Achieving System-Wide Architectural Qualities

Lawrence Chung

Computer Science Program
The University of Texas at Dallas
P. O. Box 830688
Richardson, TX 75083–0688, U.S.A.
chung@utdallas.edu
(972) 883–2178
Facsimile (972) 883–2349

Eric Yu

Department of Computer Science
University of Toronto

Toronto, Ontario, Canada M5S 1A4
eric@cs.toronto.edu
(416) 978–6027
Facsimile (416) 978–1455

System-wide properties such as reliability, availability, maintainability, security, responsiveness, adaptivity, evolvability, survivability, nomadicity, manageability, and scalability (the "ilities", "ities" and the like), are crucial for the success of large software systems. Although these properties have been a major concern of software engineering since its inception, most of the effort on software architecture has focused on achieving functionality. For example, in current visions of component software architectures (CORBA, WWW, ActiveX, etc.), there is no provision for systematically achieving system-wide properties. As noted in the objectives statement of this workshop,

> "assembling components and also achieving system-wide qualities is still an unsolved problem. As long as the code that implements ilities has to be tightly interwoven with code that supports business logic, new applications are destined to rapidly become as difficult to maintain as legacy code."

A Design Space Perspective

Given a requirements description as the problem statement, there can be potentially an infinite number of architectural design alternatives as solutions. For example, an architectural design involves deciding on the number and types of components in the system, the number and types of interactions, the way data is distributed among components, the way processing is distributed among components, etc. Inevitably decisions have to be made on these choices toward a particular final system architecture, and the quality of the architecture chosen is only as good as the decisions obviously. What would be the role of system-wide qualities? In our view, it has to do with the (functional) design space, more specifically narrowing down the space such that the delivered architecture fits to use.

One can classify approaches to quality into product-oriented vs. process-oriented, and quantitative vs. qualitative approaches. For example, measurements of quality attributes of the completed code would be quantitative and product-oriented. Traditionally, approaches to system-wide properties have often been product-oriented, i.e., "build-and-evaluate" — build a product and evaluate it against system-wide properties. If not satisfactory, build another and evaluate it, etc. Unfortunately, however, this kind of (retrofitting) approaches tend to be quite coarse-grained, make over-generalizations, and provide little insight as to how to build a better one the next time, either in the initial development cycle or in the subsequent system evolution cycle. Furthermore, they barely relate functionality to system-wide properties.

A Process-Oriented Approach

We, starting with Chung's dissertation [Chung93], have been focusing on a process-oriented approach to addressing software quality. The approach is requirements-centered, noting that quality issues originate from requirements, and that high quality can be achieved by properly representing quality requirements (also called non-functional requirements, *NFRs*) and analyzing them in the context of the intended applications during requirements engineering. This is a "generative" approach, and it can benefit from a tool support for systematically applying such requirements to guide the exploration of, and selection among, design alternatives during system design. Putting system-wide quality requirements up-front as goals to be achieved is especially important during high-level component design, i.e., in making architectural decisions [CNY95].

In the context of compositional software architectures, this process-oriented approach aims to

- reason about the quality of the whole (i.e., the overall architecture) in terms of the quality of its parts (i.e., the components), and the quality of the parts in terms of the quality of their sub-parts, etc.;
- accommodate the "subjective" nature of system-wide properties by considering the characteristics of the intended applications, hence avoiding over-generalizations;
- to explore the design space and make a rational choice toward a particular system architecture, while performing tradeoff analysis;
- make all the design alternatives, decisions and rationale traceable throughout for fast initial design and subsequent evolutionary redesigns.

The approach is realized in the NFR framework, in which system-wide properties are treated as goals to be achieved. During the architectural design process, goals are decomposed, design alternatives are analysed with respect to their tradeoffs, design decisions are made and rationalised, and goal achievement is evaluated. This way, system-wide properties serve to systematically guide selection among architectural design alternatives. While the "ilities" will be manifested in the design, they are not necessarily inextricably interwoven into the design.

More specifically, in the NFR framework, NFRs are represented as *softgoals* (conflicting or synergistic), and relationships between softgoals as *contribution types* (both partial and full, as well as both "+" and "–"). Each softgoal is associated with a *satisficing status* indicating the degree to which it is satisficed or denied or in conflict. In the NFR framework, ilities are explicitly represented not only textually but diagrammatically in a *softgoal interdependency graph (SIG)*.

This graph helps preventing the designer from putting any lop-sided emphasis on system functionality only. This graph also helps maintain traceability while avoiding ad hoc, accidental design and unjustifiable efforts. During architectural design, every decision can be traceable backward to requirements, and conversely every requirement can be traceable forward to architectural decisions and designs. Naturally the intention behind this graph coincides with one of the goals of this workshop:

> "to architect systems so that both functionality and architectural -ilities can be upgraded over the application's life cycle."

From a product viewpoint, our approach emphasizes that a software product is not just the final code at the end of the development process. Software must continue to evolve so there is no "final" code. The requirements and design knowledge and decisions leading up to code is as much a part of the product as the code itself. They must be represented and encoded in a suitable form to support the ongoing evolution of the product [MBY96].

The NFR framework supports the "generative" process by a body of knowledge of softgoal satisficing represented as generic methods, a body of knowledge of design tradeoffs represented as as correlation rules. and a procedure for propagating satisficing status. The generation of a software architecture is semi-automatic, controlled by a human architect who selects softgoals to refine, selects methods apply, extends and tailors the catalogues of methods, and maximizes synergy and minimizes conflict [CNYMForthcoming].

Status and Open Issues

The NFR-Framework has been tested on several system types with a variety of NFRs: studies of research expense management, credit card, public health insurance and government administration (Cabinet Documents and Taxation Appeals) information systems [CN95]. Adaptation of the framework has also been considered in the context of software architecture [CNY95], and applied to an initial study of dealing with change in a bank loan system, with the combination of performance and requirements for accuracy, timeliness and informativeness [CNY96]. The NFR-Framework also has an associated prototype tool, currently at least to deal with security, accuracy [Chung93], performance [Nixon97] and some other NFRs. The NFR-Framework has been one of the subjects in a comparative study on several goal-oriented approaches [Finkelstein93] who use the meeting scheduler example as a basis of comparison. There are other uses of the NFR-Framework, including organization modelling [Yu94], and project risk management [Parmakson93].

There still are many open questions, as raised in the topics-of-interest statement of this workshop,

> "How to insert them into component software architectures? Say you had a system doing something. How would it look different if ility X was added or removed? Is there some kind of architectural level where the boundary between the visibility/hiddeness of the ility changes? What is needed in the architecture in order to add ilities?"

In our experience, the answers to the questions depend on many factors, such as the type of X, the criticality of X, other ilities important to the system, the type of application(s) the system is intended for, etc. For example, if ility X is about response time and the main service the system provides is sorting and the response times of other components in the system are good enough, addition or removal of X is unlikely to induce any change to the architecture but to the data structure and algorithm below. However, if X is about availability, another replica of the sorting component may be introduced at another server site so that the sorting service can be provided even when the first component fails, hence change in the architecture.

So far as changes are concenred, the NFR framework helps easily and quickly understand changing needs and propagate — through dependencies of system-wide properties, design alternatives, decision and rationale — from changes in the requirements to changes in the design [CNY95].

However, one big challenge to properly achieving system-wide properties in software architectures, we believe, lies in our ability to analyze the architecture at as finer level of granularity as needed. This will involve understanding the "principle of compositionality" in the context of software architectures, namely, "How is the behavior of the whole related to the behavior of the individual components?" and "How is the quality of the whole related to the quality of the components?".

References

[Chung93] K. L. 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.

[CN95] L. Chung, B. A. Nixon, "Dealing with Non-Functional Requirements: Three Experimental Studies of a Process-Oriented Approach." *Proc., 17th ICSE,* Seattle, WA, U.S.A., Apr. 1995, pp. 25–37.

[CNY95] L. Chung, B. A. Nixon and E. Yu, "Using Non-Functional Requirements to Systematically Select Among Alternatives in Architectural Design." *Proc. 1st Int. Workshop on Architectures for Software Systems,* Seattle, Washington, Apr. 1995, pp. 31–43.

[CNY96] L. Chung, B. A. Nixon and E. Yu, "Dealing with Change: An Approach Using Non-Functional Requirements", *Requirements Engineering Journal*, 1(4), pp. 238–259, 1996.

[CNYMForthcoming] L. Chung, B. A. Nixon, E. Yu and J. Mylopoulos, *Non-Functional Requirements in Software Engineering.* Forthcoming.

[Finkelstein93] A. C. W. Finkelstein and S. J. M. Green, *Goal-oriented Requirements Engineering.* Tech. Rept TR–93–42, Imperial College (London Univ.).

[Parmakson93] P. Parmakson, *Representation of Project Risk Management Knowledge.* M.Sc. Thesis, Institute of Informatics, Tallinn Technical Univ., Tallinn, Estonia, 1993.

[MBY96] J. Mylopoulos, A. Borgida, and E. Yu, "Representing Software Engineering Knowledge," *Automated Software Engineering*, 4(3), July 1997, pp. 291–317. Kluwer Academic Publishers.

[Nixon97] B. A. Nixon, "Dealing with Performance Requirements for Information Systems." Ph.D. Thesis, Dept. of Computer Science, Univ. of Toronto, 1997.

[Yu94] E. Yu, *Modelling Strategic Relationships for Process Reengineering.* Ph.D. Thesis, Dept. of Computer Science, Univ. of Toronto, 1995. Also Technical Report DKBS–TR–94–6.