
Reusability	−	+	+−	+

− −*: if size of data in actual domain is huge

Table 1: Correlation Table, based on Garlan and Shaw.

A correlation table for the KWIC example then can be constructed to record interactions among NFR goals and architectural design alternatives, which are present in a goal graph, for a particular application domain under consideration.

2.4 Evaluation Procedure

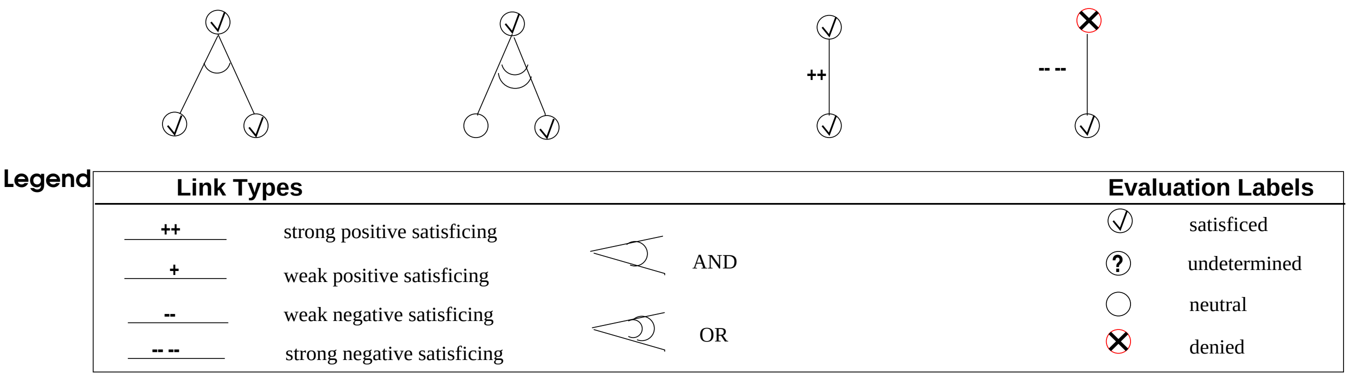

Figure 2: Some rules of the evaluation procedure.

Throughout the goal graph expansion process, the evaluation procedure propagates upwards, via labels of nodes in the graph, the effect of each design decision from offspring to parents, hence providing assessment of the degree of goal achievement.

The labels include *satisficed* ($\surd$), *denied* (**X**), and *undetermined* (?). In our approach, goal assessment is carried out by interaction between the evaluation procedure and the software architect who, by considering the characteristics of the intended application domain, makes the final decision on the label value to be propagated.

Figure 2 shows some of the rules used by the evaluation procedure. Notice that the propagation of a *satisficed* label along a *strong negative satisficing* link results in a *denied* label.

3 Architectural Design Process: Illustration

Our approach to supporting the process of architectural design is illustrated by the use of a KWIC system which was formulated by Parnas [37]:

> The KWIC (Key Word in Context) index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

The example is chosen since it is relatively well known, and used in several studies (by Parnas [37], Garlan, Kaiser, and Notkin [16], and Garlan and Shaw [17]) which provide a good illustration of tradeoffs among NFRs and design alternatives for the KWIC domain. However, it is used primarily as a "pedagogical" example in this paper, so that we can illustrate the use of the NFR Framework for selecting among alternatives for architectural design.

NFR Goals and Decomposition. For the purposes of illustration, assume that there is an initial set of NFR goals: "the system should be modifiable, with good performance, and reusable". The software architect might represent these by `Modifiability [System]`, `Performance [System]`, and `Reusability [System]`. They are shown at the top of Figure 3 which depicts the situation but with more progress (Some parameters are omitted from the figure). Details will be explained as we move along. This is an initial set of NFR goals that the software

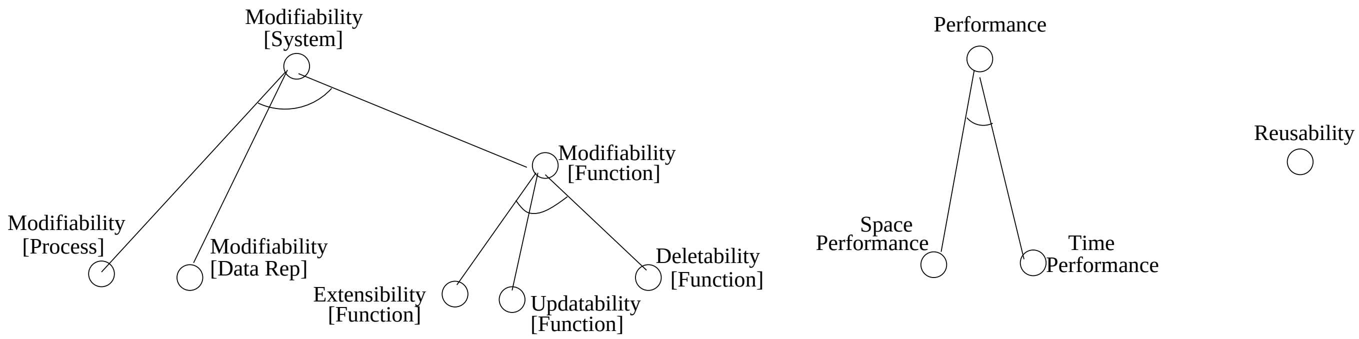

Figure 3: Initial stage of goal graph for KWIC architectural design (based on Garlan and Shaw).

architect starts with; if needs arise, she can add more NFRs during the process.

After posting NFRs as goals to satisfice, the software architect attempts to clarify them, as they firstly can mean many different things to different people, and secondly are too coarse to be put under analysis concerning their interactions. For example, security has different shades of meaning in industrial and military contexts; and performance can be decomposed into time and space considerations. For the purpose of clarification, the architect can decompose each goal either on its sort or on its parameter (Figure 1). The architect focuses on decomposing `Modifiability [System]` on its parameter. After browsing methods in consultation with domain experts, the

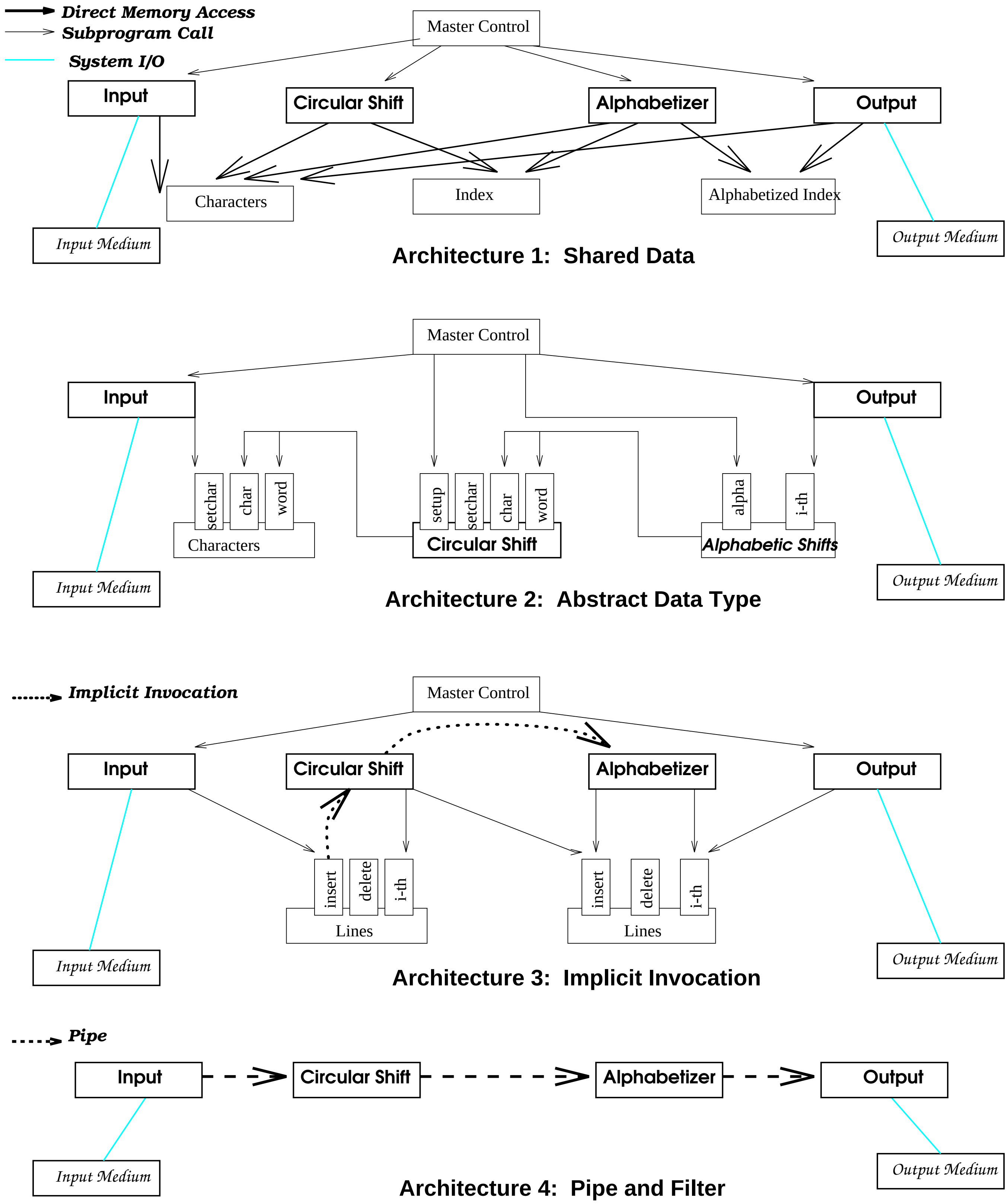

Figure 4: Architectural alternatives for a KWIC system.

architect might decompose the goal into three other offspring goals: `Modifiability [Process]`, `Modifiability [Data Rep]`, and `Modifiability [Function]`. This parameter decomposition method draws on the work by Garlan and Shaw [17], who consider changes in processing algorithm and changes in data representation, and by Garlan, Kaiser, and Notkin [16], who extend the consideration with enhancement to system function.

The software architect further decomposes `Modifiability [Function]`, this time on its sort, into `Extensibility [Function]`, `Updatability [Function]`, and `Deletability [Function]`. This sort decomposition method draws on work by Kazman, Bass, Abowd, and Webb [24], who consider extension of capabilities in terms of adding new functionality, enhancing existing functionality, and deleting unwanted capabilities.

Similarly, the software architect refines `Performance [System]` on its sort into goals for `Space Performance` and `Time Performance`, using a method which draws on work from Nixon [34] [35]; further refinements can address system responsiveness using Smith's principles [41].

Architectural Alternatives. The software architect considers four architectural design alternatives (Figure 4) that Garlan and Shaw [17] discuss (the first two were considered by Parnas [37], and the third is a variant which was considered by Garlan, Kaiser, and Notkin [16]):

1. *Shared Data:* In this design, a main program (Master Control) sequences through the four basic modules: input, shift, alphabetize, and output. Data communication between the modules is carried out by means of shared storage, which is accessed with an unconstrained sequential read-write protocol.

2. *Abstract Data Type:* Instead of direct sharing of data, each module accesses data only by invoking procedures in the interface that each module provides.

3. *Implicit Invocation:* Like the Shared Data design, modules share data, but through an interface. Unlike the Shared Data design, module interaction is triggered by an event. For example, adding a new line to the line storage triggers the Circular Shift module to do the shifting (in a separate abstract shared data store), which in turn causes the Alphabetizer to alphabetize the lines.

4. *Pipes and Filters:* Using a pipeline, each of the four filters processes data and sends it to the next filter. With distributed control, each filter can run (only) when it has data transmitted on pipes.

Design Tradeoffs and Rationale. Architectural design alternatives make partial contributions for or against NFR goals which are (potentially) conflicting or synergistic with one another. Correlation rules can be used to generate new links between existing goals, as well as to suggest the generation of new goals, hence making tradeoffs explicit.

	Shared Data	Abstract Data Type	Implicit Invocation	Pipe & Filter
Modifiability [Process]	– –	–		
Modifiability [Data Rep]	– –	+	–	– –
Space Performance	++		–	– –
Time Performance		–	– –	
Reusability	–	+	+	+

Table 2: Resultant Correlation Table for the KWC Example

During instantiation of correlation rules, uncertainties in the correlation table (e.g., +–, – –*) should be resolved. For example, the software architect could consult domain expert who knows about the characteristics of the intended application domain in order to determine if the Pipe and Filter would significantly hinder the `Space Performance` goal. Once obtained, domain characteristics can be used as an argument (e.g., `support4: expected size of data is huge (from domain expert)`, as shown in Figure 5).

Design rationale can come also from literature. For example, an argument may simply be a citation, such as `support2: [Parnas72]` which supports that the `Shared Data` scheme strongly hurts both `Modifiability [Process]` and `Modifiability [Data Rep]`.

The software architect has instantiated correlation rules to establish both positive and negative links, examined them, rejected or tailored some of them, and provided justifications. Based on Table 1 described earlier in Section 2, Table 2 illustrates a resultant correlation table, as an alternative representation to what is shown in Figure 5, which might aid visual understanding when the number of correlation links is high.

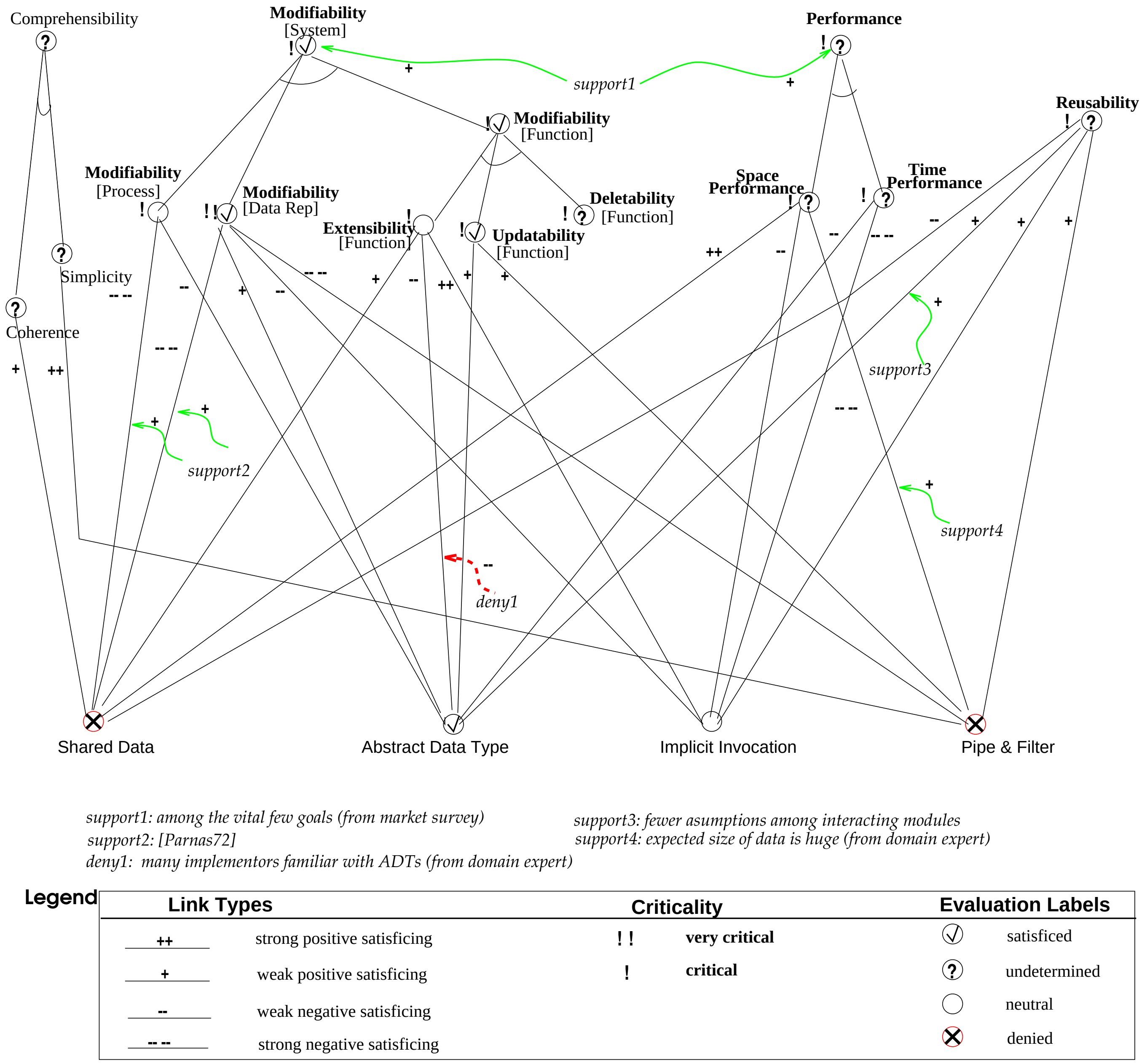

Figure 5: Goal graph selecting among architectural alternatives for a KWIC system.

The software architect also establishes new goals, such as `Comprehensibility [System]`, which add to the number of goals.

Goal Criticality. The software architect, on the one hand, has limited time; she has a number of goal conflict and synergy to deal with, on the other. In order to handle the situation, the architect prioritises goals, here into three categories: *non-critical*, *critical*, and *very critical*. This decision can be justified by way of design rationale. For example, treating modifiability, performance, and reusability as critical goals can be supported, via the *vital few* argumentation method, possibly with a market survey.

With the prioritisation, the software architect can put emphasis on (very) critical goals, and readily resolve goal conflict. For example, as `Modifiability [Data Rep]` is considered very important, architectural design alternatives which strongly hurt the goal might be eliminated from further consideration, here Shared Data and Pipe & Filter.

Evaluation and Selection. A particular architectural design can make a positive, negative, or no contribution to a goal. For example, the use of an abstract data type may help updatability, but at the cost of poorer time performance (Figure 5). Hence, selecting an architectural design requires careful examination of the degree of goal achievement, particularly for critical ones.

Throughout the goal graph expansion process, the evaluation procedure propagates, via labels, the effect of each design decision from offspring to parents. In assessing the degree of goal achievement, the evaluation procedure considers the type of link, and interacts with the software architect when uncertainties arise. Figure 5 shows a stage during software architectural design process, where one alternative is marked acceptable while the others are either *denied* or *neutral*.

This kind of approach is made possible by earler goal reduction as application of "divide-and-conquer" paradigm. Disambiguation and refinement, via decomposition, have facilitated systematic codification of and search for NFR-related reusable knowledge, clearer understanding of tradeoffs, and conflict resolution with design rationale which reflects the needs and characteristics of the intended application domain. Here, for example, the impact of using an abstract data type upon system modifiability is initially unclear, but refinement shows that ADTs support modifiability of data representation, which is critical, but hinder process modifiability.

Throughout the process of architectural design, the software architect has been in control, posting NFR goals, browsing and choosing decomposition methods, design alternatives, and correlations, supporting or denying design decisions by way of design rationale, and observing goal assessment and selecting a particular architectural design.

This process promotes communication, analysis, and comparison of architectural designs and principles. This process is used to support architectural design and results in a history record (with design rationale for both accepted and discarded alternatives considered), for later review, justification and evolution.

4 Related Work

Our proposal draws on concepts, such as elements, components, and connectors, that have been identified as essential to portray architectural infrastructure, as advocated by Perry and Wolf [38], Garlan and Shaw [17], Abowd, Allen, and Garlan [1], Callahan [5], Mettala and Graham [28], and on earlier notions on information system architecture by Zachman [45]. In our view, our emphasis on NFRs is complementary to efforts directed towards identification and formalization of concepts for functional design.

Concerning the role of NFRs, design rationale, and goal assessment, the proposal by Perry and Wolf [38] is of close relevance to our work. Perry and Wolf propose to use architectural style for constraining the architecture and coordinating cooperating software architects. They also propose that rationale, together with elements and form, constitute the model of software architecture. In our approach, weighted properties of the architectural form are justified with respect to their positive and negative contributions to the stated NFRs, and weighted relationships of the architectural form are abstracted into link types and labels, which can be interactively and semi-automatically determined. In our framework, we focus on the problem of systematically capturing and reusing knowledge about NFRs, design alternatives, tradeoffs and rationale.

Kazman, Bass, Abowd, and Webb [24] proposes a basis (called SAAM) for understanding and evaluating software architectures, and gives an illustration using modifiability. This proposal is similar to ours, in spirit, as both take a qualitative approach, instead of a metrics approach. In a similar vein but towards software reuse, Ning, Miriyala, and Kozaczynski [33] proposes an approach (called ABC), in which they suggest the use of NFRs to evaluate the architectural design, chosen from a reuse repository of domain-specific software architectures, which closely meets very high-level requirements. Both SAAM and ABC are product-oriented, i.e., they use NFRs to understand and/or evaluate architectural products; ours, however, is process-oriented, i.e., it provides support for systematically dealing with NFRs *during* the process of architectural design.

The NFR-Framework [7] [30] aims to improve software quality [9] [10] and has been tested on system types with a variety of NFRs, including accuracy, security and performance. Systems studied [13] include credit card [34, 8], public health insurance [7], government administration (Cabinet Documents [7] and Taxation Appeals [35]) and bank loan [12] information systems. The last study considered dealing with changes in requirements, including informativeness. The NFR-Framework also has an associated prototype tool: the NFR-Assistant [11] has been designed and implemented to deal with a variety of NFRs, primarily security, accuracy [7] [8], and (in progress) performance [35]. The NFR-Framework has been one of the subjects in a comparative study on several goal-oriented approaches by Finkelstein and Green [15] who use the meeting scheduler example as a basis of comparison. There are other uses of the NFR-Framework, including organization modelling by Yu [43, 44], and project risk management by Parmakson [36]. In our view, there are parallels to NFR-related work on information systems, and the current paper is one of the first which considers adaptation of the NFR-Framework specifically in the context of software architecture.

5 Conclusion

This paper has proposed an approach to systematically guiding selection among architectural design alternatives, thus providing an alternative to an *ad hoc* approach. Our approach is intended to improve the software architect's ability to understand the high level system constraints and the rationale behind architectural choices, to reuse architectural knowledge concerning NFRs, to make the system more evolvable, and to analyse the design with respect to NFR-related concerns.

More specifically, our approach facilitates explicit representation of NFRs as (potentially) conflicting or synergistic goals to be addressed *during the process* of architectural design, and use of such goals to rationalise the overall architectural design and selection process. Our approach also facilitates codification of knowledge about NFR-related architectural design and tradeoffs, and systematic management and use of such knowledge. In order to help the software architect analyse design tradeoffs, assess goal achievement, and select a particular architectural design, our approach offers an interactive evaluation scheme in which design decisions are rationalised in terms of design rationale which reflects the needs and characteristics of the intended application domain.

The underlying framework has already been applied to information systems. We have studied a number of such systems, considered NFRs which are relevant to them, and provided tool support.

In the context of architectural design, however, our proposal is only preliminary, with its use illustrated only on a pedagogical example. Broader case studies are needed to gain experience and feedback, both benefits and weaknesses, and to see whether our approach can be effectively applied to industrial-strength domain-specific software architectures, application-frameworks, and reference architectures. Our studies of NFRs in the context of information system development have helped us and domain experts evaluate the effectiveness of the framework for such systems. Once a more codified catalogue of architectural methods is developed, it could be used in studies of using the framework to deal with architectural design. Along with feedback from industrial and academic experts, such studies would enhance the framework's coverage and evaluate its usefulness.

A tool which embeds our approach is also needed to assist the software architect to systematically select among architectural design alternatives. Such a tool would incorporate knowledge about a variety of methods and correlations for a wide range of non-functional requirements. Our NFR-Assistant tool could serve as a starting point for such an architecture assistant tool.

We have only illustrated the application of the NFR Framework to selecting among architectural design alternatives at a very abstract level. An important aspect of future work is to deal with more complex architectural problems, and to show the scalability of the framework. This of course requires codification of current and future knowledge about architectural alternatives and design criteria. The results, we trust, would be the provision of more satisfactory goal assessment and guidance for selection.

Acknowledgements

We would like to thank Prof. John Mylopoulos for his support and encouragement for the development of the NFR-Framework. Our gratitude to Barry Boehm, John Callahan, David Garlan, and Dewayne Perry for providing us with references which have been most helpful in identifying the issues and developing the example of this paper. We thank our families for their support and sacrifices.

References

[1] G. Abowd, R. Allen and G. Garlan, "Using Style to Understand Descriptions of Software Architectures", *Software Engineering Notes,* 18(5): 9–20, 1993. *Proc. of SIGSOFT '93: Symposium on the Foundations of Software Engineering.*

[2] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. MacLeod and M. J. Merritt, *Characteristics of Software Quality.* Amsterdam: North-Holland, 1978.

[3] Barry Boehm and B. Scherlis, "Megaprogramming", *Proc. the DARPA Software Technology Conference*, 1992.

[4] B. Boehm, "Software Architectures: Critical Success Factors and Cost Drivers", Proc., *16th Int. Conf. on Software Engineering*, May 1994, p. 365.

[5] J. R. Callahan, *Software Packaging*, Technical Report CS-TR-3093, University of Maryland, 1993.

[6] K. Lawrence Chung, Panagiotis Katalagarianos, Manolis Marakakis, Michalis Mertikas, John Mylopoulos and Yannis Vassiliou, "From Information System Requirements to Designs: A Mapping Framework." *Information Systems,* Vol. 16, No. 4, 1991, pp. 429–461.

[7] K. Lawrence Chung, *Representing and Using Non-Functional Requirements: A Process-Oriented Approach.* Ph.D. Thesis, Dept. of Computer Science, Univ. of Toronto, June 1993. Also Technical Report DKBS–TR–93–1.

[8] Lawrence Chung, "Dealing With Security Requirements During the Development of Information Systems." In Colette Rolland, François Bodat and Corine Cauvet (Editors), *Advanced Information Systems Engineering,* Proc., 5th Int. Conf. CAiSE '93, Paris, France, June 8–11, 1993. Berlin: Springer-Verlag, 1993, pp. 234–251.

[9] Lawrence Chung, Brian A. Nixon and Eric Yu, Using Quality Requirements to Drive Software Development. Presented at the *Workshop on Research Issues in the Intersection Between Software Engineering and Artificial Intelligence,* Sorrento, Italy, May 16–17, 1994.

[10] Lawrence Chung, Brian A. Nixon and Eric Yu, Using Quality Requirements to Systematically Develop Quality Software. *Proceedings, Fourth International Conference on Software Quality,* McLean, VA, U.S.A. October 3–5, 1994.

[11] Lawrence Chung and Brian A. Nixon, "Tool Support for Systematic Treatment of Non-Functional Requirements." Manuscript, December 1994.

[12] Lawrence Chung, Brian A. Nixon and Eric Yu, Using Non-Functional Requirements to Systematically Support Change. *Proceedings, RE '95, The Second IEEE International Symposium on Requirements Engineering,* 27–29 March 1995, York, England.

[13] Lawrence Chung and Brian A. Nixon, Dealing with Non-Functional Requirements: Three Experimental Studies of a Process-Oriented Approach. *Proceedings, 17th International Conference on Software Engineering,* Seattle, WA, U.S.A., April 24–28, 1995.

[14] C. Estrin, R. S. Fenchel, R. R. Razouk, and M. K. Vernon, "SARA (System Architecture Apprentice)", *IEEE Transactions on Software Engineering,* Vol. SE-12, No. 2, Feb. 1986. pp. 293–311.

[15] Anthony C. W. Finkelstein and Stewart J. M. Green, *Goal-oriented Requirements Engineering.* Technical Report TR–93–42, Imperial College (London Univ.), forthcoming.

[16] D. Garlan, G. E. Kaiser, and D. Notkin, "Using Tool Abstraction to Compose Systems," *IEEE Computer,* Vo. 25, June 1992. pp. 30–38.

[17] D. Garlan and M. Shaw, "An Introduction to Software Architecture", To appear in *Advances in Software Engineering and Knowledge Engineering: Vol. I*, World Scientific Publishing Co., 1993.

[18] D. Garlan and D. Perry, "Software Architecture: Practice, Potential, and Pitfalls", *Proc. 16th Int. Conf. on Software Engineering*, 1994, pp. 363–364.

[19] Orlena C. Z. Gotel and Anthony C. W. Finkelstein, An Analysis of the Requirements Traceability Problem. *Proc. Int. Conf. on Requirements Engineering,* Colorado Springs, 1994. IEEE Computer Society Press, 1994.

[20] Sol Greenspan, John Mylopoulos and Alex Borgida, "On Formal Requirements Modeling Languages: RML Revisited." *Proc 16th Int. Conf. on Software Engineering,* Sorrento, Italy, May 1994, pp. 135–147.

[21] M. Jarke, J. Mylopoulos, J. W. Schmidt, Y. Vassiliou, "DAIDA: An Environment for Evolving Information Systems," *ACM Trans. Information Systems*, vol. 10, no. 1, Jan. 1992, pp. 1–50.

[22] Jarke92b Matthias Jarke (Editor), *ConceptBase V3.1 User Manual.* Univ. of Passau, 1992.

[23] J. M. Juran, Frank M. Gryna Jr., and R. S. Bingham Jr. (Eds.), *Quality Control Handbook,*, 3rd Ed., New York: McGraw-Hill Book, 1979.

[24] R. Kazman, L. Bass, G. Abowd and M. Webb, "SAAM: A Method for Analyzing the Properties of Software Architectures", *Proc. Int. Conf. on Software Engineering*, May 1994, pp. 81–90.

[25] J. Kramer, "Exoskeletal Software", *Proc. 16th Int. Conf. on Software Engineering*, May 1994, p. 366.

[26] Jintae Lee, "Extending the Potts and Bruns Model for Recording Design Rationale," *Proc. 13th Int. Conf. on Software Engineering,* Austin, Texas, May 13–17, 1991, pp. 114–125.

[27] Thomas J. McCabe and G. Gordon Schulmeyer, "The Pareto Principle Applied to Software Quality Assurance," In G. Gordon Schulmeyer and James I. McManus (Eds.) *Handbook of Software Quality Assurance,* New York: Van Nostrand Reinhold, 1987, pp. 178–210.

[28] E. Mettala and M. H. Graham (eds.), *The Domain-Specific Software Architecture Program",* Technical Report CMU/SEI-92-SR-9, Carnegie Mellon University, June 1992.

[29] John Mylopoulos, Alex Borgida, Matthias Jarke, and Manolis Koubarakis, "Telos: Representing Knowledge about Information Systems," *ACM Trans. on Information Systems*, vol. 8, Oct. 1990, pp. 325–362.

[30] John Mylopoulos, Lawrence Chung and Brian Nixon, "Representing and Using Non-Functional Requirements: A Process-Oriented Approach," *IEEE Trans. on Software Engineering,* Special Issue on Knowledge Representation and Reasoning in Software Development, Vol. 18, No. 6, June 1992, pp. 483–497.

[31] John Mylopoulos, Lawrence Chung, Eric Yu and Brian Nixon, *Requirements Engineering 1993: Selected Papers.* Technical Report DKBS–TR–93–2, Dept. of Computer Science, Univ. of Toronto, July 1993.

[32] Nils Nilsson, *Problem-Solving Methods in Artificial Intelligence.* New York, McGraw-Hill, 1971.

[33] Jim Q. Ning, Kanth Miriyala and W. (Voytek) Kozaczynski, "An Architecture-driven, Business-specific, and Component-based Approach to Software Engineering", *Proc. Int. Conf. on Software Reusability,* 1993.

[34] Brian A. Nixon, "Dealing with Performance Requirements During the Development of Information Systems." *Proc. IEEE Int. Symp. on Requirements Engineering,* San Diego, CA, January 4–6, 1993. Los Alamitos, CA: IEEE Computer Society Press, pp. 42–49.

[35] Brian A. Nixon, "Representing and Using Performance Requirements During the Development of Information Systems." In Matthias Jarke, Janis Bubenko, Keith Jeffery (Eds.), *Advances in Database Technology - EDBT '94,* 4th International Conference on Extending Database Technology, Cambridge, United Kingdom, March 1994, Proceedings. Berlin: Springer-Verlag, 1994, pp. 187-200.

[36] Priit Parmakson, *Representation of Project Risk Management Knowledge.* M.Sc. Thesis, Institute of Informatics, Tallinn Technical Univ., Tallinn, Estonia, 1993.

[37] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, Vol. 15, Dec. 1972, pp. 1053–1058.

[38] Dewayne E. Perry and Alexander L. Wolf, "Foundations for the Study of Software Architecture", *ACM SIGSOFT Software Engineering Notes,* 17(4), 1992, pp. 40–52.

[39] M. Shaw, "Larger Scale Systems Require Higher-Level Abstractions", *Proc. 5th Int. Workshop on Software Specification and Design,* Pittsburgh, PA, May 1989, pp. 143–146.

[40] Herbert A. Simon, *The Sciences of the Artificial,* Second Edition. Cambridge, MA: The MIT Press, 1981.

[41] Connie U. Smith, *Performance Engineering of Software Systems.* Reading, MA: Addison-Wesley, 1990.

[42] Eric S. K. Yu, Modelling Organizations for Information Systems Requirements Engineering. *Proc. IEEE Int. Symp. on Requirements Eng.,* San Diego, CA, January 4–6, 1993. Los Alamitos, CA: IEEE Computer Society Press, pp. 34–41.

[43] Eric S.K. Yu and John Mylopoulos, 'Understanding "Why" in Software Process Modelling, Analysis, and Design.' *Proc. 16th Int. Conf. on Software Engineering,* Sorrento, Italy, May 1994, pp. 159–168.

[44] Eric Yu, "Modelling Strategic Relationships for Process Reengineering." Ph.D. Thesis, Dept. of Computer Science, Univ. of Toronto, 1994. Also Technical Report DKBS–TR–94–6.

[45] J. A. Zachman, "A Framework for Information Systems Architecture," *IBM Systems Journal,* Vol. 26, No. 3, 1987. pp. 276–292.