Attribute-Based Architecture Styles

Mark H. Klein, Rick Kazman, Len Bass, Jeromy Carriere, Mario Barbacci, and Howard Lipson
Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213
{mk, rkazman, ljb, sjc, mrb, hfl}@sei.cmu.edu

Key words: Architecture, architecture styles, quality attributes

Abstract: Architectural styles have enjoyed widespread popularity in the past few years, and for good reason: they represent the distilled wisdom of many experienced architects and guide less experienced architects in designing their architectures. However, architectural styles employ qualitative reasoning to motivate when and under what conditions they should be used. In this paper we present the concept of an ABAS (Attribute-Based Architectural Style) which includes a set of components and connectors along with their topology, but which adds to this a quality attribute specific model that provides a method of reasoning about the behavior of component types that interact in the defined pattern. We will define ABASs in this paper, show how they are used, and argue for why this extension to the notion of architectural style is an important step toward creating a true engineering discipline of architectural design.

1. INTRODUCTION

An architectural style (as defined by Shaw and Garlan (Shaw and Garlan, 1996) and elaborated on by others (Buschmann, et al., 1996)) 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

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35563-4_35

P. Donohoe (ed.), *Software Architecture*

reasoning to explain why each class has certain properties."Use the pipe and filter style 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. The purpose of this paper is to move the notion of architectural styles toward having the reasoning (whether qualitative or quantitative) based on quality attribute-specific models. We call these enhanced architectural styles, Attribute-Based Architecture Styles (ABASs) and we view them as the next generation in the development of architectural styles.

We define an ABAS as a triple

1. the topology of component types and a description of the pattern of data and control interaction among the components (as in the standard definition),
2. a quality attribute specific model that provides a method of reasoning about the behavior of component types that interact in the defined pattern, and
3. the reasoning that results from applying the attribute specific model to the interacting component types.

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.

Software architecture styles are useful during both design and analysis. Styles are useful during design because the software architect can choose a style based on an understanding of the desired quality goals of the system under construction. The goal of those cataloguing architectural styles (Buschmann, et al., 1996) is to provide a handbook that the software architect can use as a reference to have design options with known qualities from which to choose.

In this paper, we make two points. The first (rather obvious) point is that architectural styles are also useful in analysis. When analyzing a system, the recognition of the use of pipe and filter, for example, leads to questions about how performance is handled and about the assumptions that the filters make that might impact their reuse. The second point (somewhat less obvious) is that when considering architectural styles as analysis tools, focussing on particular quality attributes (McCall, 1994) leads to the ability to attach known analytic models for these attributes to the architecture being analyzed. This in turn leads to the ability to *predict* the effect of particular architectural decisions and changes to the architecture. Thus, instead of the designer having vague guidance about a particular style's effect on performance, the designer is given a model, its analysis, and its explicit connection to aspects of the architectural style so that the designer can answer questions such as "What is the effect on performance of moving a particular piece of functionality from one component to another within a pipe and file based architectural design?"

In the remainder of this paper, we discuss the roots of the ABAS concept, the pieces of an ABAS, the types of attribute models that exist and how they would be used in constructing an ABAS, an extended pedagogical example, and an example drawn from our experience using ABASs in architectural analysis that shows how ABASs work in practice.

2. MOTIVATIONS

The motivation for ABASs comes from three different sources:

1. architectural styles, such as those catalogued by Shaw and Garlan in (Shaw and Garlan, 1996) and by Buschmann et al in (Buschmann, et al., 1996)
2. analytic models of quality attributes, such as rate monotonic analysis for performance (Klein, et al., 1993) or Markov models for availability
3. architecture evaluation questionnaires, such as those used by AT&T (Maranzano, 1993)

ABASs are a kind of architectural style, and hence they build squarely upon the foundational work of Shaw and Garlan, as well as the similar work of the design patterns community (Gamma, et al., 1994). However, in each of these cases, the kinds of reasoning that the architectural styles support is heuristic. For example, in describing the layered style, Shaw and Garlan write "if a system can logically be structured in layers, considerations of performance may require closer coupling between logically high-level functions and their lower-level implementations". While this is important information for the designer who is considering using this style, it does not give the designer a principled way of understanding when a specific number and organization of layers will cause a performance problem. The answer to this dilemma lies in our second influence, analytic models of quality attributes.

Mature analytic models exist for several quality attributes that are of central concern to complex software systems, such as performance, reliability and, to a lesser extent, security. These models not only provide a way to establish a more precise understanding of, for example, "considerations of performance", but also can allow the analyst to associate particular measurable performance criteria with architectural choices. This gives the designer a way to rigorously experiment with, and plan for, architectural quality requirements.

However, analytic models are typically quite general and it requires a substantial amount of training to be able to use them effectively. Frequently, when we perform architectural evaluations, we need to be able to assess a design's effectiveness and risk within a 2 or 3 day period. This leads to the our third motivation. A proven technique for aiding in risk assessments of architectures, first used widely by AT&T's software architecture validation exercises (Maranzano, 1993), is a questionnaire or checklist. A proven set of questions can help organize the line of reasoning and investigation into an architecture, and can provide a first insight into problem areas. These questions can be a first approximation for an analysis, and can lead the analyst in probing the architecture.

3. MODELING ARCHITECTURAL DECISIONS USING AN ABAS

One of the reasons for focusing attention on an architecture is to highlight and analyze critical early design decisions. Translating these decisions into some modelling framework that supports predictive reasoning at the architecture level is key for attaining the potential benefits of focusing on the architecture. The structure of an ABAS reflects this goal of *mapping* an architecture style onto an attribute-modelling framework. This notion is represented by Figure 1.

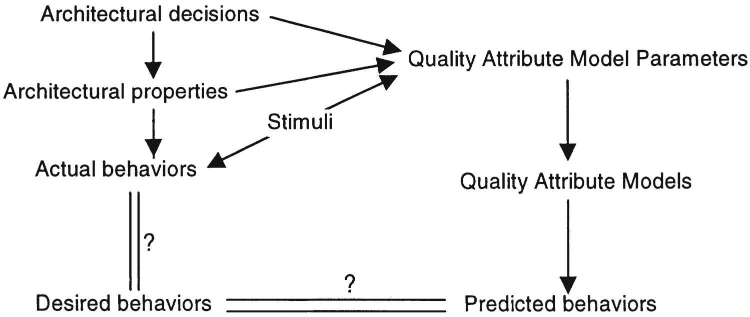

Figure 1. Mapping architectural models to attribute models

The left side of the figure says that architectural decisions directly and/or indirectly affect the behavior ultimately manifested by the architecture. That is obvious, but what is less clear is how to characterize those behaviors and to understand how they compare with the desired behaviors. For example, allocating functionality to a collection of processes (a subset of the architectural decisions that a designer will make) that are in turn allocated to processors (more architectural decisions) result in a set of process execution times (that is, architectural properties). Architectural properties in conjunction with stimuli such as message arrival rates ultimately lead to the performance behavior that will be exhibited. The actual behavior of the system is unknowable without constructing the system and so we use models of the behavior as a method of characterizing the actual behaviors.

We can, of course, “hope” that the actual behavior will satisfy the desired behavior, but there is no way to know unless some type of model is used. The architecture abstraction needs to be mapped to some other abstraction that is more supportive of reasoning. For example, if the goal is to reason about reliability, the salient features of the architecture (such as redundancy) need to be mapped onto reliability models (such as Markov models). At this point the behaviors predicted by the models can be compared with the desired behaviors. Such reasoning can become the basis for comparing architectures and for making decisions regarding the form of the final software architecture.

3.1 The Structure of an ABAS

We define an ABAS to have five parts:

1. **Problem description** - describes the design problem that the ABAS is intended to solve, including the quality attribute of interest, the context of use, constraints, and relevant attribute-specific requirements.
2. **Quality attribute measures** - a condensation of what was discussed in the problem description, but in specific terms pertinent to the *measurable aspects* of the quality attribute model. This includes a discussion of *stimuli*: events that cause the architecture to respond or change.
3. **Architectural style** - a description of the architectural style in terms of component, connections, properties of the components and connections, and patterns of data and control interactions.
4. **Quality attribute parameters** - a condensation of what was discussed in the architectural style section but in specific terms relevant to the *parameters* of the quality attribute model.
5. **Analysis** - a description of how the quality attribute models are *formally related* to the elements of the architectural style and the conclusions about architectural behavior that are drawn via the models.

Note that these parts rely on a description of the architectural style and on a description of a quality attribute. Describing quality attributes is discussed in the next section.

4. QUALITY ATTRIBUTE MODELS PARAMETERS

To assess an architecture for adherence to quality requirements, those requirements need to be expressed in terms that are measurable or at least observable. We call these the *quality attribute measures*. These parameters depend on properties of the architecture, called *quality attribute parameters*, and on the *stimuli*. The quality attribute parameters are the adjustable parameters of the architecture that determine whether the dependent parameters will satisfy the quality requirements. Stimuli are the events to which the architecture will have to respond.

Consider performance: performance is concerned with timeliness, usually measured as either latency or throughput. Therefore two **quality attribute measures** are:

- **Latency** - time from the occurrence of an event until the response to that event is complete; expressed in units of time.
- **Throughput** - the rate at which the system can respond to events; expressed in terms of transactions (or responses) per unit time.

The **stimuli** are changes of state to which the architecture must respond. For performance the arrival pattern of an event is important. It can be one of:

- **Periodic** - there is a fixed interval between event arrivals.
- **Sporadic** - there is a bound on how short the interval between arrivals can be.
- **Stochastic** - arrivals can be described probabilistically.

When a stimulus occurs, the system responds to it by using its resources to carry out computations or transmit data. Multiple concurrent stimulus responses require an arbitration or scheduling policy to resolve conflicting requests. Thus we think of performance-related architectural parameters in terms of the resources that are needed, the policies for allocating resources, and properties of how the resources are requested and used. Therefore **quality attribute parameters** include:

- **Resource characteristics** - include the type of resource such as CPU or network and characteristics such as processor speed or network bandwidth.
- **Resource scheduling policy** - includes CPU scheduling, CPU allocation, and bus and network arbitration; and queuing policies.
- **Resource usage** - includes the priority of processes and messages, preemptability of response and magnitude of use such as execution time.

ABASs map a characterization of architectural properties onto quality attribute parameters, and then map (via modelling) quality attribute parameters and stimuli onto predicted behaviors. Models such as those for scheduling and queuing provide the basis for relating quality attribute parameters (such as queuing policies and execution time estimates) to quality attribute measures (such as latency and throughput). Some parameters such as execution time might not be easily quantifiable at the architecture level. In this case execution time budgets can be assigned, which then become derived requirements for fleshing out the details of the component. This is further illustrated in the next section in which we discuss a sample ABAS for reliability.

For other attributes such as *reusability* or *modifiability,* where there are no universal quality attribute measures, scenarios can used to provide context dependent measures. (Kazman, et al., 1996)

5. ABASs

We will illustrate the notion of an ABAS by an example. We are currently collecting, documenting, and testing many such examples in the hope of creating an engineering handbook of ABASs. The example given here uses a form of redundancy known as analytic redundancy as a means of achieving high levels of availability. First, we will lay out a portion of the relevant attribute model[1].

5.1 Reliability/Availability Attribute Model

Reliability is usually measured in terms of mean time to failure (MTTF). Availability is usually measured in terms of the long-run fraction of time that a system is working. Component failures (and faults)[2], and repair (or recovery) are the stimuli of concern. Architecture parameters include fault detection and fault containment and recovery strategies. An attribute model for reliability/availability looks as follows[3]:

Quality attribute measures

- **Steady state availability** — fraction of time that the system is working (that is, not in a failed state)
- **Reliability** — usually measured in terms of mean time to failure
- **Faults detectable** — passive failures (detectable via time-out mechanisms), active failures, timing failures, semantic failures

[1] An attribute model does not have to be developed for every ABAS. Only one attribute model is needed per attribute and it is then applied to all ABASs for which that specific attribute is relevant.

[2] In this paper we do not distinguish between failures and faults.

[3] This is not meant to be a complete attribute model, but rather one that focuses attention on architectural decisions.

Stimuli -- characterized in terms of the types of failures and repairs and their rates.

Quality attribute parameters

- **Detection** - mechanisms for detection of failures such as voting, post-condition checking, and deadline detection
- **Recovery** - mechanisms for recovering from failure including forward and backward recovery mechanism
- **Modes** - A system can operate in various degraded modes and the availability/reliability of each mode of operation have to be calculated separately.

5.2 Simplex ABAS

We will now describe the documentation that accompanies an ABAS by means of an example of a particular ABAS, called Simplex.

5.2.1 Problem Description

The purpose of this section is to describe the architectural design problem being addressed or in other words the goals of the architecture.[4]

The Simplex (Sha, et al.) ABAS focuses on the problem of software reliability in control systems. In particular, Simplex addresses the problem of tolerating software faults introduced as a consequence of upgrading control algorithms. Simplex also addresses the problem how to take advantage of redundancy to increase reliability while avoiding "common mode" software failures.

To illustrate the problem consider, "the update paradox", as described in (Sha, et al., 1996). Consider the case in which a component is replicated to ensure its reliability. Each replica performs its calculations and sends its results to a voter. If the results do not agree (to within a specified tolerance), the voter "votes for the majority".

Let's say that a key algorithm is updated which will yield a different output value than the older algorithm. Here's the paradox: if the new algorithm is placed in a minority of the replicated components then it will be voted out and have no effect; if it's placed into a majority of the replicated components and is faulty, the bad output will used.

There are two problems highlighted by the upgrade paradox. First of all, even components that have been implemented by different groups and hence have different implementations can suffer from common mode failures. Hence the first problem is, how to introduce redundancy to ensure the proper

[4] The text in italics in this section is commentary for the reader, and is not part of the ABAS.

level of reliability/availability without introducing common mode failures? The second problem is, how do you upgrade a system without compromising its reliability/availability?

5.2.2 Quality Attribute measures and Stimuli

Based on the desired architectural behavior, the stimuli of the reliability/availability attribute model, and the problem description there are set of specific issues that should be highlighted. These issues are raised in this section. A checklist of such issues would include those that follow.

The availability/reliability issues of concern for this ABAS are:

- What types of faults need to be tolerated by the architecture
- What the levels of (degraded) service are
- What the reliability/availability is for each level of service

Types of faults: the goal of this architecture is to handle timing faults (e.g., timing overruns), semantic faults (wrong output values) and system faults (such as memory overruns due to bad pointers).

Reliability of service levels: There is a specified desired level of availability for the upgraded or higher performance level of service and specified level of reliability for the baseline level of service.

5.2.3 Architectural Style

This section starts by identifying the relationship between this ABAS and other similar ABASs. In this case the Simplex ABAS is an instance of a more general pattern.

Simplex is an architectural style which belongs to a general family of reliability styles that could be called redundancy styles. The general pattern for a redundancy style is shown in Figure 2 below. The pattern, from a reliability point of view, consists of multiple redundant components. Data flows into one or more redundant components, which then send their output to another component (or possibly components) responsible for detecting failures, switching to a working component and possibly initiating recovery of the failed component.

The Simplex style, as shown in Figure 3, is an instance of the redundancy style in which the redundant components are processes. The components don't necessarily receive the same input or generate the same output. Moreover, the components are not all peers. The components are *analytically redundant*, meaning they are redundant with respect to the general effect their output has in controlling their environment, but not necessarily redundant in the algorithms used or the output produced. [5]

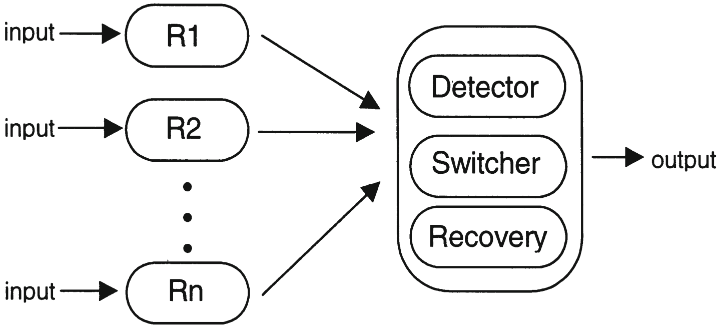

Figure 2: A redundancy-specific architectural style

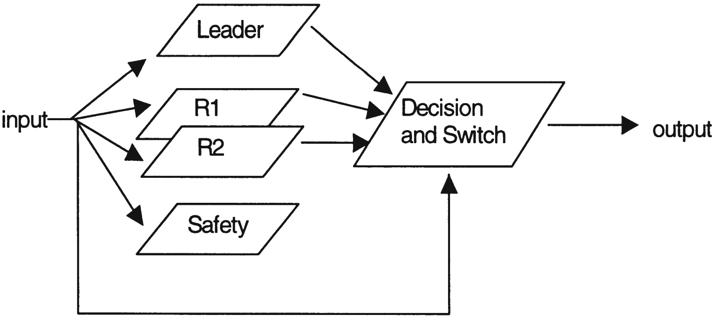

Figure 3: The Simplex architectural style

The "leader" component, the other redundant components (R1 and R2) and the "safety" component are analytically redundant. The "leader" is typically the upgraded version of a critical component. All components execute concurrently. The leader's output is used if it passes the acceptance test applied by the decision and switch unit. The acceptance test is based on a model of the controlled environment and the ability of the safety

[5] You can think of the relationship between power steering and mechanical steering as analytic redundant. Both mechanisms have the same effect on the environment, that is, they change the direction of the wheels, but the mechanisms used, the output produced, and their performance are all different.

component to recover from actions of the other components. If the leader doesn't pass this test a new leader is picked (either R1 or R2). The "safety" component is a simple, highly reliable analytically redundant component that is used as a last resort. The safety might be used to affect a recovery to the point where one of the other (more able) components can once again take over. Note that the decision and switch component receives a copy of the input and uses it as a basis for performing its acceptance test.

The Simplex style assumes that mechanisms exist to bound the execution time of the components, thereby preventing timing overruns. Another (related) style would address performance issues. The Simplex style also assumes that the concurrent units are processes with address space protection thereby preventing the propagation of system faults such as memory overruns.

5.2.4 Quality attribute parameters

Based on the architecture parameters of the reliability/availability attribute model and on the pattern of interactions, there are set of specific issues that should be raised to refine the pattern.

The quality attribute parameters of concern for this ABAS are:

- Analytic redundancy (possibly different input; different implementation; possibly different output) is the form of *redundancy*
- A leadership based "*voting*" mechanism is used.
- Estimates are needed for *failure rates* and *repair rates* of the various components. We assume that the failure rates for the decision unit and the safety component are very low in comparison to the failures rates of the other components.

5.2.5 Analysis

This section ties together the preceding sections. It discusses how to use the architectural decisions and properties and the stimuli to model the architectural behavior.

To model the availability of this style you have to make estimates of the failure rates and repair rates of the components to calculate the availability of the system. Reliability growth models can be used for obtaining estimates. In addition, it can be very illustrative to compare one architecture style to another simply by making assumptions about the various failure and repair rates. This is the approach we will use.

The first step in this section is to map the architectural decisions and properties into a quantitative or qualitative model that helps you to predict the architectural behavior.

Modelling a Simpler Problem

Before discussing the analysis of the Simplex style we'll first take a look at a similar style, the *majority voting* style.[6] This is the style that we used in the problem description to illustrate the update paradox. For this style there are three redundant components[7]. At least 2 or the 3 components must produce results that agree, otherwise the system has failed. When the system is working (in this case controlling some aspect of its environment, for example, the trajectory of a missile or the temperature and pressure of a chemical process) it is performing at a constant level of service. The system can be in one of three states: 3 working components, 2 working components, or failed. If F failures per year occur and a component repair takes on the average 1/R years then the Markov model shown in Figure 4 can be used to calculate the availability (that is, the proportion of time that the system is not in the failed state).

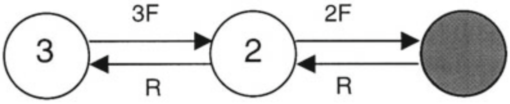

Figure 4: A Markov model for majority voting

The representation of a Markov model in Figure 4 can be viewed as a state diagram. State "3" represents the state in which 3 components are working, state "2" represents the state in which 2 components are working and the grey state is the failed state. The transition arrows are labelled with failure (F) and repair (R) rates. Since each component fails independently with an average rate of F, 3 components fail with an average fail rate of 3F and hence the label for the transition from state "3" to state "2".

The steady state solution of the Markov model yields the long-term proportion of time that the system is in each state. Therefore the availability of the majority voting case is the proportion of time in which the system is in state "3" or state "2", and hence not in the failure state.

More information about Markov models can be found in standard texts on probability. Our goal is to illustrate the mapping from architectural parameters to a predictive model and to show how the model provides the motivation for the characterization of the ABAS. In this case the predictive

[6] The majority voting style would probably have its own entry in a handbook of ABASs and be referenced in the Simplex ABAS.

[7] This is known as Trimodular redundancy (TMR). However, majority voting is not restricted to 3 components.

model is a mathematical model. In other ABASs qualitative reasoning techniques might also be used. For this case we use the model to gain an understanding of how the availability varies as a function of the assumed failure and repair rates, not to get absolute availability estimates. The trends of the majority voting style will then be compared with Simplex style.

Modeling Simplex

The Simplex style achieves relatively high levels of availability of the high performance (e.g., a very precise algorithm) variant by using a highly reliable but lower performing (e.g., a less accurate algorithm) variant to recover from faults. To illustrate the concept consider a system with two redundant controllers (R1 and R2), a safety controller, and a monitoring and decision unit. The Simplex style preserves the total number of active components, but allocates functions to components differently depending on their states, and hence the components have different failure properties. The Markov model for this style is shown in Figure 5.

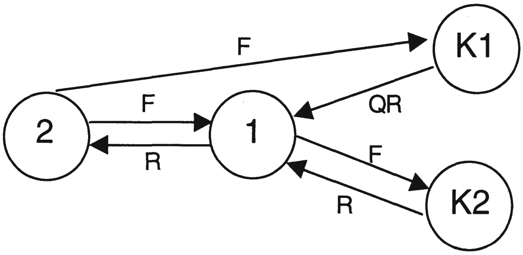

Figure 5: Markov model for Simplex

The system starts in state "2" with two high performance controllers, the outputs of which are compared. If they agree we assume that they're correct (that is, we assume no common mode failure, but rather random failures). If they disagree, one is picked. If the right one is picked the model transitions from state "2" to state "1". If the wrong one is picked the model transitions from state "2" to state "K1", where K1 stands for the a state in which the safety component becomes active. Since one of the high performance controllers continues to work, the transition from "K1" to "1" is relatively quick and thus has a quick repair (QR) rate. We assume that QR=n*R, for some n greater than 1. If a failure occurs while in state "1", the system also transitions to the safety controller, but in this case the repair rate is that of a "normal" repair (i.e. a software or hardware fix).

The final objective is to gain insight into the architecture by using the model as a basis of reasoning.

A key to the availability properties of this style is the relatively quick repair rate (QR) from state "K1" to state "1". To see this imagine that QR is so quick that virtually no time is spent in state "K1". In this case the model in Figure 5 closely approximates the model in Figure 6, below. The availability properties of the model shown in Figure 6 are better than for majority voting (shown in Figure 4) due to the higher transition rates for majority voting. The higher transition rates for majority voting are a consequence of needing a majority of the redundant components to agree in order to detect a failure, whereas this style uses a semantic check of the output for failure detection.

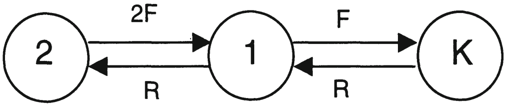

Figure 6: An approximate Markov model for Simplex

6. USING ABASs

We have applied ABASs on a real-world system during an enactment of the Architecture Tradeoff Analysis Method (ATAM) (Kazman, et al., 1998). During the course of the architectural analysis, ABASs relevant to several properties (availability, performance, and modifiability) were applied to aid in the understanding of the system and the consideration of design alternatives.

The system being evaluated—which we will call LAOB (Leader And One Backup) here[8]—comprises a collection of independently operating nodes (computers), communicating via a radio network, with a single node acting as leader. The leader has the responsibility to plan the activities of the other nodes. To perform this planning, it must accumulate and maintain data concerning the states of the other nodes.

Because the availability of the system is critical, we used a reliability ABAS to map the quality attribute parameters (i.e. the architectural decisions, such as the mechanisms for detection and recovery) and the predicted stimuli (e.g. failure of a node) onto the quality attribute measures (i.e. the predicted behavior) of the system via a reliability model. The resulting analysis was used to understand how well the system will meet its

[8] The actual name, developing organization, and details of this application are proprietary, but their suppression does not affect the analysis.

availability goals and to inform decisions for refining the architecture. In particular, by looking at the system via ABASs, we were able to determine that its reliability had not been adequately addressed in either requirements or implementation.

Quality Attribute Measures: Based on the reliability/availability attribute model presented in Section 5.1, the quality attribute measure of interest for this ABAS is its steady-state availability.

Stimuli: The stimuli of interest for this ABAS are hardware or software failures of the nodes.

Structure: The structure of the ABAS is shown in Figure 7. Communication takes place exclusively between the leader and the other nodes (i.e. the nodes do not communicate with each other). If the leader fails, a node pre-designated as a backup must reconfigure to take on the planning responsibilities of the leader and must acquire any additional data it needs to begin performing the leadership responsibilities. Also, another node must be identified to act as the new backup.

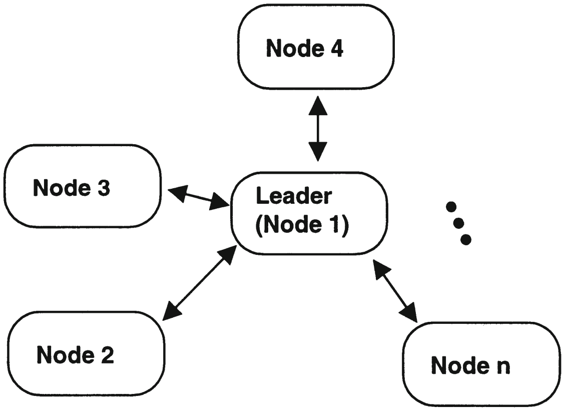

Figure 7: The ABAS-relevant structure of the LAOB system

Quality Attribute Parameters: The quality attribute parameters of interest in this style are:

- The mechanism used for detecting the failure of the leader: In the LAOB system, the lack of communication between the backup and the leader signals that the leader has failed.

- The mechanism for recovering system operation: When the leader fails, a designated backup takes over. The backup must acquire whatever data it requires to begin acting as leader and reconfigure itself.
- The failure and repair rates for the leader and the other nodes: The failure rates for the leader and the other nodes may be different as the leader has different responsibilities and is executing different software. The repair rates will need to include the time required for a node to take over as leader and the time required for a node to take over as backup. When we speak of repair, we are referring to the repair of the system, returning it to a functioning state from a non-functioning one. Individual nodes that have failed do not get repaired during the execution of the system.

Analysis: From our generic reliability ABAS we know that we can model this system using a Markov model. Figure 8 shows the Markov model for a three node system (it is easily generalized to more nodes). Each state in the model is labelled with a triple: (number of leaders active/number of backups active/total number of nodes active). F_l is the failure rate of a leader, F_b is the failure rate for the backup node, F_o is the failure rate for another node, R_l is the repair rate for the leader (i.e. the reciprocal of the time taken to transform a backup into the leader) and R_b is the repair rate for a backup (i.e. the reciprocal of the time taken to transform another node into a backup). The model makes the assumption that the transformation of the backup into leader and the transformation of another node into backup are sequential.

The steady-state availability can now be computed as the probability of the system being in a state in which a leader is active (four of the eight states). Based on expected failure and repair rates, the model can be used to understand how well the system will meet its availability goal.

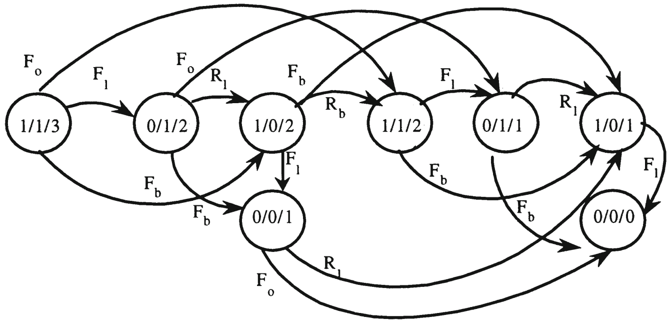

Figure 8: A Markov model for a reliability ABAS

Reasoning about the system in the context of the reliability ABAS led us to a consideration of other architectural alternatives for the LAOB system. The primary alternative considered was the use of multiple backups to turn the LAOB into a LAMB (Leader And Many Backups), where each of the backups would maintain the state necessary to quickly take over upon failure of the leader.

This alternative will result in better availability due to a reduced repair time, but at a cost of higher utilization of the radio network. Since the radio network had a relatively low bandwidth, this was not a trivial consideration: keeping additional backups informed of the state of the leader meant additional transmissions and retransmissions. The performance issues for the LAOB/LAMB system alternatives were considered using a separate ABAS, one for communicating processes, and the confluence of these two ABASs identified an architectural tradeoff, since higher levels of availability meant higher utilization of the network.

The purpose of this example is not to present the design decisions made for this system, or even the details of the analysis, for they are not the point of this paper. The point is that a consideration of ABASs led us to ask questions of the system: reliability ABASs made us ask questions about failures and recovery of components and their effects on the predicted level of system availability; performance ABASs made us ask questions about resource characteristics and resource utilization and their effects on the predicted level of system response times. Using these models, we could play with different architectural alternatives, constantly gauging the performance of these alternatives against the system's requirements. For example, we could explore versions of the LAMB system with varying numbers of backups and with different strategies for keeping them synchronized with the leader. Strategies include:

1. The backups could be passive recipients of updates, not worrying about any missed information until they are called upon to become the leader. In this case they would not be guaranteed of being true functional replicas of the leader.
2. They could be active recipients, requesting re-sends of any missed packets (they could identify missed packets via noting holes in the packet number sequence, for example). In this case they will be functional replicas of the leader most of the time, but at the cost of additional communication with the leader.
3. A single backup could be an active recipient and all other backups could be passive recipients. When the primary backup was called upon to become the leader, it would designate a new primary backup and negotiate with it to provide it with any missed packets, at the cost of additional communication at switchover time.

The various strategies each have different availability and performance implications—different bandwidth requirements, different time to repair, different probabilities of failure—and these can be modelled analytically before committing to one strategy for prototyping or implementation. Perhaps more importantly, these analyses can be used to find architectural tradeoff points—critical areas of the design with respect to some qualities of interest—and these can become the focus of additional analysis or prototyping as a means of mitigating the risk of building a large, complex software-intensive system.

7. CONCLUSIONS

An ABAS is an extension of the notion of an architectural style. To make architectural styles more rigorous, we associate analytic models of quality attributes with them, in much the same way that Allen and Garlan associate formal semantics with architectural elements to better describe the correctness of a design (Allen and Garlan, 1994). So, an ABAS has associated with it a set of analytic models (such as performance or reliability models) that allow a designer to predict its behavior with respect to some desired quality attributes. ABASs provide to the designer a pre-analyzed structural framework, an analysis, and a mapping between the structure and the analysis.

Associated with the mapping from architectural style to analytic model are two related processes:

1. from a design perspective, there are a set of decisions that accompany turning a style into an implementable design. For example, when decomposing a system's functionality into a set of processes, there is an allocation of functionality to each process, and an allocation of processes to processors. For a performance style we might also make decisions such as choosing the priorities of the processes.
2. from an analysis perspective, there are a set of questions that accompany an architectural style that aid in understanding the style. These questions will ask about the allocation, for example, of processes to processors, their communication mechanisms, the speeds of their connections.

If these questions relate to designs that are repeated over and over again within an organization, then they are often organized into checklists (Maranzano) that are employed during architectural reviews. The answers to the questions form the input to the attribute models. This is the key linkage that comprises the reasoning behind an ABAS: architectural parameters—the things that you can change when you do architectural design—are explicitly related to parameters in an analytic model. In solving the model, we are then modeling the expected behavior of the architecture. The results of this model solving can then be compare back to the expected behavior.

We envision, and are actively working on, a handbook with many ABASs that can be looked to for pre-packaged design and/or analysis wisdom. This is the start of an attempt to make architectural design more of an engineering discipline; one where design decisions are made upon the basis of known properties and well-understood analyses, rather than the currently popular practice of "patch-and-pray".

REFERENCES

R. Allen, D. Garlan, "Formalizing Architectural Connection", *Proceedings of ICSE 16*, (Sorrento, Italy), May 1994, 71-80.

L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, Addison Wesley, 1998.

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture*, Wiley, 1996.

E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns—Microarchitectures for Reusable Object-Oriented Software*, Addison-Wesley, 1994.

R. Kazman, G. Abowd, L. Bass, P. Clements, Scenario-Based Analysis of Software Architectures, IEEE Software, November 1996.

R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, "The Architecture Tradeoff Analysis Method", *Proceedings of ICECCS '98*, (Monterey, CA), August 1998, to appear.

M. Klein, T. Ralya, B. Pollak, R. Obenza, M. Gonzales Harbour, *A Practitioner's Handbook for Real-Time Analysis*, Kluwer Academic, 1993.

J. Maranzano, *Best Current Practices: Software Architecture Validation*, AT&T, 1993.

J. McCall, "Quality Factors", *Encyclopedia of Software Engineering* (Marciniak, J., ed.). *Vol. 2*. Wiley, 1994, 958-969.

L. Sha, R. Rajkumar, M. Gagliardi, "A Software Architecture for Dependable and Evolvable Industrial Computing Systems", CMU/SEI-95-TR-005, Pittsburgh, PA: Software Engineering Institute, 1996.

M. Shaw, D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.