Achieving System-Wide Architectural Qualities

Lawrence Chung

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

Eric Yu

University of Toronto
Faculty of Information Studies

Toronto, Ontario, Canada M5S 3G6
yu@fis.toronto.edu
(416) 978–3107
Facsimile (416) 978–1455

System-wide properties such as reliability, availability, maintainability, security, responsiveness, adaptivity, evolvability, survivability, nomadicity, manageability, and scalability (the "ilities"), 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 [WCSA98],

> "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 potentially be a combinatorially large 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, and so on. Inevitably decisions have to be made on these choices toward a particular final system architecture. Clearly, the quality of the architecture chosen is only as good as the decisions taken to arrive at it.

What then is the relationship between system-wide qualities and architectural design? If we view architectural design in terms of a design space, then the ilities are constraints on that space. If we view the *process* of architectural design as the incremental construction of the design space and the progressive narrowing down of that space towards a "good enough" solution, then the ilities or system-wide qualities can be held as *goals* to explicitly guide the generation of alternatives at each step, and to guide the selection among alternatives throughout the process.

When building systems from components, one needs to pay attention to many system-wide qualities at the same time, since they can interact with each other in many ways. These qualities and their interactions must be identified and analyzed for each design decision. For example, in the absence of any general theories that can relate reliability, maintainability, scalability and a whole host of other architectural properties, one must consider how each of the relevant qualities compete with or complement each other as they apply to each particular design option (i.e., each configuration and choice of components). Many tradeoffs are made as the designer balances out the many desirable qualities. These analyses and tradeoffs need to be made at each level while constructing a compositional architecture.

Without a proper conceptual framework to guide the process, and effective tools to support it, these decisions tend to be *ad hoc*, haphazard (e.g., over-emphasizing some qualities while overlooking some other more important ones), unrecorded, and untraceable. The result is software that are full of implicit design criteria and decisions that are buried in the code, thus making the software extremely difficult to modify as time goes on.

A Process-Oriented, Goal-Directed Approach

Our research group, starting with Chung's dissertation [Chung93], have been focusing on a process-oriented approach to addressing software quality [CNYM98]. 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, as it offers 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;
- accommodate the "subjective" nature of system-wide properties by considering the characteristics of the intended applications, hence avoiding over-generalization;
- explore the design space and make a rational choice toward a particular system architecture, while supporting 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* [Chung93] [CNYM98], 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.

More specifically, in the NFR framework, NFRs (or ilities) are represented as *softgoals* (which is based on a "good enough" (satisficing) approach to qualitative reasoning, and which can be conflicting or synergistic with other softgoals), and relationships between softgoals as *contribution types* (both partial and full, as well as "+" 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 symbolically in an object-based language

The system-wide ilities of the system and their refinements at various levels of composition are diagrammatically represented in a *softgoal interdependency graph (SIG)*. This graph helps prevent the designer from putting a lop-sided, exclusive emphasis on functionality. It therefore complements other diagrammatic architectural notations which emphasize (functional) compositional configurations or the functional design space.

The 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. The intention behind this graph thus 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."

The NFR framework further supports the "generative" process with a body of knowledge of softgoal satisficing represented as *generic methods*, a body of knowledge of design tradeoffs represented as *correlation rules*, and a *graph labeling procedure* for propagating satisficing status. The generation of a software architecture is semi-automated, controlled by a human architect who selects softgoals to refine, selects methods to apply, extends and tailors the catalogues of methods, and maximizes synergy and minimizes conflict [CNYM98].

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]. The "ilities" are explicitly represented and will be manifested in the design and the code, but unlike in legacy systems, they are not inextricably interwoven into the code.

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].

Status and Open Issues

The NFR Framework has been tested on several classes of systems with a variety of NFRs [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]. A prototype tool has been built to deal with security and accuracy [Chung93], performance [Nixon97] and some other NFRs.

There still are many open questions, as raised in the topics-of-interest statement of this workshop. A particularly important challenge in addressing system-wide qualities in software architectures, we believe, lies in the need to analyze an architecture at many levels of granularity. This requires an understanding of the "principle of compositionality" that encompasses both functionality and quality at the same time, i.e., not only "How is the behavior of the whole related to the behavior of the individual components?" but also "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.

[CNYM98] L. Chung, B. A. Nixon, E. Yu and J. Mylopoulos, *Non-Functional Requirements in Software Engineering.* forthcoming monograph.

[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.

[WCSA98] OMG-DARPA-MCC Workshop on Compositional Software Architectures, Monterey, California, January 6–8, 1998.