Evaluating Software Architectures for Real-Time Systems.

Article *in* Annals of Software Engineering · October 1999

DOI: 10.1023/A:1018902725104 · Source: DBLP

Evaluating Software Architectures for Real-Time Systems

R. Kazman, M. Klein, P. Clements
Software Engineering Institute[1]
Carnegie Mellon University
Pittsburgh, PA 15213 USA

1. Introduction

Software architecture has emerged as the foundational linch pin for designing systems that meet their behavioral and quality requirements, which include real-time constraints. Since architects presumably do not work randomly, but make architectural design decisions based on rational goal-based considerations, it follows that architectures (the sum of their design decisions) can be evaluated to see if the systems to which they lead will in fact meet their requirements.

Given that architecture is the keystone of system development, and given the enormous cost to a project if the architecture is inappropriate for the system being built, why then is not architecture evaluation a standard part of most organizations' system development process?

It may be the case that until now, there has no been no practical, repeatable method for evaluating architectures that was inexpensive enough to apply on a regular basis. This paper introduces the Architecture Tradeoff Analysis Method (ATAM), developed at the Software Engineering Institute. ATAM normally takes about 3-4 calendar days to apply to architectures for medium-sized systems, and can be used to evaluate the appropriateness of a software architecture against its explicitly stated goals (which ATAM also helps to articulate).

In short, ATAM

- facilitates the articulation of specific quality goals for an architecture and the system(s) it spawns;
- identifies sensitivity points in the architecture, design decisions that if changed affect one of the system's desired quality attributes;
- identifies tradeoff points in the architecture, design decisions that if changed affect more than one quality attribute

ATAM not only reveals how well an architecture satisfies particular quality goals for it (such as performance or modifiability), but it also provides insight into how those quality goals interact with each other—how they *trade off* against each other.

1. This work is sponsored by the U. S. Department of Defense.

ATAM draws its inspiration and techniques from three areas: the Software Architecture Analysis Method (SAAM) [3], which was its predecessor; quality attribute communities; and the notion of architectural styles. ATAM is intended to analyze an architecture with respect to its quality attributes, not its functional correctness. Although this is ATAM's focus, there is a problem in operationalizing this focus: We (and the software engineering community in general) do not understand quality attributes well: what it means to be "open" or "interoperable" or "secure" or "high performance" changes from system to system and from community to community.

Recent efforts on cataloguing the implications of using design patterns and architectural styles contribute, frequently in an informal way, to ensuring the quality of a design [1]. More formal efforts also exist to ensure that quality attributes are addressed. These consist of analyses in areas such as performance evaluation [4], Markov modeling for availability [2], and inspection and review methods for modifiability [3].

But these techniques, if they are applied at all, are typically applied in isolation and their implications are considered in isolation. This is dangerous. It is dangerous because *all* design involves tradeoffs and if we simply optimize for a single quality attribute, we stand the chance of ignoring other attributes of importance. Even more significantly, if we do not analyze for multiple attributes, we have no way of understanding the tradeoffs made in the architecture—places where improving one attribute causes another one to be compromised.

The next section discusses architecture evaluation in general. The following section introduces some of the technical concepts central to ATAM. The last section describes the ATAM steps.

2. Evaluating a software architecture

This section lays the conceptual groundwork for architectural evaluation. It defines what we mean by software architecture, explains the kinds of properties for which an architecture can (and cannot) be evaluated, and introduces the role of architectural styles in evaluating for quality attributes.

First, let's establish what it is we're evaluating:

> *The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them. [5.]*

The architecture defines the components (such as modules, objects, processes, subsystems, compilation units, and so forth) and the relevant relations (such as "calls", "sends data to", "synchronizes with", "uses", "depends on", "instantiates," and many more) among them. The architecture is the result of early design decisions that are necessary before a group of people can collaboratively build a software system.

Why evaluate an architecture?

One of the truths about architecture that motivates architecture evaluation is that architectures largely allow or preclude nearly all of the system's quality attributes. This is particularly true of real-time systems: If your system has stringent performance requirements, then you need to pay attention to inter-component communication and intra-component deadlines. But it applies to other quality attributes as well: If you have modifiability goals for your system, then you need to pay attention to encapsulation properties of your module components. If security is a consideration, then you need to pay attention to inter-component communication and data flow, and perhaps introduce special components such as secure kernels, encrypt/decrypt functions, or impose authentication protocols between processes. If reliability is important, then the architecture needs to provide redundant components with warm or hot restart protocols among them. And so forth. All of these approaches to achieving quality attributes are architectural in nature, having to do with the decomposition of the total system into parts and the ways in which those communicate and cooperate with each other to carry out their duties.

Why evaluate an architecture? Simply put, the earlier you find a problem in a software project, the better off you are. Architecture evaluation is cheap insurance. ATAM is meant to be applied while the architecture is a paper specification, and so it involves running a series of simple thought experiments. ATAM requires assembling relevant stakeholders for a structured session of brainstorming, presentation, and analysis. All told, ATAM evaluations add no more than a few days to the project schedule.

Who is involved in architecture evaluation?

There are three groups of people involved in an architecture evaluation.

1. Project decision-makers. These are people who are interested in the outcome of the evaluation and have the power to make decisions that affect the future of the project. They include the architect, designers of components, and the project's management. Management will have to make decisions about how to respond to the issues raised by the evaluation. In some settings (particularly government acquisitions), the customer or sponsor may be a project decision-maker as well.
2. Evaluation team. These are the people who will conduct the evaluation and perform the analysis. The team members and their precise roles will be defined later, but for now simply realize that they represent one of the three classes of participant.
3. Stakeholders. Stakeholders are people who have a vested interest in the architecture, and the system that will be built from it. The ATAM method uses stakeholders to articulate the specific requirements that are levied on the architecture, above and beyond the requirements that state what functionality the system is supposed to exhibit. Some, but not all, of the stakeholders will be members of the development team: coders, integrators, testers, maintainers, and so forth.

What does the evaluation produce?

In concrete terms, an architecture evaluation produces a report. Primarily, though, an architecture evaluation produces information. In particular, it produces answers to the following kinds of questions:

1. Is this architecture suitable for the system for which it was designed?
2. Which of two or more competing architectures is the most suitable one for the system at hand?

Suitability for a given task, then, is what we seek to investigate. We say that an architecture is suitable if

- the system that results from it will meet its quality goals. That is, the system will run predictably and fast enough to meet its performance (timing) requirements. It will be modifiable in planned ways. It will meet its security constraints. It will provide the required behavioral function. Not every quality property of a system is a direct result of its architecture, but many are, and for those that are the architecture is suitable if it provides the blueprint for building a system that achieves those properties.
- the system can be built using the resources at hand: the staff, the budget, the legacy software (if any), and the time allotted before delivery. That is, the architecture is *buildable*.

Suitability is only relevant in the context of specific (and specifically articulated) goals for the architecture and the system it spawns. The first major step of ATAM is to capture explicit, specific goals that the architecture must meet in order to be considered suitable. In a perfect world, these would all be captured in a requirements document, but as we shall see, this notion fails for two reasons. First of all, complete and up-to-date requirements documents don't always exist. But second, requirements documents express the requirements for a system, and there are additional requirements levied on an architecture besides just enabling the system's requirements to be met. (Buildability is an example.)

We are not interested in precisely characterizing any quality attribute (using measures such as mean time to failure or end-to-end average latency). That would be pointless at an early stage of design because the actual parameters that determine these values (such as the actual execution time of a component) are often implementation-dependent. What we are interested in doing—in the spirit of a risk mitigation activity—is learning where an attribute of interest is affected by architectural design decisions, so that we can reason carefully about those decisions, model them more completely in subsequent analyses, and devote more of our design, analysis, and prototyping energies on such decisions

What are the costs and benefits of performing an architecture evaluation?

We have already stated the main, and obvious, benefit of architecture evaluation: It uncovers problems early in the development cycle that if left undiscovered would be orders of magnitude more expensive, if not impossible, to correct later. In short, architecture evaluation produces better architectures. Even if the evaluation uncovers no problems that warrant attention, it will increase everyone's level of confidence in the architecture. But there are other benefits as well. Some of them are hard to measure, but they all contribute to a successful project and a more mature organization; hence, they are extremely important. The following is a list of the benefits we've often observed.

- Facilitates communication among stakeholders, and between the stakeholders and the architect.
- Forces articulation of specific quality goals.
- Results in prioritization of conflicting goals.

- Forces a clear explication of the architecture.
- Improves the quality of architectural documentation.
- Uncovers opportunities for cross-project reuse. \
- Results in improved architectures.

The benefits discussed above all accrue to the project whose architecture is on the table. But there are also benefits to future projects in the same organization. A critical part of ATAM consists of probing the architecture using a set of quality-specific analysis questions, and neither the method nor the list of questions is a secret. The architect is perfectly free to arm himself before the evaluation by making sure that the architecture is up to snuff with respect to the relevant questions. This is rather like scoring well on a test whose questions you've already seen, but in this case it isn't cheating: it's professionalism.

The costs of architecture evaluation are all personnel costs, and opportunity costs related to those personnel participating in the evaluation instead of something else. A typical ATAM exercise consumes three days, and consists of 3-4 evaluators, 6-12 stakeholders, and roughly 4 project representatives, for a total of 40-60 staff days.

Scenarios

Quality goals form the basis for architectural evaluation, but quality goals by themselves are not a sufficient basis to judge an architecture for suitability. Often, requirements statements such as the following are written:

- "The system shall be robust."
- "The system shall be highly modifiable."
- "The system shall be secure from unauthorized break-in."
- "The system shall exhibit acceptable performance."

Without elaboration, each of these statements is subject to interpretation and misunderstanding. What you might think of as "robust," your customer might consider barely adequate—or vice versa. Perhaps the system can easily adopt a new database, but cannot adapt to a new operating system. Is that system maintainable or not? Perhaps the system uses passwords for security, which prevents a whole class of unauthorized users from breaking in, but does nothing to keep out anyone who has stolen a password. Is that system secure from intrusion or not?

The point here is that quality attributes are not absolutes, but are meaningful only in the context of specific goals. In a perfect world, the quality requirements for a system would be completely and unambiguously specified in a requirements document that is completed before the architecture begins. In reality requirements documents are not written, or are written poorly, or are not finished when it is time to begin the architecture. So the first job of an architecture evaluation is to elicit the specific quality goals against which the architecture will be judged.

The mechanism we use is the *scenario.* A scenario is a short statement describing an interaction of one of the stakeholders with the system. A user would describe using the system to perform some task; his scenarios would very much resemble *use cases* in object-oriented parlance. A maintenance stakeholder would describe making a change to the system, such as upgrading the operating system in a particular way or adding a specific new function. A developer’s scenario might talk about using the architecture to build the system or predict its performance. A customer’s scenario might describe the architecture re-used for a second product in a product line, or might assert that the system is buildable given certain resources.

In ATAM we use three types of scenarios: *use cases* (these involve typical uses of the existing system and are used for information elicitation); *growth scenarios* (these cover anticipated changes to the system the system), and *exploratory scenarios* (these cover extreme changes that are expected to “stress” the system). These different types of scenarios are used to probe a system from different angles, optimizing the chances of surfacing decisions at risk. Examples of each type of scenario follow.

Use cases:

1. *The ship’s commander* requires the his authorization before releasing certain kinds of weapons.
2. *A domain expert* wants to determine how the software will react to a radical course adjustment during weapon release (e.g., loft) in terms of meeting latency requirements?
3. *A user* wants to examine budgetary and actual data under different fiscal years without re-entering project data.
4. *A user* wants to have the system notify a defined list of recipients by e-mail of the existence of several data-related exception conditions, and have the system display the offending conditions in red on data screens.
5. *A tester* want to play back data over a particular time period (e.g., last 24 hours).

Growth scenarios:

1. Make the head-up display track several targets simultaneously.
2. Add a new message type to the system’s repertoire.
3. Add collaborative planning: 2 planners at different sites collaborate on a plan.
4. The maximum number of tracks to be handled by the system doubles.
5. Migrate to a new operating system, or a new release of the existing operating system.

Exploratory scenarios:

1. Add new feature 3-D maps
2. Change the underlying platform to a MacIntosh.
3. Re-use the software on a new generation of the aircraft.
4. The time budget for displaying changed track data is reduced by a factor of 10.
5. Improve the system’s availability from 95% to 99.99%.

Each exploratory scenario stresses the system. Systems were not conceived to handle these modifications, but at some point in the future these might be realistic requirements for change.

Attribute taxonomies

As the selected scenarios are traced through the architecture, we ask questions that are intended to be probing. Actually the questions are inspired questions, inspired by the wealth of knowledge that already exists in the various quality attribute communities; knowledge in the performance, reliability and security communities that can be drawn on to enhance our ability to elicit architectural information relevant to quality attribute goals.

To facilitate using this information we have created a taxonomy for each attribute. Attribute information is placed into three basic categories: *External stimuli, architectural parameters,* and *responses*. External stimuli (or just stimuli for short) are the events that cause the architecture to respond or change. Architectural parameters comprise topological and component properties of the architecture that will affect a response. To assess an architecture for adherence to quality requirements, those requirements need to be expressed in terms that are measurable or at least observable. These measurable/observable quantities are described in the responses section of the taxonomy.

For performance, for example, the external stimuli are events arriving at the system (such as messages or missiles). The architectural parameters are processor and network arbitration mechanisms, concurrency structures including processes, threads and processors and properties including process priorities and execution times. Responses are characterized by measurable quantities such as latency and throughput. For modifiability, the external stimuli are changes to the system, the mechanisms for controlling the cost of changes are encapsulation and import coupling, and the measurement is in terms of the type and amount of change. The figures below show a portion of the performance taxonomy.

The taxonomies help to ensure attribute coverage, but just as importantly offer a rationale for asking elicitation questions. For example, latency (a measure of response) is a function of

- resources such as CPUs and LANs;
- resource arbitration such as scheduling policy
- resource consumption such as CPU execution time;
- and external events such message arrivals.

Therefore, given a scenario such as *"Unlock all of the car doors within 1 second of pressing the correct key* sequence", the taxonomy inspires questions such as:

- Is the 1 second deadline a hard deadline (response)?
- What are the consequences of not meeting the 1 second requirement (response)?
- What components are involved in responding to the event that initiates unlocking the door (architectural parameters)?
- What are the execution times of those components (architectural parameters)?
- Do the components reside on the same or different processors (architectural parameters)?
- What happens if several "unlock the door" events occur quickly in succession (stimuli)?

We refer to these questions as *elicitation questions*. Elicitation questions are inspired by the attribute taxonomies and result from applying the taxonomy to architecture being evaluated.

Attribute taxonomies also inspire screening questions. While the goal of elicitation questions is to facilitate the comprehensive elicitation of relevant attribute-specific information, the goal of screening questions is to guide or focus the elicitation. Architecture evaluations are relative-

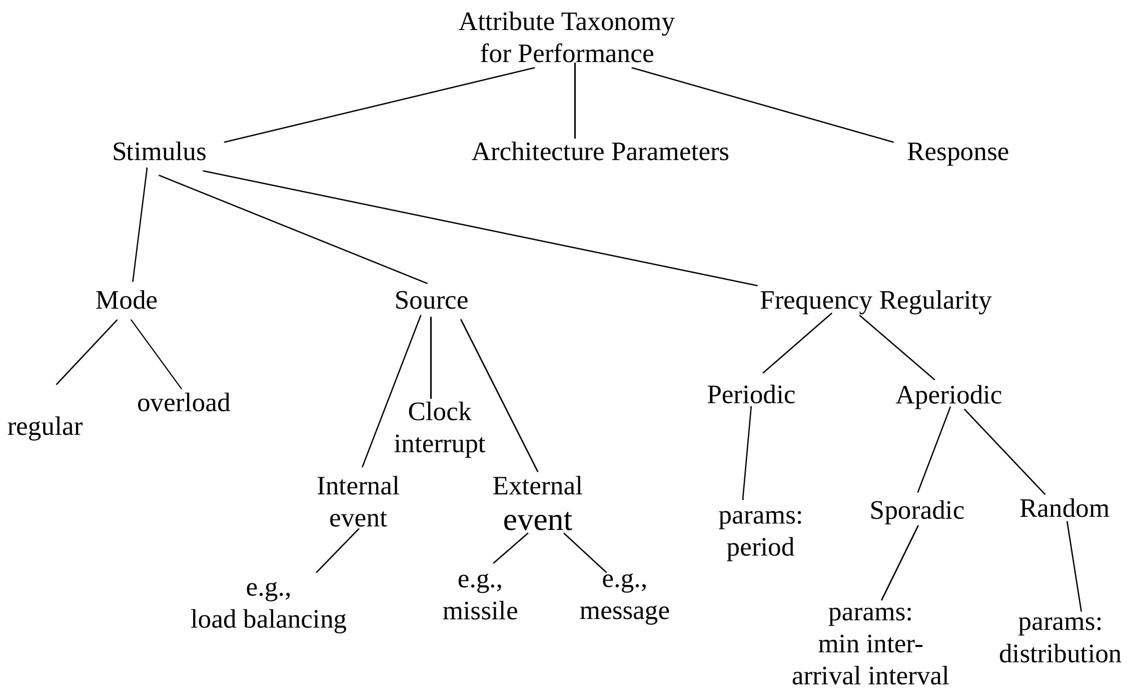

Figure 1: Performance Taxonomy: Stimuli

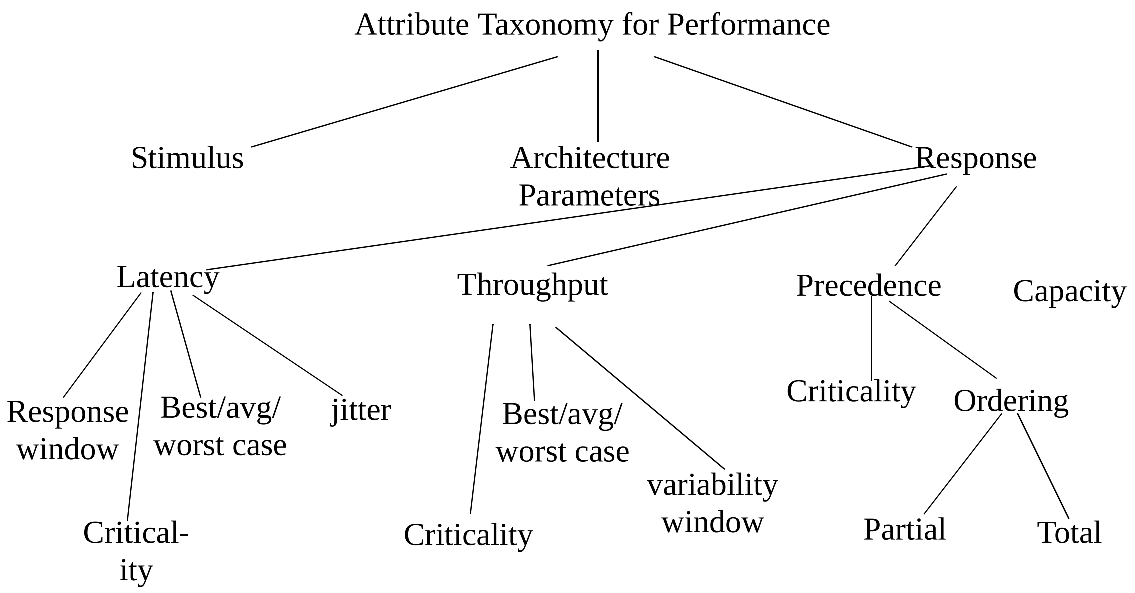

Figure 2: Performance Taxonomy: Response

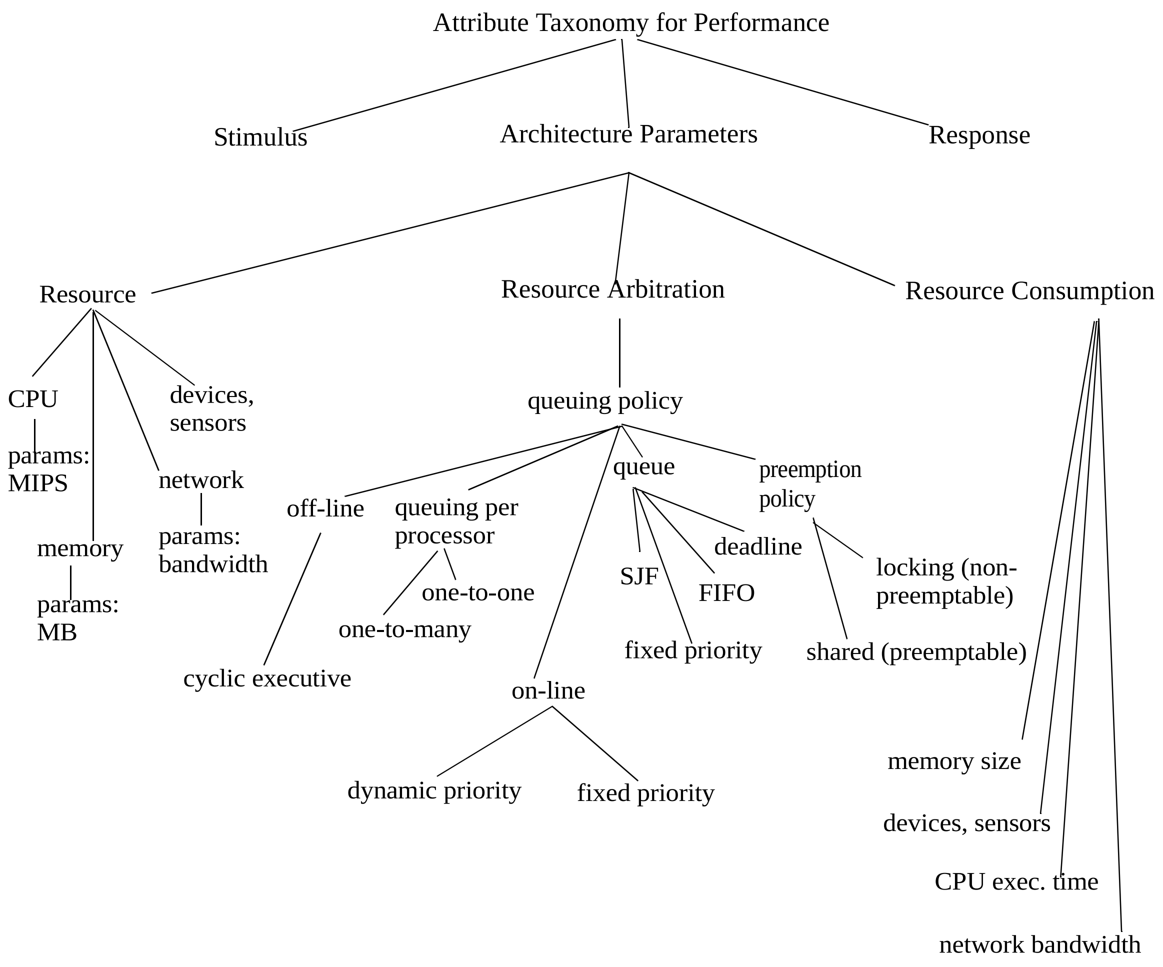

Figure 3: Performance Taxonomy: Architectural Parameters

ly short. For large systems (and even for subsystems of large systems) it will be very difficult to ensure complete coverage during a two or three day evaluation. Screening questions accelerate the process by guiding the evaluators to the more “influential” places in the architecture. Here’s a sample of screening questions:

- What components are likely sites of future changes (modifiability)?
- What interfaces are likely sites of future changes (modifiability)?
- What shared data types are likely sites of future changes (modifiability)?
- Are there any system resources shared by many/most portions of the system (performance)?
- Are there critical paths through the system for which end-to-end latency is important (performance)? On those paths are there processes that dominate utilization?
- Are there critical paths through the system for which total throughput is important (performance)?
- Are there critical system services without which the system cannot operate (reliability)?

- Is there information for which it is important to prevent unauthorized access (security)? If so, what components are responsible for handling this information? Are there multiple access paths to this information and what are they?

Representing architectural information

While an architecture is an abstraction, the goal of which is to highlight key design information and hide less important detail, a proper representation of an architecture is still replete with information. Thus it is important to have a way of managing this information throughout the development process, including architectural design and analysis, detailed design, and implementation. This information provides the basis for ensuring that the detailed design and implementation conform to the architecture. Managing the information starts with suitably representing it because the representation will affect how well the architecture serves as a communication vehicle, how accurately it describes the system's key abstractions and how well it supports analysis. Clarity is important to support communication. Completeness and precision are needed to support analysis.

While there are many software structures, there are five canonical or foundational structures that provide the basis for describing an architecture. These five structures are derived from Kruchten's "4+1 views" (we divided his logical view into function and code structures). These five structures plus the appropriate mappings between them can be used to completely describe an architecture.

Sensitivity and tradeoff points

We describe key decisions as sensitivity points and tradeoff points. A *sensitivity point* is a property of one or more components (and/or component relationships) that is critical for achieving a particular quality. For example, the level of confidentiality in a virtual private network is sensitive to the level of encryption, or the latency for processing an important message is sensitive to the priority of the lowest priority process involved in handling the message. Sensitivity points tell you where to focus your attention when trying to achieve a quality goal. They serve as yellow flags: *"Use caution when changing this property of the architecture"*. A sensitivity point can be characterized by using the stimuli and architectural parameters branches of taxonomies described in the previous section. For example, the priority of a specific component might be a sensitivity point if it is key property for achieving an important latency goal of the system. A method interface might be a sensitivity point if the flexibility of the interface is key in achieving a certain class of important system extensions.

A *trade-off point* is a property that affects more than one attribute and is a sensitivity point for at least one attribute. For example, changing the level of encryption could have a significant impact on both security and performance. If the processing of a confidential message has a hard real-time requirement, the level of encryption could be a tradeoff point.

It is not uncommon for an architect to answer an elicitation question by saying: "we haven't made that decision yet". In this case you can not point to a component or property in the architecture and call it out as a sensitivity point because the component or property might not exist yet. We have informally referred to these "non-decisions" as *unbounded sensitivity points*. They are unbounded in the sense that the property has not yet been bound to the architecture. The key point here is that it is important to flag key decisions that have been made as well as key decisions that have not yet been made.

Attribute-based architecture styles (ABASs)

An architectural style includes a description of component types and their topology, a description of the pattern of data and control interaction among the components and an informal description of the benefits and drawbacks of using that style. Architectural styles are important since they differentiate classes of designs by offering experiential evidence of how each class has been used along with qualitative reasoning to explain why each class has certain properties.“Use pipes and filters when reuse is desired and performance is not a top priority” is an example of the type of description that is a portion of the definition of the pipe and filter style.

An attribute-based architecture style (ABAS) helps to move the notion of architectural styles toward the ability to reason (whether qualitatively or quantitatively) based on quality attribute-specific models. The goals of having a collection of ABASs are

- to make architectural design more routine and predictable
- to have a standard set of attribute-based analysis questions
- to tighten the link between design and analysis

We define an ABAS as having four parts:

1. *Problem Description*: what problem is being solved by this structure,
2. *Stimuli/Responses*: a characterization of the stimuli that this ABAS is designed to respond to, and a description of the quality attribute-specific measures of the response,
3. *Architectural Style*: the set of component and connector types, the topology, a description of the patterns of data and control interaction among the components (as in the standard definition of an architectural style), and any properties of the components or connectors that are germane to the quality attribute,
4. *Analysis*: a quality attribute specific model that provides a method of reasoning about the behavior of component types that interact in the defined pattern, and

Thus, to further use the pipe-and-filter example, a pipe-and-filter *performance* ABAS would be one that has a description of what it means to be a pipe or a filter and how they would legally be connected, a queuing model of the pipe-and-filter topology together with rules to instantiate the model, and the results of solving the resulting queuing model under varying sets of assumptions. I more detailed example is described in the next section.

The following sections illustrate the four parts of an ABAS that is concerned with achieving performance goals when the architecture comprises multiple processes synchronizing access to a shared resource.

ABAS Part 1: Problem Description

For this ABAS we consider a single processor on which multiple processes reside, each of which perform computations on their own input data stream. Each final output from the system must be produced within a specified time interval after the arrival of an input, after all computations have been performed. We will refer to the input data as a *message*. The requirement then is to completely process each message with a specified bounded end-to-end latency.

This ABAS might be relevant if your problem inherently

- has real-time performance requirements
- consists of multiple processes that share a resource

ABAS Part 2: Stimulus/Response Attribute Measures

We characterize the important stimuli and their measurable, controllable responses as follows:

- **Stimuli**: two or more periodic or sporadic input streams
- **Response**: end-to-end worst-case latency

“End-to-end” refers to a measure beginning at the point of message input, through all stages of computation, to its final output.

ABAS Part 3: Architectural Style

The synchronization style is shown in Figure 4 below, shown in a concurrency view mapped onto a hardware view. In this ABAS there is a single processor and a set of processes with associated known (or estimated) properties (those listed in Table 1) that are transforming input streams into output streams, and some of these processes need to synchronize to share a resource.

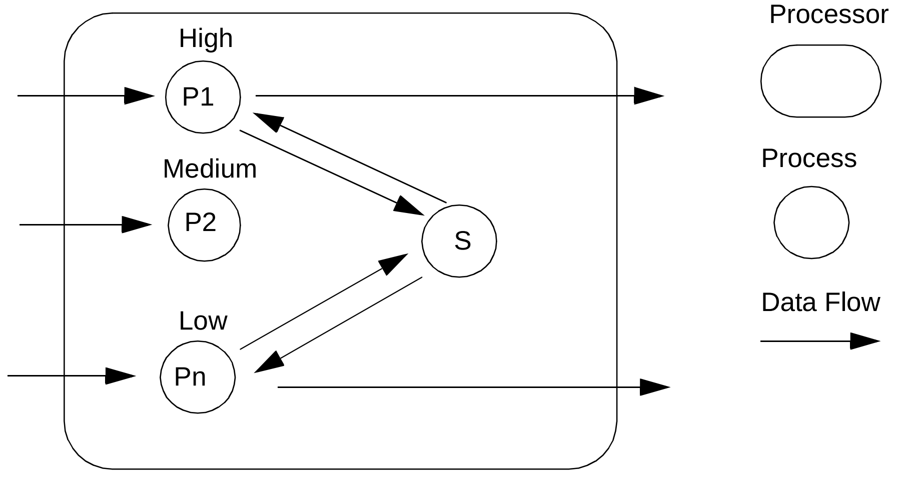

Figure 4: Synchronization ABAS

The architectural parameters of concern for this ABAS are those necessary for creating an analytic model of end-to-end worst-case latency, which include those shown in Table 1.

Performance Architectural Parameters
the shared resource topology
the preemption policy
the execution time for each process associated with processing each input
the priority associated with each process
the scheduling discipline used to schedule the processes (although in the analysis section we will assume a fixed priority scheduling discipline)
the synchronization protocol including: • the queuing discipline (e.g., FIFO or priority) for the server process • how the priority is managed during the critical section (i.e., the section of code during which other processes are locked out)

Table 1: Architectural Parameters for the Synchronization ABAS

ABAS Part 4: Analysis

In this section we present both a formal analysis, showing how to predict the end-to-end worst case latency in this ABAS, given a knowledge of the parameters presented in Table 1, and a set of informal qualitative analysis heuristics, telling the designer what issues to be aware of in creating and assessing any design of a synchronization ABAS.

Modeling the Synchronization ABAS. There are three types of time “experienced” by an arbitrary process under these circumstances: preemption, execution, and blocking time. Let C_i denote the execution time of process i, T_i denote the period of process i, and B_i denote the blocking time incurred by process i. The latency for process i, assuming that processes 1 through i-1 are of higher priority, can be found by iteration using the following formula:

(1)

$$l_{n+1} = \sum_{j=1}^{-1} \left\lceil \frac{l_n}{T_j} \right\rceil C_j + C_i + B_i$$

Given an initial value for l_n of C_i, iterating until l_n equals l_{n+1} results in the worst-case latency for process P_i. If the iterations do not converge or they converge beyond the process’s deadline, this is an indicator or potential timing problems. Equation (1) also illustrates the potential sensitivity of latency to higher priority processes.

Equation (1) is a reflection of the architecture parameters shown in Table 1. The formula is used to calculate the latency for process i. The first term reflects the use of a priority-based preemptive scheduling policy. This term computes the number of times higher priority processes can preempt process i in a window of time that starts at time 0 and extends to time l_n. Each iteration the formula accumulates the execution time associated with each of these preemptions, adds in the execution of process i, C_i, itself and then adds in the blocking time, B_i. The blocking time is directly affected by the synchronization protocol that is used. The next paragraph briefly discusses the effect of priority assignment on latency and the following paragraph discusses the effects of the synchronization protocol on latency.

Priority Assignment. One potential pitfall is the prioritization strategy. It is possible to have very low levels of CPU utilization and still miss deadlines if an inappropriate prioritization strategy is used. Consider the situation in which there are two process with the following characteristics

- Process 1: High priority; execution time is 10 and period is 100
- Process 2: Low priority; execution time is 1, period and deadline are 10

If the two processes are initiated at the same time, process 2 will miss it's deadline and yet the utilization of this set of processes is only 20%

Priority Inversion. This ABAS is a classical situation in which priority inversion and potentially so-called unbounded priority inversion can arise. Consider the following situation. Assume that process P1 has a high priority, P2 has a medium priority and Pn a low priority (and for now the priority of S, the server process, is unspecified). While the low priority process is synchronizing with process S, the high priority process will have to wait if it also needs to synchronize with S. This is priority inversion in the sense that the high priority process is waiting while the low priority process is executing (or strictly speaking while process S is executing on behalf of the low priority process). This could easily happen if the high priority process preempts process S while it is executing at a low priority on behalf of the low priority process. The medium priority process could further exacerbate the situation by preempting the critical section and causing even further delay for the high priority process. This is unbounded priority inversion in the sense that the high priority process could be delayed arbitrarily long by adding other medium priority processes.

This problem illustrates the sensitivity of latency of the high priority process to the priority of the server process. The key to circumventing unbound priority inversion is to ensure that medium priority processes do not have an opportunity to preempt the critical section that is blocking the high priority process. One prevention technique is to set the priority of process S to be at least as high as the high priority process. Another technique is to use a priority inheritance protocol, which raises the priority of S to the highest priority process that is blocked waiting for process S's services.

Even if one does not build a formal analytic model of the latency in a synchronization ABAS, such as we have presented above, a designer should keep in mind that the latency of a process that synchronizes to access shared data is very sensitive to:

- the prioritization strategy including,
- the priority used during the critical section

Utility of ABASs

ABASs are useful for architecture evaluation even if none of the previously catalogued styles seem to be explicitly apparent in the architecture being evaluated. First of all, ABASs serve as examples of the type of analysis framework that should be documented, that is, the explicit connection between patterns in the architecture and predicted behavior. All architectures will have some embedded patterns, since no large system is a random collection of components and connection. The patterns might not always be desirable, but nevertheless there are patterns. Calling out these patterns during an architecture evaluation and explicitly reasoning about their behavior, associating analysis questions with the patterns and as a result identifying sensitivity and tradeoff points is central to the evaluation.

Qualitative analysis heuristics

The qualitative analysis heuristics are meant to be coarse grained versions of the kinds of analyses that we perform when we build precise analytic models of a quality attribute, such as we do when we analyze ABASs. While building quantitative models on the spot during an evaluation is probably not practical, asking these questions is. These questions capture the essence of the typical problems or issues that are discovered by a more rigorous, more formal analysis.

You can see how to construct a typical qualitative analysis heuristic by looking at the ABAS discussed in the previous section. The problem description listed the following criteria for when the ABAS would be applicable: “has real-time performance requirements and consists of multiple processes that share a resource”. The analysis section listed the following rules of thumb: “the latency of a process that synchronizes to access shared data is very sensitive to the prioritization strategy including the priority used during the critical section”. A quality analysis heuristic derived from this ABAS would be:

- If this architecture has real-time performance requirements and consists of multiple processes that share a resource then what is process prioritization strategy is used including the priority used during the critical sections?

Other examples of qualitative analysis heuristics are:

- If this architecture includes layers/facades, are there any places there where the layers/facades are circumvented?
- If this architecture includes a data repository, how many distinct locations in the architecture have direct knowledge of the data types and layout of the repository?
- If a shared data type changes, how many other parts of the architecture are affected?
- If there are multiple processes/threads competing for a shared resource, how are priorities assigned to these processes/threads and the process/thread controlling the shared resource?
- If there exist multiple pipelines of processes/threads, what is the lowest priority for each process/thread in each pipeline.
- If multiple message streams arrive at a shared message queue, what are the rates and distributions of each stream?
- Are there any relatively slow communication channels along an important communication path (e.g. a modem)?
- If redundancy is used in the architecture, what type of redundancy (analytic, replication, functional) and how is the choice made between redundant components?

- How are failures identified? Can active as well as passive failures be identified?
- If redundancy is used in the architecture, how long does it take to switch between instances of a redundant component?

3. Description of ATAM steps

ATAM is typically planned to take three full days where each day consists, in some measure, of

- scenario elicitation
- architecture elicitation
- mapping of scenarios onto the architecture representation, and
- analysis.

In Days 1 and 2 there is more emphasis on the early steps of the method (scenario elicitation, architecture elicitation and scenario mapping) and more emphasis in Days 2 and 3 on the later steps of the method (model building and analysis, sensitivity point identification, tradeoff point identification). Graphically, we see the relationship of the three activities with respect to time as shown in Figure 5:

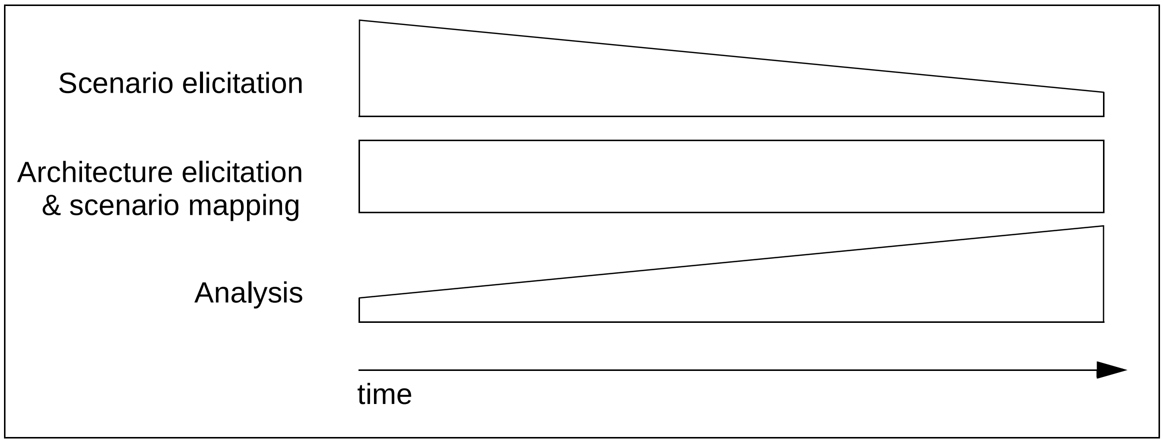

Figure 5: The activities of ATAM and their relative importance over time

where the width of the polygon at each activity shows the amount of anticipated activity within that activity at that time (i.e. on Day 1 we expect relatively little analysis to take place, and on Day 3 we expect most of the activity to be taken up with scenario mapping, and analysis).

The steps in the ATAM are presented below. These steps are divided into three days of activities but the division is not a hard-and-fast one. Sometimes there must be dynamic modifications to the schedule to accommodate the availability of personnel or architectural information.

This is not a waterfall process. There will be times when an analyst will return briefly to an earlier step, or will jump forward to a later step, or will iterate among steps, as the need dictates.

The importance of the steps is to clearly delineate the activities involved in ATAM along with the outputs of these activities. What is more important than the particulars of the schedule is the set of dependencies among the outputs of the steps, as indicated in Figure 6.

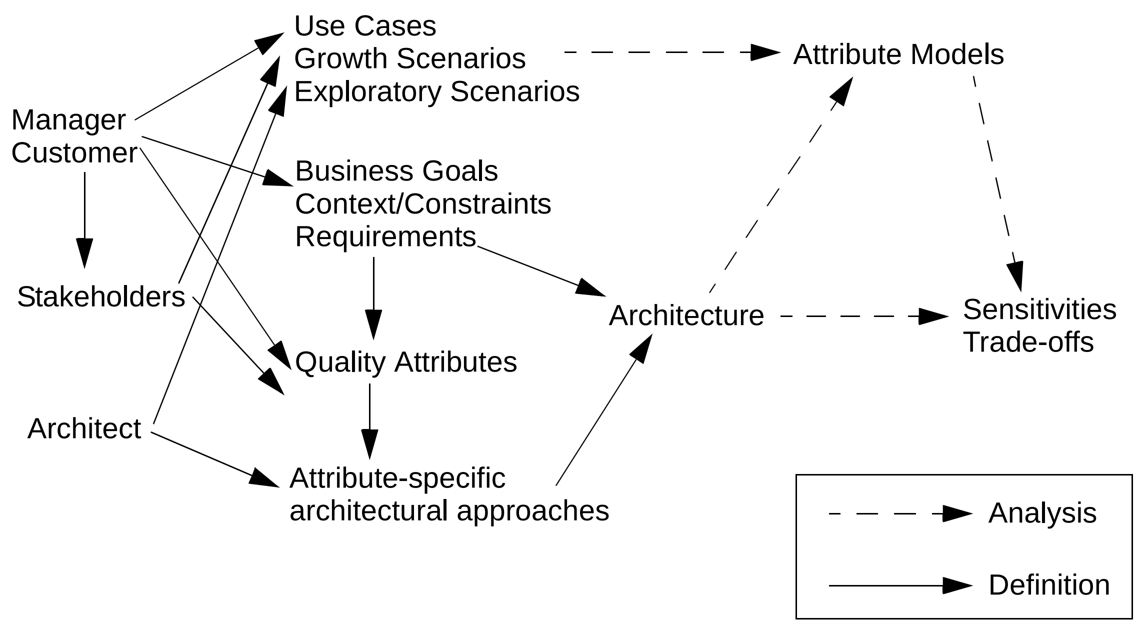

Figure 6: Dependencies among ATAM Outputs

Day 1 Activities

On day 1, the ATAM team meets with the team being evaluated, perhaps for the first time. This meeting has two concerns: organization of the rest of the analysis activities, and information collection. Organizationally, the manager of the team being evaluated needs to make sure that the right people attend the meetings, the people are prepared and that they come with the right attitude (i.e. a spirit of non-adversarial teamwork). The information collection aspect of day 1 is meant to ensure that the architecture is represented in sufficient detail to be evaluated. Also, some initial scenario collection and analysis may be done on day 1, as a way of understanding the architecture, understanding what information needs to be collected and represented, and understanding what it means to generate scenarios.

Step 1 - Lead evaluator presents ATAM. In this step the ATAM is presented to the assembled stakeholders (typically managers, customer representatives, and architects). This sets the context and expectations for the remainder of the activities.

Step 2 - Manager/Customer presents system overview. The system to be evaluated needs to be understood by all participants in the evaluation. This understanding has several facets. The system itself must be presented, initially at a high level of abstraction, typically describing: its major functions, its requirements, its constraints, its business goals, and its context. Also, when describing the system, all of the system's stakeholders need to be identified, so that they can be invited to contribute to later steps of the ATAM.

Step 3 - Architect describes important attribute-specific requirements. The architect needs to explain the driving architectural requirements, in terms of performance, modifiability, security, reliability, and so forth. It is these requirements that shape the architecture, much more than its functional requirements.

Step 4 - Architect presents architecture. The architecture will be presented in as much detail as is currently documented. This is an important step, as the amount of architectural information available and documented will directly affect the analysis that is possible, and its quality. Frequently the evaluator will have to specify additional architectural information that is required to be collected and documented before proceeding to Days 2 and 3 of the ATAM.

Step 5 - Architect presents attribute-specific architectural approaches. For each of the attribute-specific requirements presented in Step 3, the architect will present the architectural approach that is intended to meet the requirement. These approaches will be important to analyze in Days 2 and 3, since they represent the major architectural decisions intended to meet the major quality attribute requirements.

Step 6 - Evaluator elicits "seed" use cases and scenarios. Scenarios are the motor that drives the ATAM. Generating a set of "seed" scenarios (and use cases) has proven to be a great facilitator of discussion and brainstorming during days 2 and 3, when greater numbers of stakeholders are gathered to participate in the ATAM. These serve as models to the stakeholders, by defining the nature and scope of appropriate scenarios for evaluation.

Step 7 - Architect maps seed use cases onto architecture. In this step the architect walks through a selected set of the seed use cases and scenarios, showing how each affects the architecture (e.g. for modifiability) and how the architecture responds to it (e.g. for quality attributes such as performance, security and availability).

Step 8 - Evaluators build initial skeleton analysis. As a result of the presentation of attribute-specific architectural approaches and the mapping of seed scenarios and use cases on to the architecture, the evaluators should now be able to build initial models of each important quality attribute. These will likely not be formal models at this point, but rather will be informal models. For example, informal models will include the known strengths and weaknesses of a particular approach, and sets of questions that help identify these strengths and weaknesses. Such models will guide the evaluators in the later stages of the ATAM in probing for more precise and detailed information.

Step 9 - Evaluators determine action items. A number of needs have typically become evident at this point in the evaluation. For example, as stated above, the architectural documentation is often insufficient to support a complete evaluation and this needs to be remedied as an action item. A set of stakeholders needs to be contacted and meeting dates for days 2 and 3 need to be determined. In addition, any action item may become a go/no-go decision for the continuation of the ATAM.

Day 2 Activities

In day 2, the main analysis activities begin. At this point, it is assumed that the architecture has been documented in sufficient detail to support analysis, the appropriate stakeholders have been gathered and have been given advanced reading materials so that they know what to expect from the ATAM, and that the seed scenarios have been collected and distributed to the stakeholders.

Step 1 - Lead evaluator presents ATAM. Since there will be a different set of stakeholders attending day 2, and since a number of days or weeks may have transpired between days 1 and 2, it is useful to recap the steps of the ATAM, so that all attendees have the same understanding and expectations of the day's activities.

Step 2 - Evaluators brainstorm scenarios/use cases with stakeholders. The stakeholders now undertake two related activities: brainstorming *use cases* (representing the ways in which the stakeholders expect the system to be used) and scenarios (representing the ways in which the stakeholders expect the system to change in the future). The scenarios are further subdivided into two categories: *growth scenarios* and *exploratory scenarios*. Growth scenarios represent ways in which the architecture is expected to accommodate growth and change: expected modifications, changes in performance or availability, porting to other platforms, integration with other software, and so forth. Exploratory scenarios, on the other hand, represent extreme forms of growth: ways in which the architecture might be stressed by changes: dramatic new performance or availability requirements (order of magnitude changes, for example), major changes in the infrastructure or mission of the system, and so forth. Growth scenarios are a way of showing the strengths and weaknesses of the architecture with respect to anticipated forces on the system. Exploratory scenarios are an attempt to find sensitivity points and tradeoff points. The identification of these points help us assess the limits of the system with respect to the models of quality attributes that we build.

The seed scenarios created in Day 1 help facilitate this step, by providing stakeholders with examples of relevant scenarios.

Step 3 - Evaluators prioritize scenarios/use cases with stakeholders. Once the use cases and scenarios have been collected, they must be prioritized. We prioritize each category of use case/scenario separately. We typically do this via a voting procedure where each stakeholder is allocated a number of votes equal to 30% of the number of scenarios within the category, rounded up. So, for instance, if there were 18 use cases collected, each stakeholder would be given 6 votes. These votes can be allocated in any way that the stakeholder sees fit: all 6 votes allocated to 1 scenario, 2 votes to each of 3 scenarios, 1 vote to each of 6 scenarios, etc. This can be an open or a secret balloting procedure (a particular method is described and recommended in Chapter 6). Once the votes have been made, they are tallied and the use cases and scenarios are prioritized by category. A cutoff is typically made that separates the high priority use cases from the lower ones, and only the high priority ones are considered in future evaluation steps. For example, a team might only consider the top 5 use cases.

Step 4 - Architect maps important use cases onto architecture. The architect, considering each high priority use case in turn, maps the use cases onto the architecture. In mapping these use cases, the architect will walk through the actions that the use case initiates, highlighting the areas of the architecture that participate in realizing the use case. This information can be documented as a set of components and connectors, but it should also be documented with a pictorial representation that shows the flow of responsibility among the components and connectors. The choice of which documentation vehicle to use depends in part on the granularity of information required, the perceived risk implied by the scenario, and the stage of development of the architecture. For a mature project and/or high risk project it may be possible and important to trace through behavior diagrams. For a less mature or less risky project use case maps might be appropriate. The evaluators can use this walkthrough of the use case to ask attribute-specific questions (such as performance or reliability questions) and to annotate the architectural representation. In doing so, the evaluators will record a set of issues, sensitivity points, and tradeoff points.

Step 5 - Architect maps important growth scenarios onto architecture. As with step 4, the architect will consider each high priority growth scenario in turn and will map each of these onto the architecture. Since growth scenarios imply a change to the system, the architect will identify all changed components and connectors, and the evaluators will record this information using standard documentation templates. In addition to understanding and documenting the modifications to the system, the evaluators will probe for the issues, sensitivities, and tradeoffs uncovered by this scenario as they affect quality attributes such as performance, security, and reliability.

Step 6 - Evaluators build skeleton analyses of each quality attribute. Using screening questions and qualitative analysis heuristics, the evaluators are now in a position to begin building more complete models of each quality attribute under scrutiny. Building a model of even a single quality attribute can be a daunting task in a complex system; there is simply too much information to assimilate. Our solution to this information overload problem is to use screening questions and qualitative analysis heuristics. The screening questions serve to limit the portion of the architecture under scrutiny. The qualitative analysis heuristics suggest questions for the evaluator to ask of the architect that help in identifying common architectural problems. If these questions do uncover potential problems then the analyst can begin to build a more comprehensive model of the quality attribute aspect under scrutiny.

Day 3 Activities

Step 1 - Evaluators brainstorm exploratory scenarios with selected stakeholders. As was done on day 2, day 3 begins with the brainstorming of exploratory scenarios. The difference here is that the stakeholder group on the third day has now been pared down to just the lead architects and key developers. These scenarios, as with all scenario brainstorming, are examined in light of the quality attributes that they cover (to ensure that all important attributes are represented fairly) and the stakeholders whose concerns they address.

Step 2 - Prioritization of exploratory scenarios. As was done on day 2, we then prioritize the newly captured exploratory scenarios. This prioritization can proceed exactly as it did on day 2 (if the group of stakeholders is of moderate size) or can be done informally if the stakeholder group is small (say, fewer than 6).

Step 3 - Mapping of important exploratory scenarios onto relevant architectural views. Those scenarios that are of highest priority are now mapping onto the relevant architectural view or views, as on day 2. Once again, attribute specific questions will be asked during this mapping process and the answers will be recorded in the appropriate documentation template.

Step 4 - Analysis building using screening questions and qualitative analysis heuristics. The skeleton analyses that were created on day 2 are further fleshed out during this stage. Once again, we need to use the screening questions and qualitative analysis heuristics to limit the scope of the investigation, but, building on the models from day 2, we can begin to explore areas that appear to be potential locales of risks, sensitivities, and global trade-offs.

Step 5 - Debriefing. By the end of three days of analysis, the analysts have amassed a substantial amount of information on the system under scrutiny. The analysts have also begun to form models of the important quality attributes

4. An Example Evaluation: The BCS

We now present an example of the ATAM as it was realized in an evaluation. Although some of the details have been changed to protect the identity and intellectual property of the customer and contractor, the architectural issues that we uncovered are not materially affected by these changes.

We will call this system BCS (Battlefield Control System). This system is to be used by Army battalions to control the movement, strategy, and operations of troops in real time in the battlefield. This system is currently being built by a contractor, based upon government furnished requirements. These requirements state that the system supports a Commander who commands a set of Soldiers and equipment, including many different kinds of weapons and sensors. The system needs to interface with numerous other systems that feed it commands and intelligence, and collect its status with respect to its missions.

In describing the BCS ATAM, we will not describe every step of the method due to space considerations. Table 2 shows a few of the 40 growth and exploratory scenarios that were elicited in the second day of the ATAM evaluation[2].

Table 2: Sample Scenarios for the BCS Evaluation

Scenario	Scenario Description
1	Same information presented to user, but different presentation (location, fonts, sizes, colors, etc.).
2	Additional data requested to be presented to user.
3	User requests a change of dialog.
4	An new device is added to the network, e.g. a location device that returns accurate GPS data.
5	An existing device adds additional fields that are not currently handled to existing messages.
6	Map data format changes.
7	The time budget for initialization is reduced from 5 minutes to 90 seconds.
8	Modem baud rate is increased by a factor of 4.
9	Operating system changes to Solaris.
10	Operating schedule is unpredictable.

Table 2: Sample Scenarios for the BCS Evaluation

Scenari o	Scenario Description
11	Can a new schedule be accommodated by the OS?
12	Change the number of Soldier nodes from 25 to 35.
13	Change the number of simultaneous missions from 3 to 6.
14	A node converts from being a Soldier/client to become a Commander/server.
15	Incoming message format changes.

As mentioned above, the architectural documentation covered several different structures and views of the system: a dynamic view, showing how subsystems communicated; a set of message sequence charts, showing run-time interactions; a system view, showing how software was allocated to hardware; and a source view, showing how components and subsystems were composed of objects. For the purpose of this example, we will just show a coarse hardware structure of the architecture, using the notation from [BCK98].[3]

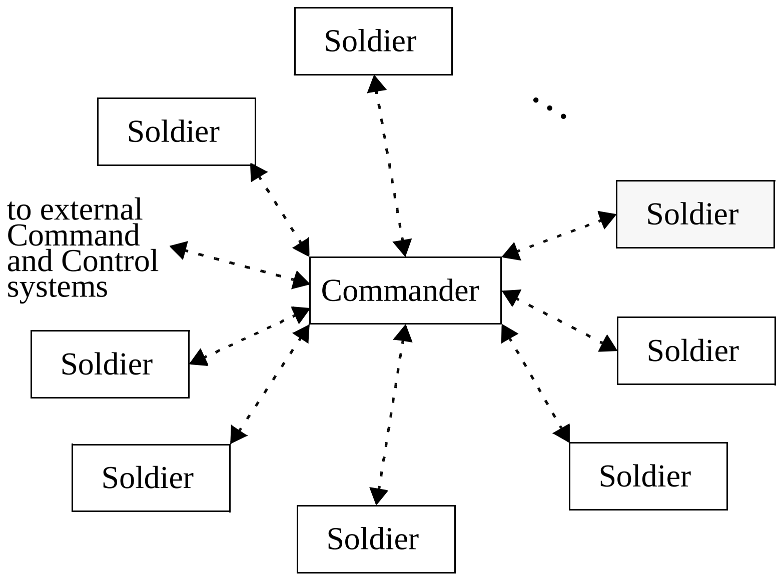

Figure 7: System Architecture of the BCS

This hardware architecture, shown in Figure 7, illustrates that the Commander is central to the system. In fact, the Commander node acts as a server and the Soldier nodes are its clients, making requests of it and updating the server's database with their status. Interaction between the client and server is only through encrypted messages sent via a radio modem; neither sub-

2. These scenarios have been cleansed of proprietary specifics, but their spirit is true to the original.
3. In this notation, rectangles represent processors and dashed lines represent data flow.

system controls the other. Note also that the radio modem is a shared communication channel: only one node can be broadcasting at any moment.

For the BCS, each of the use cases and growth scenarios were mapped onto the appropriate architectural view. For example, when a scenario implied a modification to the architecture, the ramifications of the change were mapped onto the source view, and scenario interactions were identified as sensitivity points. For availability and performance, use cases describing execution and failure of the system were mapped onto run-time and system views of the architecture, and models were built of latency and availability based upon the information elicited by these mappings.

At this stage in the analysis we had a set of architectural documentation but little insight into the way that the architecture worked to accomplish the BCS's mission. So we used a set of screening questions and qualitative analysis heuristics to flesh out our understanding of the architecture. These questions probed both the normal operation of the system and its boundary conditions. For example:

- For what functions of the system is performance *not* important?
- For those functions for which performance is *not* important what is the consequence of extremely long response times or extremely low throughput?
- How is performance impacted by scaling up the workload?
- If there are multiple processes/threads competing for a shared resource, how are priorities assigned to these processes/threads and the process/thread controlling the shared resource?
- If this architecture includes layers/facades, are there any places there where the layers/facades are circumvented?

By itself, Figure 7 tells us little about the system. However, when illuminated by a small number of use cases, growth scenarios, and qualitative analysis heuristics, this view became the focus for availability (or, in the customer's terms, survivability) and performance analyses.

Our next step in the ATAM was to select additional growth and exploratory scenarios for more detailed consideration and to aid in the creation of analysis models.

For the BCS system we realized through the qualitative analysis heuristics that three quality attributes were the major architectural drivers for overall system quality: availability, modifiability, and performance. Hence we can say that system quality (Q_S) is a function f of the quality of the modifiability (Q_M), the availability (Q_A), and the performance (Q_P):

$$Q_S = f(Q_M, Q_A, Q_P)$$

The next section will describe the analysis that we created for performance; the analyses for availability and modifiability proceeded in analogous fashion and are omitted for brevity. Each of these analyses was created by first employing the qualitative analysis heuristics to elicit information about the quality attribute in question, and then building more formal analytic models based upon the information elicited.

Performance

By building a simple performance model of the system and varying the input parameters to the model (the various processing times and communication latencies), it became clear that the slow speed of radio modem communication between the Commander and the Soldiers (9600 baud) was the single important performance driver for the BCS. The performance measure of interest—average latency of client-server communications—was found to be insensitive to all other architectural performance parameters (e.g. the time for the system to update its database, or to send a message internally to a process, or to do targeting calculations). But the choice of modem speed was given as a constraint, and so our performance model was focused on capturing those architectural parameters that affected message sizes and distributions.

We begin by identifying the scenarios, from among all those considered, that have performance implications and the communication requirements implied in each scenario. For example, we considered three scenario groups (not all of which appear in Table 2) when building our performance model:

- Scenarios 14, 23, 25, 29 (turning a Soldier node into a backup): a switchover requires that the backup acquires information about all missions, updates to the environmental database, issued orders, current Soldier locations and status, and detailed inventories from the Soldiers.
- Scenarios 26, 27 (regular, periodic data updates to the Commander): various message sizes and frequencies.
- Scenarios 13, 20: Increasing number of weapons from 30 to 60 or missions from 3 to 6.

We created performance models of each of these scenario groups. For the purposes of illustration in this example, we will only present the performance calculations for scenario group A), the conversion from Soldier backup to Commander.

After determining that a switchover is to take place the Soldier backup will need to download the current mission plans and environmental database from the external command and control systems. In addition, the backup needs the current locations and status of all of the remaining Soldiers, inventory status from the Soldiers, and the complete set of issued orders.

A typical calculation of the performance implications of this scenario group will take into account the various message sizes needed to realize the scenario, the 9600 baud modem rate (which we equate to 9600 bits/second), and the fact that there are a maximum of 25 Soldiers per Commander (but since one is now being used as a Commander, the number of Soldier nodes in these calculations is 24):

Downloading mission plans:
280 Kbits / 9.6 Kbits/second $\cong$ 29.17 seconds.

Updates to environmental database:
66 Kbits / 9.6 Kbits/second $\cong$ 6.88 seconds.

Acquiring issued orders:
24 Soldiers * (18 Kbits/9.6 Kbits/second) = 45.0 seconds.

Acquiring Soldier locations and status:
24 Soldiers * (12 Kbits/9.6 Kbits/second) = 30.0 seconds.

Acquiring inventories:
24 Soldiers * (42 Kbits/9.6 Kbits/second) = 105.0 seconds.

Total $\cong$ 216.05 seconds for Soldier to become backup

Note that, since the radio modem is a shared communication channel, no other communication can take place while a Soldier/backup is being converted to a Commander.

There was no explicit requirement placed on the time to switch from a Commander to a backup. However, there was an initialization requirement of 300 seconds which we will use in lieu of an explicit switchover time budget. If we assume that the 280K bits in the mission plan file contains the 3 missions in the current configuration, then doubling the number of missions (scenario 13) would imply doubling the mission message from 280K bits to 560K bits and the transmission time would increase by almost 30 seconds, still meeting the time budget. If, on the other hand, the number of Soldiers increases to 35 (scenario 12), the total time will increase by about 90 seconds, which would not meet the time budget.

Keeping each backup in a state of high readiness requires that they become acknowledging backups, or for a lower state of readiness they can be kept as passive backups. Both classes of backups require periodic updates from the Commander. From an analysis of scenario group B), we have calculated that these messages average 59,800 Kbits every 10 minutes. Thus, to keep each backup apprised of the state of the Commander requires 99.67 bits/second, or approximately 1% of the system's overall communication bandwidth. Acknowledgments and resends for lost packets would add to this overhead. Given this insight, we can characterize the system's performance sensitivities as follows:

$$Q_P = h(n, m, CO)$$

That is, the system is sensitive to the number of acknowledging backups (*n*), passive backups (*m*), and other communication overhead (*CO*). The main point of this simple analysis is to realize that the size and number of messages to be transmitted over the 9600 baud radio modem is important with respect to system performance and hence availability. Small changes in message sizes, or frequencies can cause significant changes to the overall throughput of the system. These changes in message sizes may come from changes imposed upon the system

Global Tradeoff Identification

As a result of these analyses we identified three sensitivities in the BCS system, and two of these are affected by the same architectural parameter: the amount of message traffic that passes over the shared communication channel employed by the radio modems, as described by some functions of *n* and *m*, the numbers of acknowledging and passive backups. Availability and performance were characterized respectively as:

$$Q_A = g(n, m)$$

and

$$Q_P = h(n, m, CO)$$

These two parameters control the tradeoff point between the overall performance of the system, in terms of the latency over its critical communication resource, and between the availability of the system in terms of the number of backups to the Commander, the way that the state of those backups is maintained, and the negotiation that a backup needs to do to convert to a Commander. To determine the criticality of the tradeoff more precisely, we can prototype or estimate the currently anticipated message traffic and the anticipated increase in message traffic due to acknowledgments of communications to the backups. In addition, we would need to estimate the lag for the switchover from Soldier to Commander introduced by not having acknowledged communication to the Solder backup nodes. Finally, all of this increased communication needs to be considered in light of the performance scalability of the system (since communication bandwidth is the limiting factor here).

One way to mitigate against the communication bandwidth limitation is to plan for new modem hardware with increased communication speeds. Presumably this means introducing some form of indirection into the modem communications software—such as an abstraction layer for the communications—if this does not already exist. This possibility was not probed during the evaluation.

While this tradeoff might seem obvious, given the presentation here, it was not so. The contractor was not aware of the performance and availability implications of the architectural decisions that had been made. In fact, in our initial pre-meeting, not a single performance or availability scenario was generated by the contractor; these simply were not among their concerns. The contractor was worried about the modifiability of the system, in terms of the many changes in message formats that they expected to withstand over the BCS's lifetime. However, the identified tradeoff affected the very viability of the system. If this tradeoff was not carefully reasoned about, it would affect the system's ability to meet its most fundamental requirements.

In addition to the sensitivities and tradeoffs, in building the models of the BCS's availability and performance, we discovered a serious architectural weakness that had not been previously identified: there exists the possibility of an opposing force identifying the distinctive communication pattern between the Commander and the backup and thus targeting those nodes specifically. The Commander and backup exchange far more data than any other two nodes in the system. This identification can be easily done by an attacker who could discern the distinctive pattern of communication between the Commander and (single) backup, even without being able to decrypt the actual contents of the messages. Thus, it must be assumed that the probability of failure for the Commander and its backup increases over the duration of a mission under the existing BCS architecture. This was a major architectural flaw that was only revealed *because* we were examining the architecture from the perspective of multiple quality attributes simultaneously. This flaw is, however, easily mitigated by assigning multiple backups, which would eliminate the distinctive communication pattern.

5. Conclusions and Future Work

ATAM is a young method and is undergoing refinement and improvement, but has already been applied for paying customers to help them evaluate large, complex real-world architectures. More ATAM exercises are scheduled in the near future. To help make the method repeatable, we are building repositories of scenarios, screening and elicitation questions, ABASs, and qualitative analysis heuristics. This information, as well as the more mundane details such sample agendas, method overview slides, final report templates, supply lists, and moderator experience reports, are being assembled into a handbook for evaluation teams. The goal of the handbook is to serve as the medium in which evaluation experience is captured and stored for new evaluators.

To date, ATAM has shown its value in evaluating an architecture for suitability in terms of imbuing a system with the following quality attributes:

- Performance
- Reliability and availability
- Security
- Modifiability
- Functionality
- Variability (suitability to serve as the architecture for a product family)
- Subsetability (suitability to support the production of a subset of the system, or to incrementally develop a system).

At the beginning of this paper, we opined that the reason architecture evaluation was not a standard part of organization's development processes was that there was not yet a practical, repeatable method available. We believe that ATAM fills that need.

6. References

1. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., *Pattern-Oriented Software Architecture*, Wiley, 1996.
2. Iannino, A., "Software Reliability Theory", *Encyclopedia of Software Engineering,* Wiley, 1237-1253.
3. Kazman, R., Abowd, G., Bass, L., Clements, P., "Scenario-Based Analysis of Software Architecture", *IEEE Software*, Nov. 1996, 47-55.
4. Klein, M., Ralya, T., Pollak, B., Obenza, R., Gonzales Harbour, M., *A Practitioner's Handbook for Real-Time Analysis*, Kluwer Academic, 1993.
5. Bass, Clements, Kazman: Software Architecture in Practice. Addison-Wesley. 1998.
6. Avritzer, Alberto, and Weyuker, Elaine; "Investigating Metrics for Architectural Assessment," *Proceedings, Fifth International Symposium on Software Metrics*, Bethesda, MD, November 1998.