On Quality Attribute Based Software Engineering

Otto Preiss
Department of Information Technologies
ABB Corporate Research Ltd
5405 Dättwil, Switzerland
otto.preiss@ch.abb.com

Alain Wegmann
Department of Computer Science
Swiss Federal Institute of Technology
1015 Lausanne, Switzerland
alain.wegmann@epfl.ch

Jason Wong
Department of Information Technologies
ABB Corporate Research Ltd
5405 Dättwil, Switzerland
jason.wong@ch.abb.com

Abstract

Software components are an incarnation of architectural means to better cope with the variety of quality aspects of software systems. Unfortunately, architectural artifacts appear somewhat magically sometimes, and so do components. Components are not a major extension to OO in the programming language or functional modeling sense, but a basis to address many of the quality requirements, be they discernable or non-discernable at system runtime. CBSE, being the discipline of engineering with components, is a promising basis to more explicitly and systematically design with and for quality attributes. After defining the context and classifying quality attributes, we first illustrate the important relationship of quality attributes to use case realizations. Second, we argue for components as the fulcrum point for the realization of functional and extra-functional roles. Third, we identify ongoing research directions that we consider conducive towards a software engineering process that supports the design for functional and extra-functional requirements.

1. Introduction

There is hardly any software development process with its set of design methods and in particular modeling techniques that addresses the design with and for quality attributes, i.e. that explicitly focuses on the traceability of quality requirements. Consequently, the prediction of the quality properties of the software system and the degree with which the requirements are going to be met is almost impossible. By quality attributes we mean the large group of properties, sometimes referred to as "Ilities" [1], which are either discernable at runtime (such as dependability, usability, safety, security, consistency) or observable over the product lifecycle (such as extendibility, evolvability, reusability, etc.). As discussed later, we refer to the former category as QoS attributes.

The intention of a software process model is to guide you from use case to code effectively and quickly in a preferably mechanic and prescriptive way. Although current approaches increasingly support capturing of functional as well as extra-functional requirements, they still support designing for the former, but less so for the latter. How then are these extra-functional qualities introduced into today's software systems? The key word is (software) architecture. Almost any quality attribute is dependent on architectural means. In fact, the aspects of architecture are about quality attributes. The results of architectural decisions often manifest themselves in architectural styles, which in turn lead to the realized or chosen hard- and software infrastructure (e.g. frameworks, middleware). Because they are largely independent of the application logic, architectural artifacts hardly fall out of a functional decomposition approach. Similarly, if one follows an OO-based approach to analysis, design and programming, a component is not a natural concept that would obtrude upon a designer during the functional decomposition. In other words, an orderly design of the system's behavior and static organization according to the best principles of OO does not call for any new abstraction. For that matter, components cannot do more than classes and objects can [2]. However, if we look at all the promises of CBSE (classified according to different viewpoints in [3]), it is evident that software components are here to support the different quality-related aspects of a software system[1]. Software components thus represent the incarnation of architectural decisions and constitute architectural abstractions. Because an emphasis on quality attributes, and consequently on (software) architecture, is

[1] For instance, the first commercial components (Visual Basic components) were intended to support quality attributes that are observable over a product or product family lifecycle, predominantly reusability and deployability. Current commercial component technologies (COM+/.NET, EJB) and their frameworks also address quality attributes that are discernable at runtime (as an example see Table 2)

relevant only for systems starting at a certain size and/or complexity, components do not obtrude upon us when we try to develop or analyze toy examples or very small software systems. Extendibility or data integrity across machine boundary (to name a few) is hardly an issue there, which is also why components would be overkill for such toy examples.

Today's industrial practice largely separates the design for functional and extra-functional requirements. While the functional design transforms functional requirements into a logical model that consists of objects and idealized interactions, the extra-functional design (the "architecture") transforms extra-functional requirements into an architectural model that consists of components and possibly frameworks. Merging both design paths yields the final system. However, the fundamental concepts and methodologies for both activities are not considered to be the same, documented in the fact that we still have the co-existing disciplines of OOA/D and software architecture.

This paper represents a plea to more systematically handle quality requirements in software engineering. It argues for the importance of collaborations as first-class design artifacts and introduces the notion of extra-functional roles as the basis for a role-based modeling approach to cope with extra-functional, behavior related, properties. Furthermore, it argues for a number of research directions as well as mainstream architectural approaches related to CBSE, which we consider supportive to systematically dealing with extra-functional properties.

The rest of the document is structured as follows. Section 2 categorizes the various quality attributes and introduces the reification-based, architectural approach to dealing with them. Section 3 shows the relationship of quality attributes to use cases, roles, collaborations, and components. Section 4 concludes with the specific research directions that we believe to be amenable to advance in the area of quality attribute based software engineering.

2. Quality Attributes

The quality of a software system can be assessed by a number of quality attributes. Many of these quality attributes are considered to be systemic, i.e., they are applicable to the entire software system or they are spanning across parts of it. What constitutes the important set of quality attributes is dependent on the stakeholder perspective. For instance, while an end-user may desire performance and usability, the development management may want a high degree of maintainability and reusability.

Many classification structures for quality attributes have been proposed, including elaborate facetted classifications that contain stakeholder, life-cycle and domain dimensions. Hochmüller [5] provides further details and also references to ISO/IEC standards (e.g., ISO 9126) that define many of the "ility"-terms frequently found. For this paper we adopt a simple classification derived and extended from [4].

Quality attributes fall into two classes. The first one refers to quality attributes that are discernable at system execution time. They can be observed by investigating the system behavior during execution. Since these attributes relate to the system behavior they must be part of the behavioral specifications, not at last because they need to be considered in behavior-related design decisions. The second class of qualities cannot be observed at runtime. They usually show during the product life cycle (e.g. maintainability). Although these qualities cannot be observed during runtime, they still need to be considered during the design of a system. Table 1 lists the quality attributes that are frequently found to describe the qualities of a software system[2]. We tried to define a main and a subcategory of attributes, which simplifies the reference to a set of related attributes. Other classifications are of course conceivable (for example, depending on the viewpoint dynamic extensibility could be classified as a quality that is discernable at runtime). Note, the assessment of the achieved level of quality is context dependent and often subjective because established metrics are still rare. Furthermore, some terms are almost synonyms to each other, with distinct meanings for special communities.

2.1. Definition of QoS

Although the term QoS was keyed in the telecommunications area and originally referred to performance related issues on network layers only, it is increasingly being used to refer to other Ilities too (such as listed in Table 1). Manola [1] acknowledges this fact in that he introduces the term IQoS as being the combination of the traditional QoS with the other Ilities. In our terminology and with respect to the above-mentioned classification of quality attributes, we informally define QoS as follows:

QoS attributes refer to the set of extra-functional quality attributes that are discernable at run time and can be tied to a particular use case.

In that context they represent "qualities of behavior", which reconciles with RM-ODP's [6] definition of QoS: "A set of qualities related to the collective behavior of one or more objects". We therefore prefer to use the term quality attributes to refer to all of the Ilities and QoS to

[2] It does not cover the business perspective on qualities, leading to attributes such as costs, time to market, etc.

refer to the quality attributes, which are discernable at runtime. Many places in literature (including the OMG in [7]) refer to quality attributes or QoS as the *non-functional* features of a system. However, we continue using the term *extra-functional* because we agree with the opinion expressed in [4], where non-functional is considered a misleading and even dysfunctional term. Extra-functional also emphasizes the fact that a system's quality requirement at a certain level of abstraction is likely to turn into a functional requirement on a system's component at a certain point in the realization phase.

Table 1. Working classification of quality attributes

Not observable at runtime, but over product life-cycle		*Observable at runtime*	
Main Category	**Sub-category**	**Main Cate-gory**	**Subcategory**
Testability	Accoun-ability	*Use-ability*	Accessibility
Portability	Mobility		Administrability
	Nomadicity		Understand-ability
Integra-bility	Compose-ability	*Depend-ability*	Availability
	Interoper-ability		Degradability
	Adaptability		Durability
	Openness		Reliability
Maintain-ability	Evolvability		Stability
	Extensibility		Survivability
	Modifiability		Fault tolerance
	Changability	*Security*	
	Upgradability	*Safety*	
	Tailorability	*Perfor-mance*	Responsiveness
Reuse-ability			Accuracy
Deploy-ability	Distribute-ability		Footprint
	Configure-ability		Schedulability
			Scalability
			Coherency
			Timeliness
			Integrity

Note, we believe that the separation of quality requirements from the means to address them is important. As an example, a transaction service is not a quality requirement of a software system as stated by the user or customer, but it can be a viable means to support the reliability and integrity requirements of the system. A certain quality attribute is likely systemic when viewed from the realization standpoint, but it might well belong to one single use case (or system operation) only.

The rest of the paper is concerned with the quality attributes that are observable at runtime, i.e. according to our definition with QoS.

2.2. Reification-Based Architectural Solutions

The challenge of separating functional from extra-functional issues led to the wide field of reification-based solutions to dealing with QoS. The idea is to represent key abstractions of a system as explicit objects (in this case related to the fulfillment of QoS requirements), which can then be reasoned about by the designer or even by the system, e.g. through computational reflection [1]. The rationale is to simplify the task of a designer by separating the design of the application into the steps (and even components and objects) needed to realize functional aspects from those that address the QoS requirements. It is current practice to focus on the logical aspects of the design problem first and treat the extra-functional requirements as byproducts of technology and platform issues, i.e. as implementation details. Consequently, they are considered in the final stages of the development.

Table 2. COM+ services and their support for quality requirements

QoS requirement on ...	COM+ Service
Performance (responsiveness, throughput)	In-memory database service; Object pooling
Performance (scalability, responsiveness) and dependability (fault-tolerance)	Dynamic load-balancing service
Dependability (fault-tolerance)	Queued components (message queuing)
Performance (integrity, coherency)	Transaction services
Security	Role based security services
Deployability (configurability) and usability (administrability)	Administration services

Section 3.1 will show that reification-based solutions can result from a systematic modeling and synthesis approach of functional and extra-functional roles.

Reification is the currently prevailing concept in the commercial software world. It is also the basic approach of the OMG to support QoS in their architecture [7]. In general, frameworks such as CORBA, COM+ or EJB provide application meta-services (reifications) and require some architectural support from their components to address some of the commonly required QoS requirements (as an example see Table 2).

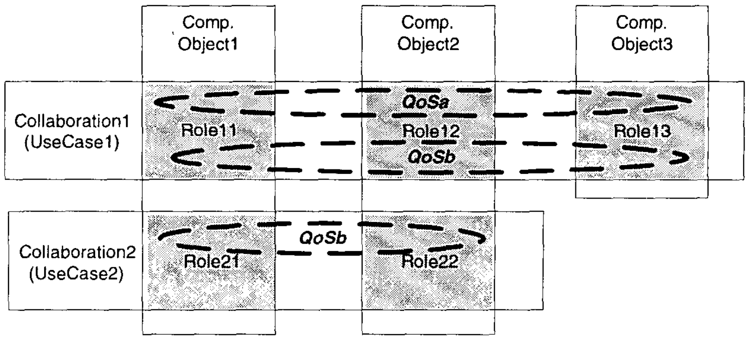

Figure 1. QoSb as a Cross-Cutting Concern of Collaborations

3. Qualities Related to Collaborations of Components

In order to meet quality related user requirements, one must involve actions in the (software) system to be developed. This requires the mapping of such requirements to requirements on the specific realization approach and its technologies (as discussed in detail in [8]).

Functional and extra-functional, but behavior related, requirements are applicable to high-level system operations. If we assume an object-oriented approach they pertain to use cases. Use cases are realized by collaborations of computational objects, as shown in Figure 1 (ignoring the QoS ellipses for now). More specifically, the collaboration is composed of (a) roles, which are played by objects, and (b) the interactions among roles. A certain object may of course play different roles in different collaborations.

The implementation of the object roles and the implementation of their interactions realize these collaborations. QoS requirements must therefore be met by the realization of collaborations, i.e. by the realization of the roles and their connectors. Conceptually, this relationship can be modeled as depicted in Figure 2. The

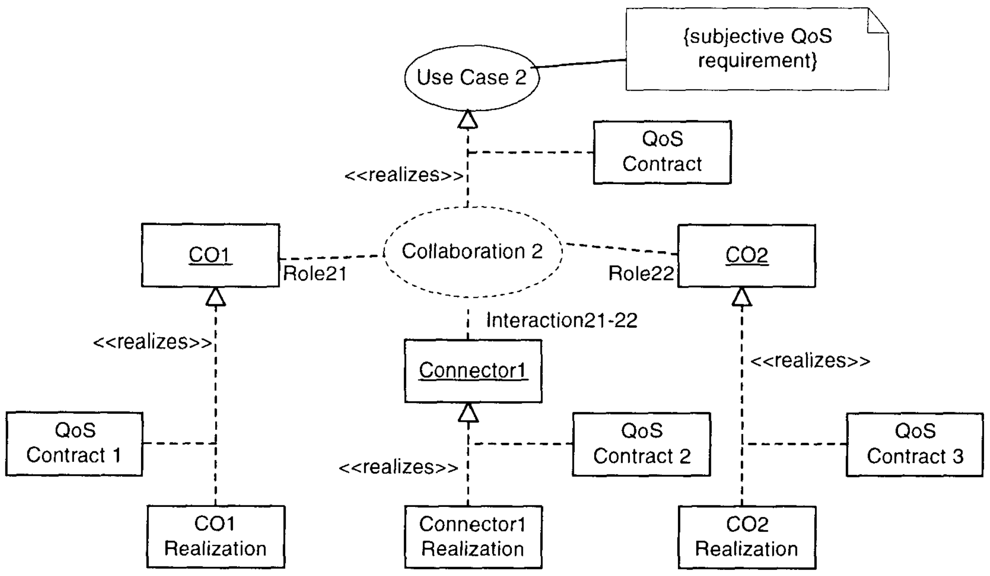

Figure 2. From use case QoS to the realization of its collaboration

notion of a contract is used as the association class of the realization relationship, as introduced by Selic [9] for resource modeling. It represents the commitment of the higher-level concept (e.g. the use case) to these and only these QoS requirements and the commitment of the realizing concept to fulfill the requirements. This departs from the more traditional usage of contract [10] for at least two reasons. First, the contract is a binding element between abstraction levels or different models. Second, a contract is not a "per component" (or even per interface) issue, but rather a self-standing concept realized by the collaboration of entities.

It should be noted that Figure 2 is compatible with the ISO/ITU meta-standard for open distributed processing (RM-ODP [11]). The collaboration (dotted ellipse in Figure 2), which realizes the use case, essentially represents the computational viewpoint and its objects. The roles correspond to ODP interfaces, and the connectors to binding objects. The realization contracts (e.g. "QoS contract 2") represent RM-ODP's environmental contracts that are attached to objects of the computation model. The realizations of the latter with adherence to these contracts correspond to the engineering viewpoint.

3.1. From Computational Object to Software Component

The essence of Figure 1 can be redrawn as depicted in Figure 3. Each of the computational objects plays two roles: one to realize the functional aspect of the use case (Figure 3b, Role21), and another one to realize the quality aspect of it (Figure 3b, QoSb21). In general, the assignment of concrete responsibilities to computational objects ("who does what") is the next natural step to make the joined collaborations more concrete. This design decision is treated as an explicit task in the Catalysis [12] software engineering process. Actions, an exact representation of something that happens between a set of participants, need to be refined into directed or localized actions, i.e. responsibilities assigned to the participating objects. This entails the decision of who is initializing what action, and thus defines the sequence of actions (or in Catalysis words, renders an action to a dialog). In our example, this also requires that the QoSb collaboration be broken down and responsibilities distributed among the computational objects. But what computational object would be the natural place to put the bulk of the responsibility to? To make matters more complicated: based on Figure 1 we know that also our "collaboration 1" is required to meet QoSb. Moreover, the fact that we have "Comp.Object3" involved in collaboration 1, which is not part of collaboration 2, could yield two different design approaches to meet QoSb. The resolution is to adhere to the rules of good design, i.e. design for change. This mandates, firstly, to encapsulate functionality in only a few (well encapsulated and loosely coupled) places, and secondly, if it needs to be distributed, to do it in such a way that later changes can be carried out in a uniform way. These rules lead us to the introduction of a new role, namely that of the "QoSb manager" (see Figure 4). It shall take over the bulk of the responsibility so that the other computational objects are relieved to a hopefully uniform minimal responsibility. The synthesis of the functional with the extra-functional collaboration should thus be the basis for the implementation of a system that meets both the functional and extra-functional requirements that were originally specified for a certain use case. Packaging the realization of both (and usually more) types of roles of a computational object into a single entity yields the application dependent software components (Figure 4). They behave according to the specified functionality and support a well-specified QoS management. The "QoSb manager" would typically become part of a component infrastructure framework.

Note, the integration of the functional with the extra-functional roles results in application components with different degrees of structural dependency

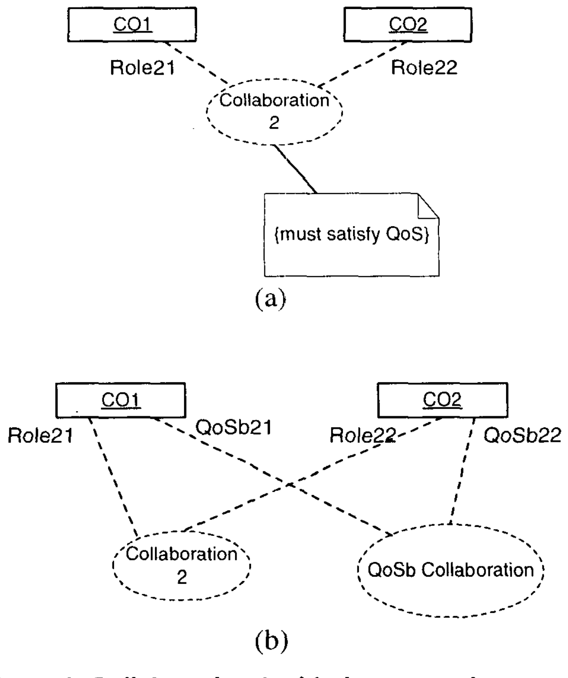

Figure 3. Collaboration 2 with the constraint to fulfill QoSb (a) is broken into two collaborations (b)

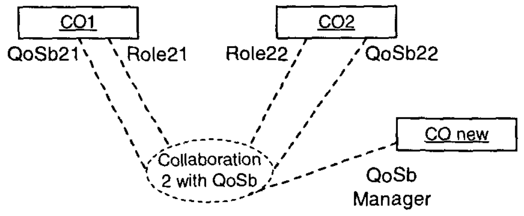

Figure 4. Distributing responsibilities yields a new role

4. Conclusion and Discussion

It was argued, that quality requirements that can be observed during runtime originate at the level of use cases and impact the collaboration by which uses cases are realized. This justifies the need for explicitly capturing (a) the concept of collaboration, (b) the realization of component roles, and (c) the realization of connectors in the realization media. Those needs are currently addressed as follows:

(a) AOP[13] and APPC[14] make collaboration relationships explicit in their programming paradigm. An APPC (Adaptive Plug and Play Component) is even considered a linguistic counterpart of a collaboration diagram for high-level system operations. For QoS considerations, however, these programming approaches fail to acknowledge the explicit visibility and design of connectors. In addition, if an aspect represented a collaboration of roles, crosscutting concerns of aspects would be needed to deal with common quality requirements among different aspects.

(b) Traditional OO design and modeling methods and their functional decomposition techniques cover the realization of roles. Note, since almost all QoS requirements at use case level turn into functional requirements on the architectural artifacts to be realized, the extra-functional roles can be designed with the same OO design methods. Two software engineering methods are of particular interest for our discussion: Catalysis [12] and OOram [15]. The former treats interactions ("actions" in Catalysis terms) and collaborations as first-class units of design work. The latter is specialized on role modeling and role model synthesis and has the advantage of being formally specified.

(c) Treating a connector first-class is not new, although mainly used in the software architecture community [16] [17]. Connectors are key in works around ADL [18] [19]. OO-based functional design methods usually model a connector as an idealized method invocation. We believe that connectors must be handled as self-standing semantic entities because their realizations not only influence but also depend on QoS characteristics. Note, frameworks may well be considered connectors with rather rich protocol semantics. Again, as mentioned in (b), Catalysis with its notion and refinement of abstract connectors seems to be the most promising software design approach.

Software component abstractions are the currently most promising approach for the realization of the combination of application functionality with quality requirements, because they are the tangible interface between software architecture and design of application logic. A role-based approach to coping with extra-functional requirements can be viewed as the underlying model for the currently prevailing, reification-based, architectural solutions. However, since the design for and with extra-functional requirements is not an integral part of current software design approaches, architectural artifacts and their impact on programming seem to appear somewhat magically during implementation or even deployment, i.e. in late phases of a development project.

In order to design with and reason about extra-functional qualities of a software system, as well as to advance in the field of systematically traceable and predictable engineering for quality properties we suggest to discuss the following working areas of research:

(1) Merger of software engineering methods/processes with software architecture so that the design for functional and extra-functional requirements is isomorphic, i.e. the same basic concepts and principles, the same modeling techniques and notation, etc.

(2) Software engineering methods with their decomposition and programming approaches, which, firstly, treat collaborations first-class and thus keep collaboration relationships identifiable down to the program code, and secondly, explicitly allow the modeling of connectors and therefore simplify the mapping of object interactions to architectural connectors.

(3) Specification models that capture both the extra-functional requirements/properties and the notion of collaborations and their involved artifacts in order to be able to express the structural dependency of a software component and the possible interdependencies. For example, the Reusable Asset Specification [20], which defines a set of guidelines and recommendations about the structure, content, and descriptions of reusable software assets, allows for the definition of a reusable asset as the set of entities realizing an entire collaboration through their notion of "asset package".

(4) Formal techniques for merging functional with extra-functional roles.

(5) More rigorous employment of the notion of contracts not only as refinement or realization contracts in functional and architectural models (e.g. framework and

framework service contracts) but also as concepts realized by multiple components with multiple interfaces.

References

[1] F. Manola, "Providing Systemic Properties (Ilities) and Quality of Service in Component-Based Systems," Object Services and Consulting, Inc., Technical Report 1999.

[2] B. Meyer, "The Significance of Components," *Software Development Online*, November, 1999.

[3] D. A. d. Maur, O. Preiss, T. Siegrist, and A. Wegmann, "CBSE and embedded systems - Do they match?," presented at the ECOOP workshop on Pervasive Component Systems, Nice, France, 2000.

[4] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 6 ed: Addison-Wesley, 1999.

[5] E. Hochmüller, "Towards the Proper Integration of Extra-Functional Requirements," *The Australian Journal of Information Systems*, vol. 7, Special Edition 1999 - Requirements Engineering, 1999.

[6] ISO/IEC-JTC1/SC21, "Working Draft for Open Distributed Processing - Reference Model - Quality of Service," ISO/IEC JTC1/SC21 N QoS1, July 1997.

[7] C. Sluman, J. Tucker, J. P. LeBlanc, and B. Wood, "Quality of Service (QoS) OMG Green Paper," Object Management Group, OMG Green Paper Version 0.4a, June 12 1997.

[8] J. O. Aagedal and A.-J. Berre, "ODP-based QoS-support in UML," *IEEE Software*, vol. 8, 1997, pp. 310 - 321.

[9] B. Selic, "A Generic Framework for Modeling Resources with UML," *IEEE Computer*, vol. 33, 2000, pp. 64 - 69.

[10] A. Beugnard, J.-M. Jezeguel, N. Plouzeau, and D. Watkins, "Making Components Contract Aware," *IEEE Computer*, vol. 32, 1999, pp. 38 - 45

[11] ITU-T, "Open Distributed Processing - Reference Model," ITU-T Recommendation X.905 | ISO/IEC 10746, 1996.

[12] D. F. D'Souza and A. C. Wills, *Objects, Components, and Frameworks with UML : The Catalysis Approach*: Addison Wesley, 1998.

[13] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar, "Aspect-Oriented Programming," *ACM Computing Survey*, vol. 28, Article 154, 1996.

[14] M. Mezini and K. Lieberherr, "Adaptive plug-and-play components for evolutionary software development," presented at Conference on Object Oriented Programming Systems Languages and Applications, Vancouver, Canada, 1998.

[15] T. Reenskaug, *Working with Objects - The OOram Software Engineering Method.* Greenwich: Manning Publication, 1996.

[16] M. Shaw, R. DeLine, and G. Zelesnik, "Abstractions and Implementations for Architectural Connections," presented at Third International Conference on Configurable Distributed Systems, Annapolis, Maryland, USA, 1996.

[17] K. J. Sullivan, I. J. Kalet, and D. Notkin, "Evaluating The Mediator Method: Prism as a Case Study," *IEEE Transactions on Software Engineering*, vol. 22, 1996, pp. 563 - 579.

[18] D. Garlan, R. Monroe, and D. Wile, "Acme: An Architecture Description Language," presented at CASCON'97, Toronto, Ontario, 1997.

[19] R. Allan and D. Garlan, "A Formal Basis for Architectural Connection," *ACM Transactions of Software Engineering and Methodology*, vol. 6, pp. 213 - 249, 1997.

[20] Rational Software Corporation and Catapulse Inc., "RAS - Reusable Asset Specification," Draft Recommendation, Nov. 03, 2000.