Carnegie Mellon University
Software Engineering Institute

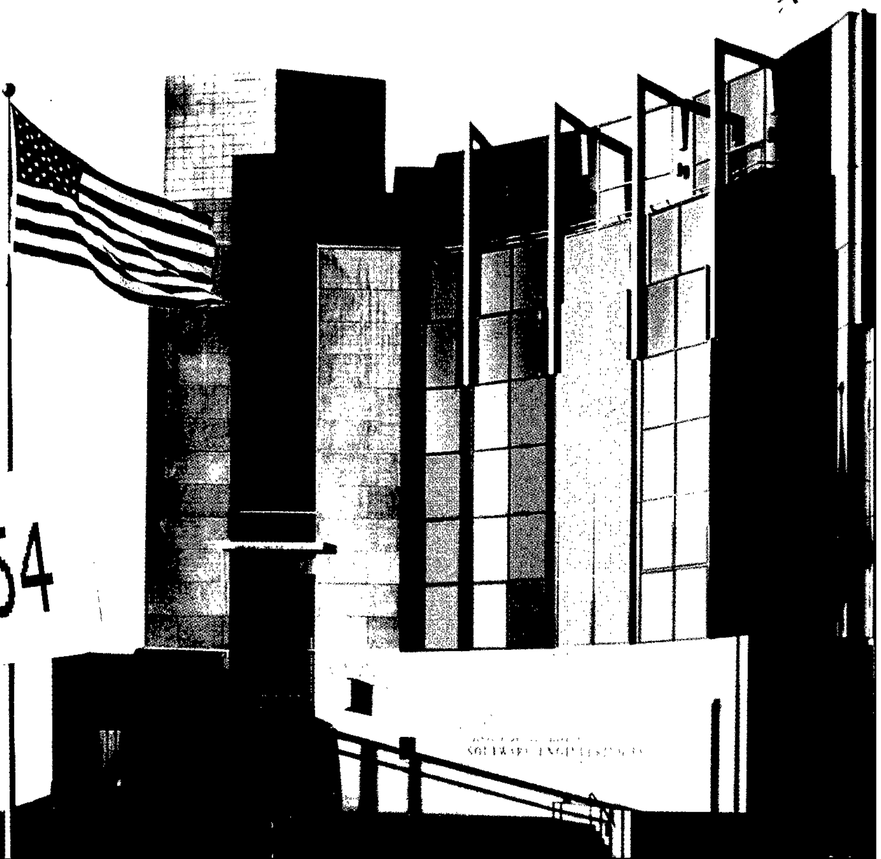
Principles for Evaluating the Quality Attributes of a Software Architecture

Mario R. Barbacci
Mark H. Klein
Charles B. Weinstock

March 1997

TR

Technical Report
CMU/SEI-96-TR-036
ESC-TR-96-036



Technical Report
CMU/SEI-96-TR-036
ESC-TR-96-036
March 1997

Principles for Evaluating the Quality Attributes of a Software Architecture

Mario R. Barbacci
Mark H. Klein
Charles B. Weinstock

Attribute Tradeoff Analysis Initiative

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Table of Contents

1 **Introduction** 1

2 **Overview of the Principles** 3
- 2.1 Information-Gathering Techniques 4
- 2.2 Attribute-Specific Analysis Techniques 6

3 **The Identification of the Contract** 9

4 **The Identification of the Software Architecture** 11
- 4.1 Structure 11
- 4.2 Behavior 11

5 **The Identification of the Hardware Resource Allocation** 13

6 **The Analysis** 15
- 6.1 Example Analysis 15
  - 6.1.1 Reliability Analysis 15
  - 6.1.2 Throughput Analysis 17
  - 6.1.3 Worst-Case Latency Analysis 17

7 **System Profiles and Software Architecture Trade-offs** 19
- 7.1 Conflicts Between Attributes 19
- 7.2 System Profiles 20
- 7.3 Need for Representative Examples 21

8 **Conclusions** 23

References 25

Appendix A Example Scenarios, Questionnaires, and Checklists 27
- A.1 Fault Propagation Paths 27
  - A.1.1 Failure Types 27
  - A.1.2 Pattern of Failure 28
  - A.1.3 Detection Actions 28
  - A.1.4 Containment and Recovery Actions 28
- A.2 Service Paths 29
  - A.2.1 Resource Consumers 30
  - A.2.2 Resources 31
  - A.2.3 Resource Allocation/Mapping 31

Appendix B Glossary of Quality Terms 33

List of Figures

Figure 2-1:	Information Gathering and Analysis	3
Figure 2-2:	Components and Connections	4
Figure 2-3:	Properties of the Evaluation Approaches	5
Figure 6-1:	Reliability Block Diagram	16
Figure 7-1:	Effect of Change on One Attribute	19
Figure 7-2:	Effect of Change on Multiple Attributes	20
Figure 7-3:	System Profiles and Architecture Trade-offs	21

Principles for Evaluating the Quality Attributes of a Software Architecture

Abstract: Software quality is the degree to which software possesses a desired combination of attributes (e.g., reliability, interoperability). In this paper we describe a few principles for analyzing a software architecture to determine if it exhibits certain quality attributes. We show how analysis techniques indigenous to the various quality attribute communities can provide a foundation for performing software architecture evaluation. We also show how the principles provide a context for existing evaluation approaches such as scenarios, questionnaires, checklists, and measurements. Our immediate goal in identifying these principles for attribute-based architecture evaluation is to better integrate existing techniques and metrics into software architecture practice, not necessarily to invent new attribute-specific techniques and metrics. A longer-term goal is to codify these principles into systematic procedures or methods for architecture evaluation. This paper is an initial step towards identifying the ingredients of such methods.

1 Introduction

Software quality is the degree to which software possesses a desired combination of attributes (e.g., reliability, interoperability) [IEEE-1061]. Software quality is one of three user-oriented product characteristics: quality, cost, and schedule. Cost and schedule can be predicted and controlled to some extent by mature organizational processes. However, process maturity does not translate automatically into product quality. Software quality requires mature technology to predict and control quality attributes. If the technology is lacking, even a mature organization will have difficulty producing products with predictable performance, dependability, or other attributes.

Quality, cost, and schedule are not independent. Poor quality eventually affects cost and schedule because software requires tuning, recoding, or even redesign to meet original requirements. Cost and schedule overruns are common because serious problems are often not discovered until the system integration phase.[1]

Few would disagree that it is more cost effective to detect potential software quality problems earlier rather than later in software development. Recently, the software architecture has received attention as being the right focal point for the detection of aberrations in software

1. Horror stories that illustrate the problem are not uncommon. "On Monday, September 2, 1991, at 9 a.m., a $6 million manufacturing support systems/network integration program went live, the largest computer project the company had undertaken. By 10:30 a.m., 'The system had miserably failed,' reported the program manager of a chemical company. 'We had not anticipated needing so much memory, consequently, the system froze in less than two hours, stopping all work at the site' " [Slavin 93]. The quickness of the disaster suggests that the designers were flying blind, so to speak, throughout the development of the system.

quality; the creation of the software architecture is the right time start analyzing software quality and the software architecture is the right artifact to analyze [Abowd 96].[2]

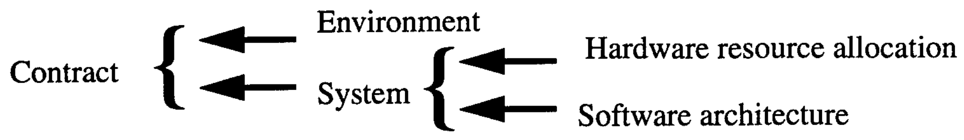
In this paper we describe a few principles for analyzing a software architecture to determine if it exhibits certain quality attributes. We show how analysis techniques indigenous to the various quality attribute communities can provide a foundation for performing software architecture evaluation. We also show how the principles provide a context for existing evaluation approaches such as scenarios, questionnaires, checklists, and measurements [Abowd 96]. Our immediate goal in identifying these principles for attribute-based architecture evaluation is to better integrate existing techniques and metrics into software architecture practice, not necessarily to invent new attribute-specific techniques and metrics. A longer-term goal is to codify these principles into systematic procedures or methods for architecture evaluation. This paper is an initial step towards identifying the ingredients of such methods.

This paper presents a brief overview of a set of principles for performing quality attribute-based architecture evaluation, illustrates attribute specific analyses, suggests the notion of architecture trade-offs based on attribute profiles, and concludes with a summary of the implications of this work.

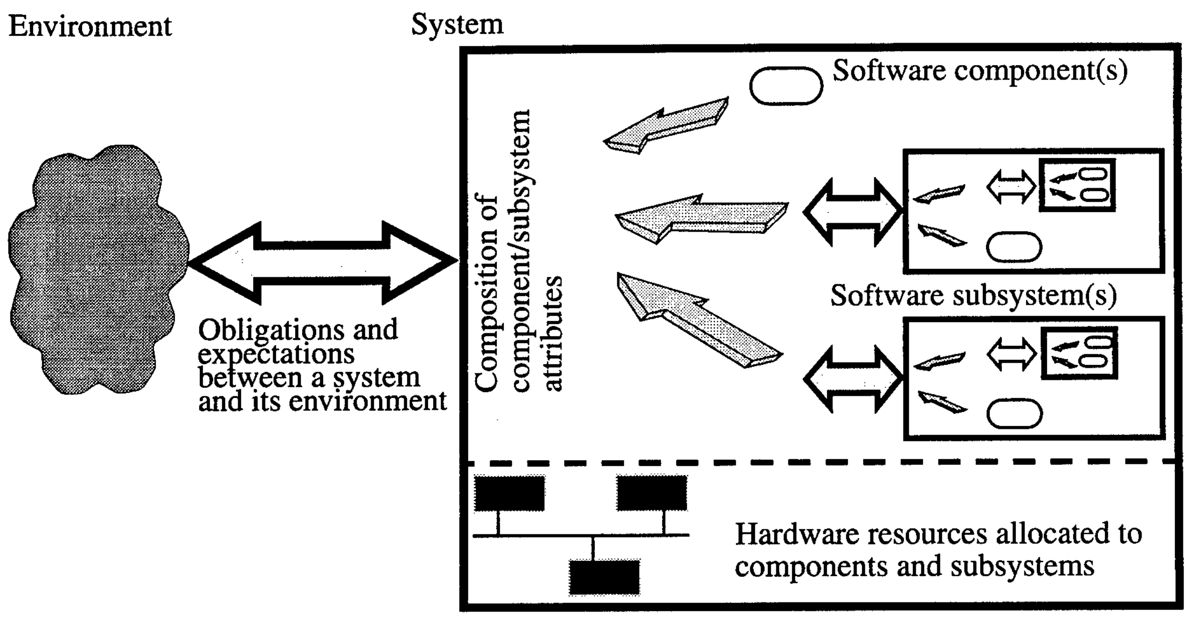
2. There is no agreement on the definition of architecture. For example, in some communities, the artifact that we refer to as the "architecture" is referred to as the "design" and an architecture is a higher level abstraction from which many designs could be derived.

2 Overview of the Principles

In this section we provide a brief description of the principles: the identification of the contract between the system and the environment, the identification of the software architecture, the identification of the hardware resources allocated to the software components, and the analysis of the information gathered by application of the previous principles. Figure 2-1 illustrates the relationships between the types of information needed for the analysis and the recursive nature of this analysis.



(a) Information needed for the analysis


Environment
System
Software component(s)
Composition of component/subsystem attributes
Software subsystem(s)
Obligations and expectations between a system and its environment
Hardware resources allocated to components and subsystems

(b) Recursive analysis to compose attributes and verify contract

Figure 2-1: Information Gathering and Analysis

We will use the system depicted in Figure 2-2 to illustrate the principles underlying an attribute-based architecture evaluation. In this system, three components process input data from the environment and pass their results to a fourth component. The last component in turn sends results back to the environment. For purposes of illustration, we will concentrate on three quality attributes: reliability (the probability that the system will continuously provide outputs over a specified amount of time), worst-case latency (the time elapsed between the arrival of an input to the system and its corresponding output to the environment), and throughput (the system output rate).

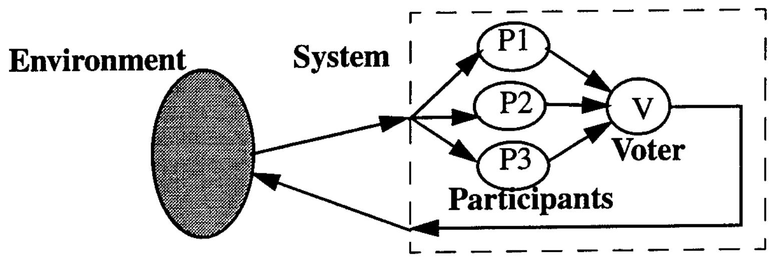


Figure 2-2: Components and Connections

2.1 Information-Gathering Techniques

Quality is relative to the intended use of the system. An evaluation of quality must take into consideration the environment surrounding the system in addition to the system itself. The system can be decomposed and any of its parts can be subject to the analysis. Everything outside of the current focus of interest constitutes its "environment." Therefore the environment is strictly relative to the system being evaluated at the moment and will change as the system changes.

The system typically consists of software and hardware components. For our purposes, we differentiate between these two types of components to focus on the architecture of the software components—i.e., the software architecture. We use the terms subsystem and component to refer to the building blocks of a system.[3] A subsystem is made up of smaller subsystems or components; a component cannot be decomposed.

The system and its environment are partners in a "contract" where the system and the environment both have expectations of each other and where both have obligations to meet these expectations. The expectations of each party must be consistent with the obligations of the other party. The principle of "design by contract" is described by Jézéquiel and Meyer [Jézéquiel 97] as

> *the principle that interfaces between modules of a software system — specially a mission-critical one — should be governed by precise specifications, similar to contracts between humans or companies. The contracts will cover mutual obligations (preconditions), benefits (postconditions), and consistency constraints (invariants).*

Jézéquiel describes the Ariane 5 launcher crash that was caused by a software error—the lack of a precise specification of a reused module [Jézéquiel 97]. Although the focus of the article (and preceding articles in the same column) is on object technology and reuse, the concept of design by contract is applicable to a broader class of software engineering situations.

3. Subsystem/components are relatively neutral terms to describe the structure of a system. The term "process" might be loaded with implications about number of threads of control or processor allocation. A subsystem/component can have any number of threads and can use any number of processors.

Scenarios, checklists, and questionnaires are qualitative techniques applicable to the identification of the contract, the hardware resource allocation, and the software architecture. When conducting an evaluation, one or more of these techniques might be applied, depending on the environment and the system. The appendix illustrates the use of these techniques for identification of fault propagation paths and resource utilization. A comparison of techniques across a number of dimensions is summarized in Figure 2-3.

Review Method	Generality	Level of Detail	Phase	What is Evaluated
Questionnaire	general	coarse	early	artifact process
Checklist	domain-specific	varies	middle	artifact process
Scenarios	system-specific	medium	middle	artifact
Metrics	general or domain-specific	fine	middle	artifact
Prototype, Simulation, Experiment	domain-specific	varies	early	artifact

Figure 2-3: Properties of the Evaluation Approaches[a]

a. This table is taken from Abowd et al., *Recommended Best Industrial Practice for Software Architecture Evaluation* [Abowd 96].

Traditional categories of engineering design knowledge can be useful in identifying the information needed for the analysis. Vincenti defines several categories of knowledge, of which the following two are specially relevant: fundamental design concepts, and criteria and specifications [Vincenti 90].[4]

- The fundamental design concepts are the operational principle and normal configuration. The operational principle defines how the device works: it defines the device and provides criteria for (technical) success. The normal configuration is the shapes or arrangements that are commonly agreed to best embody the operational principle. "Every device possesses an operational principle, and, once it becomes an object of normal every day design, a normal configuration. Engineers doing normal design bring these concepts to their task usually without thinking about them" [Vincenti 90, page 210].

4. The rest of Vincenti's categories (theoretical tools, quantitative data, practical considerations, and design instrumentalities) are more relevant to what we cover under analysis.

- The criteria and specifications are specific, quantitative goals of the design, couched in concrete technical terms, appropriate to the device and its use. During the learning phase of a technology the criteria might be unknown or only partially understood; sometimes they are obvious, sometimes they must be devised consciously, sometimes they are obscure and require great effort and time. "Translation of the utilitarian, usually qualitative, goals of a device into concrete technical terms — and the knowledge required to do it — are critical for engineering design" [Vincenti 90, page 212].

Similar concepts are emerging in the software engineering literature. Shlaer describes a "recursive design method" that requires greater precision in the specification of all system components and relies on automation to produce and assemble these components into the final system [Shlaer 97]. The article characterizes the construction of an architecture as an expert process that, although not understood in detail, could be partitioned into a set of key activities. Although the focus of the article is on object-oriented analysis and synthesis, the first three activities are applicable to a broader class of software engineering situations.

- Characterize the system — "This step elicits those characteristics of the system that should shape the architectural design. [A questionnaire] helps to focus the activity and ensure that all issues are addressed. This questionnaire emphasizes fundamental design considerations regarding size, memory usage, data access time, throughput, identification of critical threads, response time, and the like" [Shlaer, page 63].
- Define conceptual entities — "This step defines and describes precisely the conceptual entities of the architecture and the relationships that must hold among them." "Which objects appear on an OIM [Object Information Model] depend on the concepts inherent in the architecture under construction" [Shlaer, page 66].
- Define theory of operation — "This step describes precisely how the system works as a whole. We have found that an informal but comprehensive theory-of-operation document or technical note works well to develop the appropriate concepts. This document should describe the threads of control that run through the architecture domain, covering all modes in which the system operates, such as normal system operation; cold-start and initialization procedures; warm-start, restart, and failover operation, as required for the delivered system; and shutdown" [Shlaer, page 67].

2.2 Attribute-Specific Analysis Techniques

Depending on the quality attributes of interest, the evaluators can use different qualitative and quantitative techniques to conduct the analysis. These techniques have evolved in separate communities, each with its own vernacular and point of view [Barbacci 95]. Some of the terms and definitions used by these communities can be found in Appendix B on page 33.

Reliability and risk analysis are multi-disciplinary subjects and a number of practical methods are used in routine engineering activities [Modarres 93]. Some of these techniques are applicable to software development. In dependability, qualitative fault forecasting is aimed at identifying, classifying, and ordering the failure modes, or at identifying the event combinations leading to undesirable events. Quantitative fault forecasting (mainly modeling and testing) is aimed at deriving probabilistic estimates of the dependability of the system. For example, reliability growth models [Laprie 90] are aimed at performing reliability predictions from data relative to past system failures.

In safety, the focus is not the system failure but its consequences on the environment—i.e., the hazard. Hazard identification attempts to develop a list of possible system hazards before the system is built. Following the identification of a hazard, a hazard analysis process is used to develop a risk mitigation plan. Hazard identification techniques include brainstorming, consensus techniques (e.g., delphi and joint application design), and hazard and operability analysis (HAZOP). Hazard analysis techniques include fault tree analysis (FTA), event tree analysis (ETA), failure modes and effects analysis (FMEA), and failure modes effects and criticality analysis (FMECA). All of these techniques are standard practices in other engineering disciplines and are being adopted and customized for software development (e.g., HAZOP [Chudleigh 95, MOD 95]).

In security, analysis techniques include formal methods (verify that the design of the system meets the requirements and specification of the security policy), penetration analysis (standard attack scenarios to determine if the system is resilient to these attacks), and covert-channel analysis (to determine the bandwidth of any secondary data channel that is identified in the system).

In performance, analysis methods have grown out of two separate schools of thought: queueing theory and scheduling theory. Queuing analysis is mostly concerned with average case aggregate behaviors. When worst-case behavior is of interest, scheduling analysis might be more appropriate. Formal methods include various forms of timed logic systems [Jahanian 86] or timed process algebras, for example.

3 The Identification of the Contract

This principle requires that the evaluators identify the expectations and obligations of the system. This type of analysis is not inconsistent with the notion of a discovery evaluation in which the incipient architecture specifications are checked against requirements (reviewed in [Abowd 96]).

There are different types of attributes. Some are measured by system activities (e.g., latency, availability), some are measured by inspection activities (e.g., coupling, cohesion), and some are measured by user activities (e.g., time to complete a task). In addition, depending on the attributes of interest, the environment might be an operational environment (e.g., networks, users), a development environment (e.g., life-cycle organizations), or a policy environment (e.g., laws, institutional regulations). These cover a lot of ground. For the evaluation to be meaningful, expectations and obligations must be observable and measurable.

System obligations and expectations are written in the form of scenarios. These scenarios are short descriptions of a requirement, an operational situation, a modification to the system, etc. The following are illustrative examples of scenarios identified from a contract:

- The environment depends on data from the system and expects no more than 1 hour of system down time in a year (minimum availability requirement).
- The system failure rate is less than one failure/month (minimum reliability requirement).
- The environment expects the response time or latency of the system to be less than 100 milliseconds (worst-case latency requirement).
- The system must process up to 20 input events per minute (throughput requirement).
- The system can resist overflooding by excessive rate of input events (security against a type of attack).
- The environment expects the system to allow processor and network upgrades (modifiability of resources).
- The environment provides inputs with exponentially-distributed arrival times with arrival rate λ.

The scenarios define what needs to be confirmed by the analysis.

4 The Identification of the Software Architecture

A software architecture is characterized by a particular combination of software components and connections. This principle requires that the evaluators identify the components and connections within the system.

There are different kinds of connections between components: structure, the component connections showing the flow of data, and behavior, the underlying semantics of the system and the components, including the flow of control. Knowledge of the operational principle and normal configuration [Vincenti 90] are essential to the identification of the architecture.

4.1 Structure

In this example, three components (the "participants") process inputs from the environment. Their outputs feed a fourth component (the "voter") whose output, in turn, goes back to the environment. This portion of the architecture can be identified through questionnaires that elicit components and data flow connections to whatever level of detail is appropriate or desired.

4.2 Behavior

To increase the reliability of the system, the three participants perform redundant (but not necessarily identical) computations and the fourth component, the voter, chooses the "correct" result from the three components as the output from the system.[5] Once the voter detects a faulty participant, it ignores that participant from then on and continues operating with the remainder. If the voter can not make a decision, the voter fail-stops. Once the function (or functions) of each component is defined, this portion of the architecture identification can be carried out through function-specific questionnaires eliciting additional details ("What kind of voting?" "What kind of synchronization protocol?" "What kind of errors are detected?", etc.).

For example, consider the following two basic behaviors for the voter and their variants:

1. majority voting — The voter selects two out of three inputs or else selects two out of two inputs (the non-failed participants must agree) or else shuts down (the system fails). Although the three components take the same inputs and are expected to compute the same values, they do not have to use the same algorithm. Variations include
   - synchronous voting — The voter takes a periodic snapshot with period T_V and makes the decision with whatever inputs it has. When the voter takes a snapshot, it expects to see at least two identical inputs. A faulty participant could send data at the wrong time or with the wrong value. These two types of fault are indistinguishable to the voter.

5. This piece of information about the "semantics" of the system is not derivable from the structure diagram, yet it affects the evaluation of the architecture. Another drastically different interpretation could be that three separate components process three separate data streams, all of which are required to update a common data store that requires mutually exclusive access.

  - asynchronous voting — The voter waits for the inputs but has a timer interval to detect missing inputs.

2. preference voting — The voter selects P1 if P1 is working; or it selects P2 if P2 is working; or it selects P3 if P3 is working; or it shuts down (the system fails). Each input might have a different definition of "working" (e.g., error detection condition). Variations include
   - value error detection — The voter has a reasonableness test or the input data comes with an error flag.
   - time error detection — The voter has an interval timer to time-out.

In addition to the choices of component and voter behavior, availability of the system is affected by repair and reinsertion into service of failed components.

- repairs — Components are repaired as soon as the voter declares them failed. A repair action takes some time to repair.
- no repairs — Components are not repaired.

5 The Identification of the Hardware Resource Allocation

This principle requires that the evaluators collect information about the underlying computation, storage, and communication resources. These are necessary to the analysis because they are consumed by or shared between software components. To the extent that hardware resources are finite or fallible they will have an impact on the overall quality of the system.

The system under consideration is implemented with standard processors and local area networks. To simplify things, assume that we are not using shared memory multiprocessors—that is, memories belong to a processor. Variations include

- independent — Each software component executes on a separate processor.
- shared — All software components execute in the same processor as schedulable units of concurrency (i.e., process).[6] Variations include the following:
  - Priority of voter is higher than that of the participants.
  - Priority of voter is lower than that of the participant's.
    However, when the voter's priority is lower than the priority of the participants, its behavior is sensitive to the behavior of the participants, and thus can be influenced by aberrant participant behavior. This is inconsistent with the voter's purpose to mask such behavior. A mixed processor allocation might be more appropriate for this situation.
- mixed — The three participants share one processor; the voter uses a separate processor. The three participants are schedulable units of concurrency within their shared processor.

Although the number of resource allocation schemes is potentially unbounded, in reality, resources are likely to be chosen from a small collection of commercial, off-the-shelf options. The identification of the hardware resources and their allocation to software components can be carried out through questionnaires and checklists that ask what resources (processors, memory and storage devices, buses, networks) are used by each software component. It is particularly important to identify shared resources, which might not be apparent from the descriptions of the structure and behavior.

6. We're not making any distinctions between processes, tasks, or threads.

6 The Analysis

This principle requires that the evaluators use the information gathered so far to determine if the system will be able to fulfill its obligations. The goal of this analysis is to ensure that components (hardware and software) cooperate in a manner that ensures that the system's obligations are fulfilled. Analysis is performed from the point of view of the system. Part of the analysis takes place during the contract identification, to ensure that the system expectations and system obligations are stated in a manner that is unambiguous and consistent. Of course, this assumes that the system will meet its obligations; this more detailed analysis is necessary to verify that the system can fulfill each of its obligations.

Realizing that an architectural design is still a relatively high-level design, accurate prediction might not be possible. However, the evaluation process will be useful in gaining insights into the system and should continue to be used throughout the development. In the absence of quantitative data, quantitative assumptions (or budgets) can be established and modified as development progresses.

6.1 Example Analysis

For brevity, we will only consider the case of a synchronous majority voter on a shared processor in which the voter has higher priority than the participants and in which failed participants are not repaired. During the identification of the contract, we would obtain a set of scenarios that identify the attributes of interest, such as the following:

- The system failure rate is less than one failure/month (minimum reliability requirement).
- The system must process up to 20 input events per minute (throughput requirement).
- The environment expects the response time or latency of the system to be less than 100 milliseconds (worst-case latency requirement).

The goal of the analysis is to evaluate the quality attributes of the system reliability (R_S), latency (L_S), and throughput (T_S) by composing the attributes of the components and to compare these values with the obligations/expectations of the system.

6.1.1 Reliability Analysis

Different reliability modeling techniques can be used depending on the operating assumptions. In environments where repairs are not feasible (the system fails when all redundancy is exhausted) we can use combinatorial modeling techniques like reliability block diagrams and fault trees to compute the reliability of the system, Rs. In environments where repairs are feasible or where order of events (internal component or subsystem failures) matter, we must use Markov models instead.

Reliability block diagrams work for simple cases, where components are either in series (all must work) or parallel (at least one must work). In this example there are many possible reliability block diagrams, depending on the hardware resource allocation and the software architecture (structure and behavior of the software components).

An initial reliability block diagram could be deduced from the structure given that the reliability of each component (Rp1, Rp2, Rp3, R_V) has been specified. If components share resources, their reliabilities are not independent (they have common-mode failures) and the shared resources must be represented in the block diagram. Finally, depending on the nature of the "voting," the system reliability can vary. For example, a majority voter requires agreement between at least two components to determine the correct output; an averaging voter computes the average of the three inputs (perhaps subject to some "reasonability" test); a priority voter might assign weights to different components (for example, the component executing the simpler or better known algorithm might have a higher weight).

The reliability of the system is computed from the reliability of the components and their interactions. Since there are no repairs, once the voter detects a faulty component, it ignores that component from then on and, of course, expects the other two to agree from then on. If there is a disagreement, the voter fail-stops. For simplicity, assume the three participants are identical, with the same reliability and all three participants execute on the same processor. The reliability block diagram for this system is show in Figure 6-1.

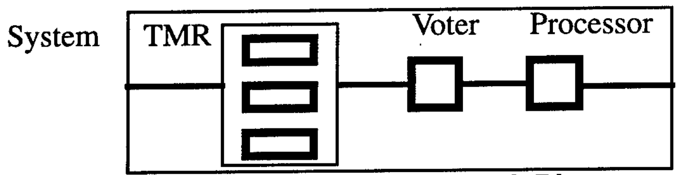


Figure 6-1: Reliability Block Diagram

If we assume that the time to failure is a random event, with exponential distribution, the reliability of a component is given by

$R_{Participant}(t) = e^{-\lambda t}$ where λ is the failure rate of the component.

The reliability of a triple-modular-redundant (TMR) system is [Trivedi 82]

$$R_{TMR}(t) = 3R_{Participant}^{2}(t) - 2R_{Participant}^{3}(t)$$

However, this does not take into account the reliability of the voter or the processor (both the shared processor and the voter must be operating for the system to operate). Thus, to be precise, the reliability of the system is:

$$R_{System}(t) = R_{Processor}(t) \times R_{Voter}(t) \times R_{TMR}(t)$$

Notice that just having replicated components and a voter does not guarantee increased system reliability. The function R_{TMR} is not always greater than $R_{Participant}$. For small values of *t*, R_{TMR} is larger than $R_{Participant}$, and for large values of *t* it is the other way around. Jalote identifies this threshold at $t_0 = 0.7/\lambda$ [Jalote 94, page 36].

6.1.2 Throughput Analysis

The throughput of a system can be calculated using schedulability theory and queueing models. If the voting execution time (C_V) $\leq$ the voting interval (T_V) then the throughput in the synchronous case is 1 vote per interval T_V and therefore 1 output per interval T_V since the voter votes at its own rate independent of the rate at which the input arrives and independent of the execution time of the participants.

Often average latency is a concern that accompanies throughput. For the case of periodic input, the average latency is simply the sum of the average execution times of the voter and the participants. For the case of stochastic input arrivals, assuming Poisson arrivals, the average case latency is a function of the input arrival rate and the average execution time. Standard M/G/1 queuing models can be used to calculate this.

Another potentially important issue is the relationship between input and output. For the synchronous case, if there is to be a one to one correspondence between output and input, the period of the voter must be equal to the input arrival period and $C_m \leq T_m$.

6.1.3 Worst-Case Latency Analysis

The latency of a system can be calculated using schedulability theory and queueing models. Worst-case latency is measured from the time an input arrives to the time the output is sent. If the voter's period is greater than the average interarrival time of the input and it votes on every input, this situation will lead to progressively longer and longer latencies. Therefore, assume that the voter's period is less than the average interarrival time of the input.

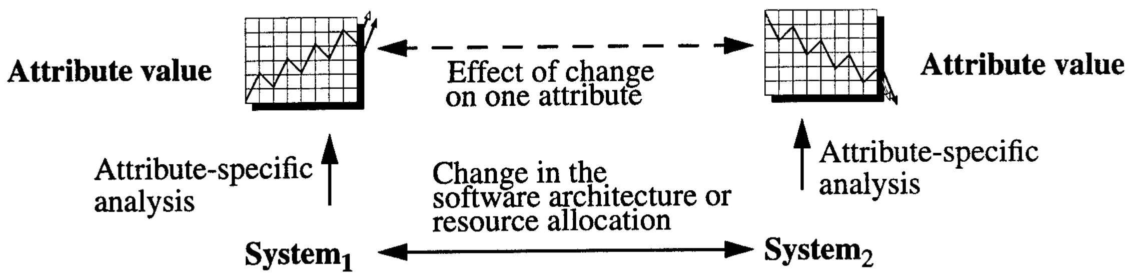
In the worst case, the voter preempts the execution of the participants before they have completely processed their inputs and another period of the voter will have to elapse before voting processing commences. The worst-case latency in this case is $C_p+T_V+C_V$ where C_p is the combined execution time of all participants and C_V is the execution time of the voter.

If the voter is a period out of phase with the arrival of input, the worst-case latency is T_p+C_V.

Therefore, the worst-case latency is $\max(C_p+T_V, T_p)+C_V$.

7 System Profiles and Software Architecture Trade-offs

As illustrated in previous sections, a system can be subject to various types of attribute-specific analysis. That is, for each quality attribute of interest we can apply a specific process to analyze the system from that attribute's point of view. For a given contract, applying the analysis process to two different systems (i.e., systems that differ in software architecture or resource allocation) is likely to yield different results along some attribute specific metric, as suggested in Figure 7-1.



Attribute value
Effect of change on one attribute
Attribute value
Attribute-specific analysis
Change in the software architecture or resource allocation
Attribute-specific analysis
System_1
System_2

Figure 7-1: Effect of Change on One Attribute

These analysis processes are not necessarily formal or quantitative; the process depends on the attribute. Moreover, we do not expect that the various attribute-specific analyses will yield results in some uniform or common units of quality. This is not a new problem, as Boehm observed:

> *Finally, we concluded that calculating and understanding the value of a single overall metric for software quality may be more trouble than it is worth. The major problem is that many of the individual characteristics of quality are in conflict; added efficiency is often purchased at the price of portability, accuracy, understandability, and maintainability; added accuracy often conflicts with portability via dependence on word size; conciseness an conflict with legibility. Users generally find it difficult to quantify their preferences in such conflict situations* [Boehm 78, p. ix].

7.1 Conflicts Between Attributes

Building on Boehm's observations, we do not expect that the effect of a change in the resource allocation or the software architecture can be so finely controlled that only selected attributes change values while other attributes remain constant. As suggested in Figure 7-2, the effect of a change is likely to affect multiple attributes.[7]

[7]. To stress the diverse and not necessarily quantitative nature of the attributes, the icons in the figure suggest attributes like weather (stormy, rainy, sunny), reading light (candle, bulb, moon), and expenses (increasing, decreasing, and oscillating). These are measured in different units and with different techniques.

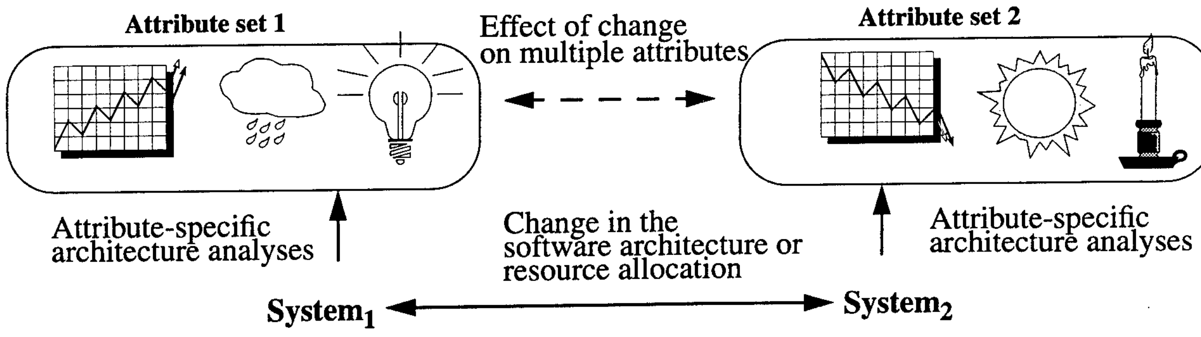

Attribute set 1
Effect of change on multiple attributes
Attribute set 2
Attribute-specific architecture analyses
Change in the software architecture or resource allocation
Attribute-specific architecture analyses
System₁
System₂

Figure 7-2: Effect of Change on Multiple Attributes

Any parameter or piece of information obtained from the contract, the resource allocation, or the software architecture that is used in the analysis of more than one attribute is a source of conflicts between these attributes.

Changing the software architecture or the resource allocation to satisfy an obligation in the contract might have consequences (good or bad) with respect to other obligations because all the attributes affected by the changed feature might have different (better or worse) values.

For example, how we allocate processors to components has an effect on reliability (shared processors are single points of failure), has an effect on throughput (each component only gets the processor for a fraction of the time), and has an effect on latency (communications within the same processor are faster than over a local-area network [LAN]). Changing the processor allocation to decrease latency might also decrease throughput and reliability.

7.2 System Profiles

For a given contract, changes in the software architecture or the resource allocation might lead to different collections of attribute values. The collection of attribute values, software architecture, and resource allocation constitutes the profile of a system.

Profiles can be used as yardsticks to compare systems — one system is better than another system if the former exhibits a better profile. In this case "better" could well be a qualitative, subjective judgement (the emphasis is on satisficing rather than optimizing a set of requirements). The goodness of a profile is always relative to the contract; a profile might look better under a less strict contract, and vice versa: it might look worse under a more strict contract.

Although we might lack precise control of individual quality attribute values, we could still chose between system profiles, as suggested in Figure 7-3. By choosing among system profiles, we are in effect performing a trade-off between their respective software architectures. Even a purely qualitative assessment of profiles would serve as a valuable guide in selecting a software architecture.

7.3 Need for Representative Examples

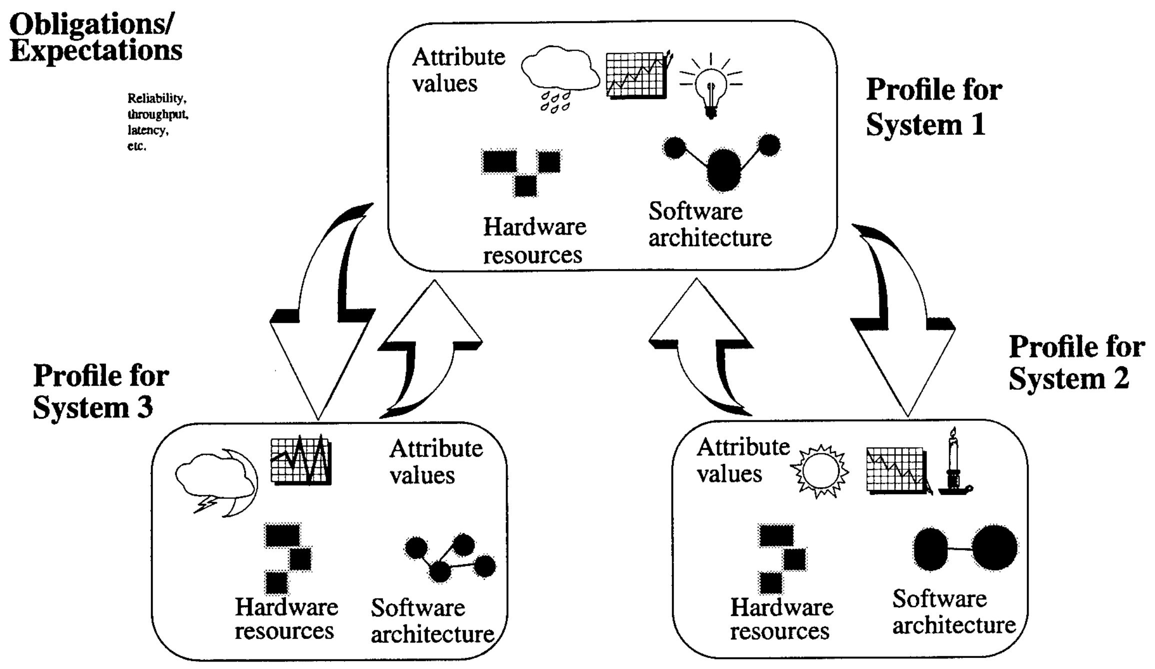


Figure 7-3: System Profiles and Architecture Trade-offs

Clearly there are an infinite number of combinations of contracts, hardware resource allocations and software architectures that could be analyzed. Our hope is that a relatively small number of representative combinations can be used to flesh out the principles and to illustrate the approach for various attributes. This is similar to the approach taken by the ESPRIT Project CASCADE [CASCADE 93]. CASCADE deals with the assessment of safety critical systems—in particular, with the assessment of the software of safety critical systems. CASCADE's goal is to formulate a generalized assessment method for the railway and the automotive sectors. Krebs describes gaps between safety standards and their applicability [Krebs 95]. Some safety standards lack detail about applicable methods and measures to be applied properly; other standards are too detailed and are not only large and complex but also have a short life span; finally, measures recommended in standards are often of dubious efficacy, unconfirmed by actual experience. The approach adopted in CASCADE is that standards should be generic and less complex, and that gaps in levels of detail could be closed by the addition of well-tried, up-to-date examples, evaluated by multiple assessors.

8 Conclusions

We suggest a set of principles that has the potential to unify techniques developed independently by different communities of practitioners and researchers. The analysis techniques associated with each attribute provide the basis for generating scenarios, questions, and checklists. The only information that needs to be gathered is the information that is needed to perform the analysis. Thus, one practice that might emerge from this work is the development of precise statements about (1) the type of information one could expect to see in a component's "attribute specification," (2) the type of attribute obligations a component should be able to make, and (3) the type of attribute compositions a component could be engaged in.

The principles outlined in this paper must be tested by conducting architecture evaluations of real or proposed systems. As we gain experience with architecture evaluations, we might expect to see certain system profiles become more or less desirable, over a range of contracts or contract types. Given a set of requirements (i.e., a collection of attribute values) we could then identify candidate architectures (and resource allocations) to implement the system by matching the requirements with the profiles in the library. Profile identification will then become another principle to include into software evaluation practice.

Merriam-Webster's Collegiate Dictionary (Tenth Edition) defines method as "a systematic procedure, technique, or mode of inquiry employed by or proper to a particular discipline or art". We are a long ways from that level of maturity in the practice but we hope that the experience gained from these experiments will lead to the codification or formalization of systematic procedures—i.e., methods, for conducting attribute-based architecture evaluations.

The analysis techniques for each attribute will change over time. Thus, the methods will require continuous updates, to keep up with current best practices. This is not different from other engineering disciplines; when better information, materials, analysis, etc. become available, the standards evolve. This is not to imply that the current generation of analysis techniques are to be trusted or taken as gospel until something better comes along — there is some skepticism about the benefits claimed by proponents of various software development methods [Fenton 93] and we might be in worse shape than imagined.

The problem of conflict highlighted by Boehm et al. is not going to disappear. However, since one of the principles calls for the explicit identification of obligations and expectations at all levels of decomposition of the system, we have a means to identify possible conflicts between attributes. Any parameter used in the analysis of more than one attribute is a source of conflicts between those attributes. By being explicit about the parameters and the analyses that use them, we are establishing a more methodical, reproducible approach to architecture evaluation and trade-off analysis.

References

Abowd 96 Abowd, G. et al. *Recommended Best Industrial Practice for Software Architecture Evaluation* (CMU/SEI-96-TR-025). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1996.

Barbacci 95 Barbacci, M.R., et al. *Quality Attributes* (CMU/SEI-95-TR-021). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1995.

Boehm 78 Boehm, B. et al. *Characteristics of Software Quality.* New York: Elsevier North-Holland Publishing Company, Inc., 1978.

CASCADE 93 "Certification and Assessment of Safety-Critical Application Development" ESPRIT Project CASCADE-9032 [online]. Available WWW <URL: http://www.newcastle.research.ec.org/esp-syn/text/9032.html> (1993).

Chudleigh 95 Chudleigh, M. F. et al. "A Guideline for HAZOP Studies on Systems which Include a Programmable Electronic System," 42-58. *Proceedings of the 14th Int. Conference on Computer Safety, Reliability, and Security (SAFECOMP95).* Belgirate, Italy, October 11-13, 1995. New York: Springer, 1995.

Fenton 93 Fenton, N. "How Effective Are Software Engineering Methods?" 295-305. *Proceedings of AQUIS '93,* 2nd International Conference on Achieving Quality in Software. Venice, Italy, October 18-20, 1993. Pisa, Italy: IEI-CNR, 1993.

IEEE-1061 IEEE Standard 1061-1992. *Standard for a Software Quality Metrics Methodology.* New York: Institute of Electrical and Electronics Engineers, 1992.

Heimerdinger 92 Heimerdinger, W. L. & Weinstock, C. B. *A Conceptual Framework for System Fault Tolerance* (CMU/SEI-92-TR-33, ADA264375). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1992.

Jahanian 86 Jahanian, F. & Mok, A. "Safety Analysis of Timing Properties in Real-Time Systems." *IEEE Transactions on Software Engineering 12,* 9 (September 1986): 890-904.

Jain 91 Jain, R. *The Art of Computer Systems Performance Analysis.* New York: Wiley, 1991.

Jalote 94 Jalote, P. *Fault Tolerance in Distributed Systems.* New Jersey: Prentice Hall, 1994.

Jézéquel 96 Jézéquel, J-M & Meyer, B. "Design by Contract: The Lessons of Ariane." *IEEE Computer 30,* 1 (January 1997): 129-130.

Kazman 96 Kazman, R. et al. "Scenario-Based Analysis of Software Architecture." *IEEE Software 13,* 6 (November 1996): 47-56.

Klein 93	Klein, M.H. et al. *A Practitioners' Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Boston: Kluwer Academic Publishers, 1993.
Krebs 95	Krebs, H. "Assessment on the Basis of Standards-Gaps and How to Bridge Them," 12-23. *Proceedings of the 14th Int. Conference on Computer Safety, Reliability, and Security (SAFECOMP95).* Belgirate, Italy, October 11-13, 1995. New York: Springer, 1995.
Laprie 90	Laprie, J.C. et al. "The Transformation Approach to the Modeling and Evaluation of the Reliability and Availability Growth of Systems in Operation," 364-371. *Proceedings of the 20th IEEE International Symposium on Fault Tolerant Computing (FTCS-20).* Newcastle Upon Tyne, UK, June 26-28, 1990. Los Alamitos, Ca.: IEEE Computer Society Press, 1990.
Laprie 92	Laprie, J.C., ed. *Dependable Computing and Fault-Tolerant Systems. Vol. 5, Dependability: Basic Concepts and Terminology in English, French, German, Italian, and Japanese.* New York: Springer-Verlag, 1992.
MOD 95	Draft Interim Defense Standard 00-58. "A Guideline for HAZOP Studies on Systems which include a Programmable Electronic System." UK Ministry of Defense 1995.
Modarres 93	Modarres, M. *What Every Engineer Should Know about Reliability and Risk Analysis.* New York: Marcel Dekker Inc., 1993.
Shlaer 97	Shlaer, S. & Mellor, S.J. "Recursive Design of an Application-Independent Architecture." *IEEE Software 14,* 1 (January 1997): 61-72
Slavin 93	Slavin, Lois. "Winning at Integration: Success Comes from Managing the Process with Care." *Enterprise 6,* 4 (April 1993): 25-29.
Trivedi 82	Trivedi, K. S. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications.* Englewood Cliffs, N.J.: Prentice-Hall, 1982.
Vincenti 90	Vincenti, W. G. *What Engineers Know and How They Know It.* Baltimore: The John Hopkins University Press, 1990.

Appendix A Example Scenarios, Questionnaires, and Checklists

Scenarios, checklists, and questionnaires can be used to identify the contract, the resource allocation, and the software architecture. In this appendix we illustrate their use to identify two kinds of information from the software architecture: fault propagation paths (identification of faults and their effects on the system) and service paths (resources required to implement a service).

These are important items to identify because the intersections of fault propagation paths and service paths are sources of risk (e.g., the system might crash). The risk is reduced if there are redundant service paths, if faults can be detected and contained, if faulty components can be replaced, etc.

A.1 Fault Propagation Paths

Faults are undesirable states, events, or conditions that propagate between components of the system. A failure occurs when a fault arrives at the system boundary (i.e., the system has been unable to prevent its propagation to the boundary).

Using fault propagation scenarios, checklists and questionnaires, the evaluators can propose a specific set of faults and assess their effects on the system. The information developed includes the failure types, the pattern of failures, and the detection, containment, and recovery actions available. Most of the cases we list in the following illustrations are derived from Heimerdinger and Laprie [Heimerdinger 92, Laprie 92].

A.1.1 Failure Types

There are several types of failure scenarios to consider.

Timing failures — Timing failures occur when the timing of a service delivered to the environment does not meet a system's obligation. Timing failures are expressed as deviations in time, such as, the event signaling the start/end of the delivery of a service is earlier, later (or "never") than the correct time.

Value failures — Value failures occur when a value delivered to the environment does not meet a system's obligation. Value failures are expressed as deviations in value, such as the computed value is smaller, or larger than the correct value, the distribution of computed values is different from the distribution of the correct value, etc.

Resource failures — Resource failures occur when the use of a resource does not meet a system's obligation. Resourse failures are expressed as deviations in resource utilization, such as, the resource is overused, underused, exhausted, etc.

A.1.2 Pattern of Failure

Patterns of failure are expressed in terms of time of occurrence of events, duration of conditions, distribution or variability of timing and duration, etc. It helps to know how often we need to deal with a fault because containment or repair might consume resources and degrade services, in violation of system obligations. Question to consider include

- Can we determine the immediate cause of the fault or fault activator?
- Is the fault attributable to a single component or resource, or to an interaction between components and resources?
- What is the duration and pattern of the fault activation? Is the fault permanent or transient? Is the fault periodic or aperiodic?
- How fast does a fault propagate and cause a failure? How long does it stay dormant in the propagation chain as it goes from active to dormant to active, etc.?
- Which components or resources are affected by the propagation and in what order?

A.1.3 Detection Actions

Detection actions are expressed in terms of the types of faults observable in different components or resources. Faults can change their type as they propagate (e.g., a value fault injected into a component might emerge later on as a timing fault). Different types of failures violate different obligations and expectations. It helps to know the type and location where can the fault be detected because there might be alternative strategies for containment and repair. Questions to consider include

- How is a fault transformed as it propagates between components? How does it change from a {timing, value, resource} fault to a {timing, value, resource} fault?
- Does a fault always propagate and becomes a failure and does the failure always manifest itself in the same way, to all the observers in the environment? Cases to consider include
  - It may disappear with no perceptible effect.
  - It may remain in place with no perceptible effect.
  - It may lead to a sequence of additional faults that result in a failure (propagation to failure).
  - It may lead to a sequence of additional faults with no perceptible effect on the system (undetected propagation without failure).
  - It may lead to a sequence of additional faults that have a perceptible effect on the system (detected propagation without failure).

A.1.4 Containment and Recovery Actions

Containment and recovery actions are expressed in terms of locality, degradation of service, and cost of repair. Containment or repair actions can have profound effects on system structure, behavior, and resource utilization. It helps to know what containment or repair actions can occur because both during the repair and afterwards the system might be degraded and certain obligations might not be met. Questions to consider include

- Is there built-in redundancy in space? Can we repeat work hoping that fault activations are localized and don't affect all components/resources the same way?
- Is there built-in redundancy in time? Can we repeat work hoping that fault activations are transient and don't affect the same components/resources?
- Is there redundancy in source of time? What kind of clocks, interval timers, protocols are in use?
- Are there fault containment regions? Can we contain value faults by limiting communication and replicating components (i.e., make regions self-contained, with fewer common-mode faults)? Can we contain resource faults by eliminating shared resources? Can we contain timing faults by ignoring an event that should not have happened, generating a missing event, or combining these two to achieve the effect of delaying an early event?
- Is there service degradation? What is the remaining capability to deliver each service? what components/resources are needed to support remaining capability? What services can disappear /persist/reappear?
- Is there fault recovery? What is being repaired or masked? Is the fault removed or is the fault activation suppressed? What is the permanency of the repair action?. Is a fault fixed permanently, not just made dormant albeit for a large amount of time (temporary repair)?
- What is the nature of the repair action? Is it forward recovery, backward recovery, compensation, or masking? What components and resources are needed to execute the repair actions? What is the mean time to repair? How long does it take to execute the repair action?
- Are there fault-free, predictable, consistent states? Can we bring the system to a safe state to replace the component? Can we replace the component at a random time to handle a detected fault?
- Who makes the decision to replace a failed component? Is it the component itself, some central controller, negotiation among other components? If multiple components must agree, which technique is used?

A.2 Service Paths

Performance is the by-product of how resources are allocated and consumed to implement a service. Using scenarios, checklists, and questionnaires the evaluators can identify the resources, the resource consumers, and the allocation or mapping of consumers to resources required to implement a service.

Resource consumers — processes and messages. Resource consumers are initiated by events that propagate through the software architecture.

Resources — processors, storage, networks. As events propagate through the software architecture they use such resources.

Resource allocation/mapping — how consumers are mapped onto resources such as process allocation and process prioritization. Inevitably competition for resources arises, and thus resources must be judiciously allocated to respond to events while meeting performance requirements.

A performance checklist identifies the types of performance attributes the system will have to exhibit

- latency — What is the window of time in which the response to an event must occur? What is the severity of the consequences of not meeting the requirement (i.e., not completing within the response window)?
- jitter — What is the allowable tolerance for deviation from an event prescribed to be generated periodically? Ideally, the event will occur precisely at some point in time.
- precedence — What events are interdependent and what is the dependence (i.e., what events are constrained to occur in some specified order and potentially with latency requirements between them)?
- throughput — How many events per unit time need to be responded to? What is the interval over which throughput must be maintained for each applicable event?
- capacity — How much demand can be placed on the system while continuing to meet latency and throughput requirements? Demand can be thought of in terms of utilization, number of event stream, etc.

Since performance requirements can vary over time, a performance checklist should also identify the various modes of operation and levels of operation.

- List significant modes of operation. How can the demand change over time? How can resources and resource topology change over time?
- List significant levels of performance. What happens when system capacity is exceeded and not all events can be responded to in a timely manner? Are there levels of degradation?

A.2.1 Resource Consumers

Resource consumption is determined by the nature of the software concurrency architecture and the nature of the system event arrivals. Items to include in resource consumer checklists include:

- Identify previously made decisions that impact performance. Are there pre-specified hardware components (e.g., processors, buses)? Are there pre-specified software components (e.g., OS, compiler)? Are there previously made design decisions (e.g., concurrency decisions, synchronization mechanisms)?
- Identify the processes involved in the thread.
- Identify embedded processes. If the component is not a process, does it contain a process or does it execute on a different resource than others in the thread? How are modules (e.g., objects, subprograms) mapped to processes.
- List characteristics of each process that impact performance. What is the priority assignment of each process? What are the estimated execution times for each process? What resources are needed (i.e., which CPU)? Can the resource be used atomically (i.e., non-preemptable sections)?
- Identify interrupts, interrupt handlers, and their priorities. Can interrupts be masked? How long?

- Identify connections between processes for every process in the response thread. Are the connections synchronous or asynchronous? Is there transfer of data or control? What resources are needed (e.g., CPU, LAN, bus)? What is the estimated execution time (resource usage)?
- Identify points in which more than one response thread must synchronize. Is synchronization necessary? Does synchronization involve coordinating the use of a resource? What synchronization protocols are used and what are their characteristics? What interprocess communication mechanisms are used and what are their characteristics?
- Identify event streams. An event stream is a sequence of events from the same source.
- Identify the arrival pattern of each stream. Is is periodic, sporadic or stochastic? What are the worst-case and steady-state patterns?
- Identify the thread through the architecture for each event stream. What components and connections are traversed by the events? What processes and messages are required to respond to each event?

A.2.2 Resources

Items to include in resource checklists include generic characteristics for all resources: speed, amount, allocation units, allocation policy as well as resource-specific characteristics that impact performance:

- CPU — scheduling discipline, number of priority levels, clock granularity, sources of OS overhead, sources of OS-induced blocking, maximum duration of masked interrupts, maximum duration of non-preemptable sections, Interrupt levels for various types of interrupts, processor speed, etc.
- storage — nature of the mutual exclusion mechanism, shared memory, virtual memory, process address space description, address space protection
- network — type of network, network bandwidth

A.2.3 Resource Allocation/Mapping

Items to include in resource allocation/mapping checklists include

- Identify the resources needed by each process/message.
- Identify any constraints on process/message allocation.
- Identify how processes/messages are allocated to processors.
- Estimate how much of each resource is used by each stream.

Appendix B Glossary of Quality Terms[1]

accidental faults — faults created by chance.

active fault — a fault which has produced an error.

aperiodic — an arrival pattern that occurs repeatedly at irregular time intervals. The frequency of arrival can be bounded by a minimum separation (also known as sporadic) or can be completely random.

attribute specific factors — properties of the system (such as policies and mechanisms built into the system) and its environment that have an impact on the concerns

availability — a measure of a system's readiness for use.

benign failure — a failure that has no bad consequences on the environment.

Byzantine failure — a failure in which system users have differing perceptions of the failure.

capacity — a measure of the amount of work a system can perform.

catastrophic failure — a failure that has bad consequences on the environment it operates in.

complex interactions — those of unfamiliar sequences, or unplanned and unexpected sequences, and either not visible or not immediately comprehensible.

component coupling — the extent to which there is flexibility in the system to allow for unplanned events. Component coupling ranges from tight (q.v.) to loose (q.v.)

confidentiality — the non-occurrence of the unauthorized disclosure of information.

consistent failure — a failure in which all system users have the same perception of the failure.

criticality — the importance of the function to the system.

dependability — that property of a computer system such that reliance can justifiably be placed on the service it delivers.

dependability impairments — the aspects of the system that contribute to dependability.

dormant fault — a fault that has not yet produced an error.

error — a system state that is liable to lead to a failure if not corrected.

event — a stimulus to the system signaling the need for the service.

1. This glossary is taken from *Quality Attributes,* Barbacci, et al., pp. 47–51 [Barbacci 95].

event stream — a sequence of events from the same source—for example, a sequence of interrupts from a given sensor.

Event Tree Analysis (ETA) — a technique similar to Fault Tree Analysis. Starting with some initiating (desirable or undesirable) event, a tree is developed showing all possible (desirable and undesirable) consequences.

fail-safe — a system which can only fail in a benign manner.

fail-silent — a system which no longer generates any outputs.

fail-stop — a system whose failures can all be made into halting failures.

failure — the behavior of a system differing from that which was intended.

Failure Modes and Effects Analysis (FMEA) — a technique similar to Event Tree Analysis (ETA). Starting with potential component failures, identifying its consequences. and assigning a "risk priority number" which can be used to determine how effort should be spent during development.

Failure Modes, Effects, and Criticality Analysis (FMECA) — an extension of Failure Modes Effects Analysis (FMEA) that uses a more formal criticality analysis.

fault — the adjudged or hypothesized cause of an error.

fault avoidance — see *fault prevention*.

fault forecasting — techniques for predicting the reliability of a system over time.

fault prevention — design and management practices which have the effect of reducing the number of faults that arise in a system.

fault removal — techniques (e.g., testing) involving the diagnosis and removal of faults in a fielded system.

fault tolerance — runtime measures to deal with the inevitable faults that will appear in a system.

Fault Tree Analysis (FTA) — a technique to identify possible causes of a hazard. The hazard to be analyzed is the root of the tree and each necessary preconditions for the hazard or condition above are described at the next level in the tree, using AND or OR relationships to link subnodes, recursively

halting failure — a special case of timing failure wherein the system no longer delivers any service to the user.

hazard — a condition (i.e., state of the controlled system) that can lead to a mishap.

Hazard and Operability Analysis (HAZOP) — evaluates a representation of a system and its operational procedures to determine possible deviations from design intent, their causes, and their effects.

human-made faults — those resulting from human imperfection.

impairments to dependability — those aspects of the system that contribute to how the system (mis)behaves from a dependability point of view.

inconsistent failure — see *Byzantine failure.*

integrity — the non-occurrence of the improper alteration of information.

intentional faults — faults created deliberately, with or without malicious intent.

interaction complexity — the extent to which the behavior of one component can affect the behavior of other components. Interaction complexity ranges from linear (q.v.) to complex (q.v.).

interlocks — implementation techniques that prescribe or disallow specific sequences of events.

intermittent faults — a temporary fault resulting from an internal fault.

internal faults — those which are part of the internal state of the system.

jitter — the variation in the time a computed result is output to the external environment from cycle to cycle

latency — the length of time it takes to respond to an event.

latency requirement — time interval during which the response to an event must be executed.

latent error — an error which as not yet been detected.

linear interactions — interactions that are in expected and familiar production or maintenance sequence, and those that are quite visible even if unplanned.

lockins — implementation techniques that lock the system into safe states.

lockouts — implementation techniques that lock the system out of hazardous states

loose coupling — characterizes systems in which processes can be delayed or put in standby; sequences can be modified and the system restructured to do different jobs or the same job in different ways; they have "equifinality"—many ways to reach the goal.

maintainability — the aptitude of a system to undergo repair and evolution.

methods — how concerns are addressed: analysis and synthesis processes during the development of the system, procedures and training for users and operators.

mishaps — unplanned events that result in death, injury, illness, damage or loss of property, or environment harm.

mode — state of a system characterized by the state of the demand being placed on the system and the configuration of resources used to satisfy the demand.

observation interval — time interval over which a system is observed in order to compute measures such as throughput.

performance — responsiveness of the system—either the time required to respond to specific events or the number of events processed in a given interval of time.

performance concerns — the parameters by which the performance attributes of a system are judged, specified, and measured.

performance factors — the aspects of the system that contribute to performance.

periodic — an arrival pattern that occurs repeatedly at regular intervals of time.

permanent fault — a fault which, once it appears, is always there.

physical faults — a fault that occurs because of adverse physical phenomena.

precedence requirement — a specification for a partial or total ordering of event responses.

processing rate — number of event response processed per unit time.

quality — the degree to which software possesses a desired combination of attributes (e.g., reliability, interoperability) [IEEE 1061].

reliability — a measure of the rate of failure in the system that renders the system unusable. A measure of the ability of a system to keep operating over time.

response — the computation work performed by the system as a consequence of an event.

response window — a period of time during which the response to an event must execute; defined by a starting time and ending time.

safety — a measure of the absence of unsafe software conditions. The absence of catastrophic consequences to the environment.

safety indicators — the aspects of the system that contribute to safety.

schedulable utilization — the maximum utilization achievable by a system while still meeting timing requirements.

security factors — the aspects of the system that contribute to security.

service — a system's behavior as it is perceived by its user(s).

Software Fault Tree Analysis (SFTA) — an adaptation to software of a safety engineering analysis methodology. The goal of SFTA is to show that the logic contained in the software design will not cause mishaps, and to determine conditions that could lead to the software contributing to a mishap.

spare capacity — a measure of the unused capacity.

temporary fault — a fault which disappears over time.

throughput — the number of event responses that have been completed over a given observation interval.

tight coupling — characterizes systems that have more time-dependent processes: they cannot wait or stand by until attended to; the sequences are more invariant and the overall design allows for very limited alternatives in the way to do the job; they have "unifinality"—one unique way to reach the goal.

timing failure — a service delivered too early or too late.

transient fault — a temporary fault arising from the physical environment.

user of a system — another system (physical or human) which interacts with the former.

utilization — the percentage of time a resource is busy.

value failure — the improper computation of a value.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. **AGENCY USE ONLY** (leave blank)	2. **REPORT DATE** March 1997	3. **REPORT TYPE AND DATES COVERED** Final
4. **TITLE AND SUBTITLE** Principles for Evaluating the Quality Attributes of a Software Architecture		5. **FUNDING NUMBERS** C — F19628-95-C-0003
6. **AUTHOR(S)** Mario R. Barbacci, Mark H. Klein, Charles B. Weinstock		
7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213		8. **PERFORMING ORGANIZATION REPORT NUMBER** CMU/SEI-96-TR-036
9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** HQ ESC/AXS, 5 Eglin Street, Hanscom AFB, MA 01731-2116		10. **SPONSORING/MONITORING AGENCY REPORT NUMBER** ESC-TR-96-036
11. **SUPPLEMENTARY NOTES**		
12.a **DISTRIBUTION/AVAILABILITY STATEMENT** Unclassified/Unlimited, DTIC, NTIS		12.b **DISTRIBUTION CODE**

13. **ABSTRACT** (maximum 200 words)

Software quality is the degree to which software possesses a desired combination of attributes (e.g., reliability, interoperability). In this paper we describe a few principles for analyzing a software architecture to determine if it exhibits certain quality attributes. We show how analysis techniques indigenous to the various quality attribute communities can provide a foundation for performing software architecture evaluation. We also show how the principles provide a context for existing evaluation approaches such as scenarios, questionnaires, checklists, and measurements. Our immediate goal in identifying these principles for attribute-based architecture evaluation is to better integrate existing techniques and metrics into software architecture practice, not necessarily to invent new attribute-specific techniques and metrics. A longer-term goal is to codify these principles into systematic procedures or methods for architecture evaluation. This paper is an initial step towards identifying the ingredients of such methods.

14. **SUBJECT TERMS** attribute-based architecture evaluation, quality attributes, software architectures, software quality			15. **NUMBER OF PAGES** 38 pp.
			16. **PRICE CODE**
17. **SECURITY CLASSIFICATION OF REPORT** UNCLASSIFIED	18. **SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	19. **SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	20. **LIMITATION OF ABSTRACT** UL