Attribute-Based Architectural Styles

Mark Klein
Rick Kazman

October 1999

TECHNICAL REPORT
CMU/SEI-99-TR-022
ESC-TR-99-022

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

Attribute-Based Architectural Styles

CMU/SEI-99-TR-022
ESC-TR-99-022

Mark Klein
Rick Kazman

October 1999

Product Line Practice

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Norton L. Compton, Lt Col., USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

List of Tables

Abstract

An architectural style is a description of component types and their topology. It also includes 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 engineering artifacts because they define *classes* of designs along with their associated known properties. They offer experience-based evidence of how each class has been used historically, along with qualitative reasoning to explain *why* each class has its specific properties.

Attribute-Based Architectural Styles (ABASs) build on architectural styles to provide a foundation for more precise reasoning about architectural design by explicitly associating a *reasoning framework* (whether qualitative or quantitative) with an architectural style. These reasoning frameworks are based on quality attribute-specific models, which exist in the various quality attribute communities (such as the performance and reliability communities).

Architectural styles, and hence ABASs, are powerful because they provide a designer with the concentrated wisdom of many preceding designers faced with similar problems. In this report we exemplify the use of ABASs in both design and analysis. We argue that ABASs provide the groundwork to create an engineering discipline of architectural design—to make design a predictable process rather than an ad hoc one.

1 Introduction

An architectural style (as defined by Shaw and Garlan [Shaw 96]) and elaborated on by others [Buschmann 96], is a description of component types and their topology, which includes a description of the pattern of data and control interaction among the components. Architectural styles also provide an informal description of the benefits and drawbacks of using that style. Architectural styles are important engineering artifacts because they define *classes* of designs along with their associated known properties. They offer experience-based evidence of how each class has been used historically, along with qualitative reasoning to explain *why* each class has its specific 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 has been found in existing definitions of the pipe and filter style. Architectural styles are powerful because they provide a reuser with the concentrated wisdom of many preceding designers faced with similar problems. Using architectural styles allows an architect to reuse the collected wisdom of the architecture design community in much the same way that object-oriented design patterns give novice designers access to a vast array of experience collected in the object-oriented design community [Gamma 95].

In 1999, we introduced the notion of an Attribute-Based Architectural Style (ABAS), the purpose of which is to make architectural styles the foundation for more precise reasoning about architectural design [Klein 99]. This is accomplished by explicitly associating a *reasoning framework* (whether qualitative or quantitative) with an architectural style. These reasoning frameworks are based on quality attribute-specific models, which exist in the various quality attribute communities (such as the performance and reliability communities). This means that ABASs are not only attribute based, but unlike existing architectural styles, are also attribute *specific*. When we design or analyze using ABASs, we consider only one quality attribute at a time, because each ABAS is associated with only one attribute reasoning framework (what we call an "attribute model"). An architectural style that is interesting from both a performance and a reliability point of view would be motivation for creating the respective performance and reliability ABASs. For example, there could be distinct pipe-and-filter performance and reliability ABASs.

We view ABASs as the next generation in the development of architectural styles. Using ABASs is a step in moving architecture design closer to being an engineering discipline [Shaw 90] and to bring the benefit of an engineering discipline—predictability—to architecture design. ABASs promote a new way of doing the design and analysis of software architecture based on reusing known patterns of software components with predictable properties. We will emphasize this point throughout this report and exemplify it in a design example in Chapter 4.

The purpose of this report is to

- take a step in showing the utility of ABASs by presenting several samples covering the quality attributes of modifiability, performance, and availability
- offer a variety of examples of how ABASs can be used in both design and analysis
- show how ABASs can aid in discovering design tradeoffs

The grander purpose of this report is to try to establish a common format for documenting ABASs in the hope that they will become the foundation of an architect's handbook when doing design and analysis.

We will begin by discussing quality attributes and how we characterize them, and next we present an example ABAS to show how we document them. Then we present an example of designing using ABASs and briefly discuss how we use ABASs in aiding architecture tradeoff analysis.

2 Quality Attribute Characterizations

ABASs show how to reason about architectural decisions with respect to a specific quality attribute such as performance, security, or reliability. However, a prerequisite for constructing and using ABASs is to have a precise characterization for the quality attribute of concern. For example, understanding a style from the point of view of modifiability requires an understanding of how to measure or observe modifiability and an understanding of how architectural decisions impact this measure. To facilitate using the wealth of knowledge that *already* exists in the various quality attribute communities, we have created a standard characterization for the quality attributes of performance, modifiability, and availability.

We divide quality attribute information into three categories: *external stimuli*, *architectural decisions*, and *responses*. External stimuli (or just *stimuli* for short) are the events that cause the architecture to respond or change. 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 attribute characterization. Architectural decisions are aspects of an architecture—components, connectors, and their properties—that have a direct impact on achieving attribute responses.

For performance, the external stimuli are events arriving at the system such as messages, missiles, or user keystrokes. The architectural decisions include: 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 change requests to the system's software. Architectural decisions include encapsulation and indirection mechanisms, and the response is measured in terms of the number of affected components, connectors, and interfaces and the amount of effort involved in changing these affected elements.

This way of characterizing a quality attribute is central to ABASs. You will see the role played by attribute characterizations in the next section, which offers a walkthrough of an ABAS. The characterization of performance that we are currently using is given in Figures 1–3.

These characterizations transcend their use in ABASs. We simply take advantage of the existing body of knowledge in each of the quality attribute communities, and we organize this knowledge in a consistent way, once, for use in every ABAS related to a particular quality attribute. That is to say, all modifiability ABASs are organized around the same modifiability

characterization; all availability ABASs are organized around the same availability characterization, and so forth.

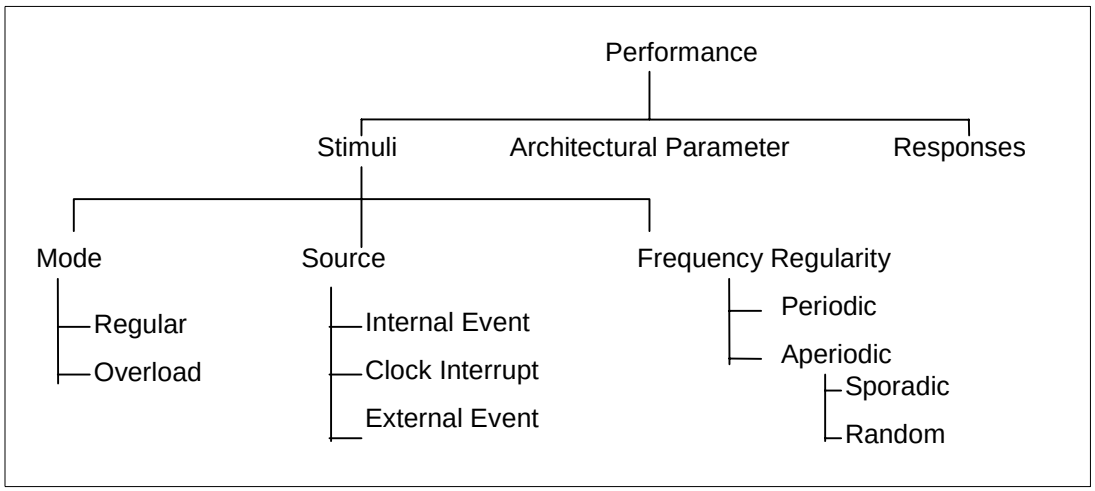

Figure 1: Performance Characterization—Stimuli

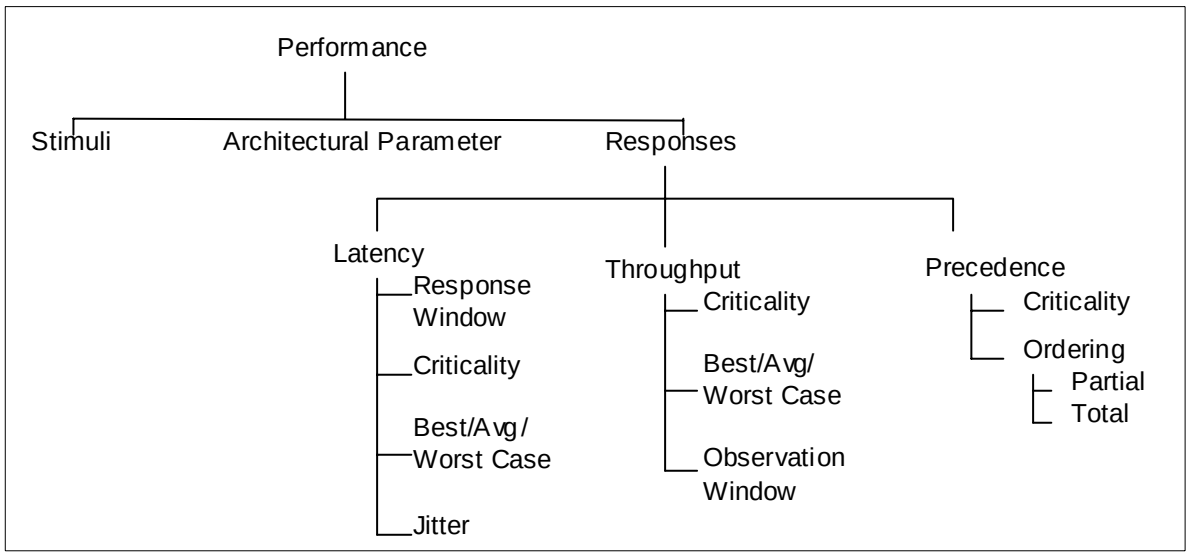

Figure 2: Performance Characterization—Responses

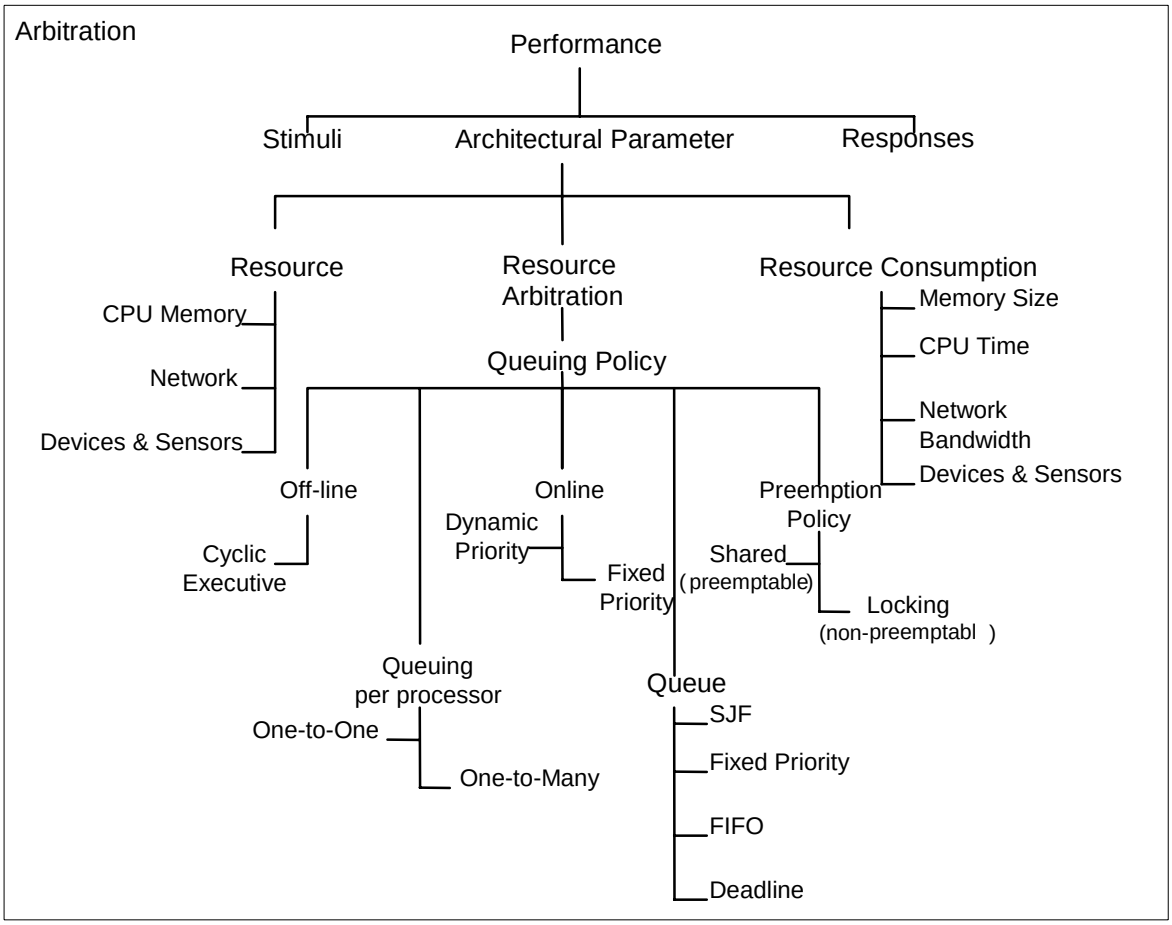

Figure 3: Performance Characterization—Architectural Decisions

3 ABAS Example

Every ABAS has the same four parts:

1. **Problem description** - informally describes the design and analysis 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. **Stimulus/Response attribute measures** - a characterization of the stimuli to which the ABAS is to respond and the quality attribute measures of the response.
3. **Architectural style** - a description of the architectural style in terms of its components, connectors, properties of those components and connectors, and patterns of data and control interactions (their topology), and any constraints on the style.
4. **Analysis** - a description of how the quality attribute models are *formally related* to the architectural style and the conclusions about "architectural behavior."

Many ABASs are intended to achieve similar goals and do so via similar structural means. For example, modifiability ABASs use various forms of indirection; availability ABASs use various forms of redundancy. So, these naturally fall into groups, which we call ABAS families. In this report, we will present one family of related modifiability ABASs. These are all related to the Data Indirection ABAS and are found in Appendices B-D. We organize ABAS families as a hierarchy, with one ABAS at the root and the remaining *sub-ABASs* inheriting (and overriding) some of the properties of the root ABAS.

For the moment, however, we will concentrate on explaining the parts of an ABAS via a single example. The following is a walkthrough of an example ABAS: Concurrent Pipelines.

This ABAS is called "Concurrent Pipelines" because it takes the pipe and filter architectural style [Shaw 96] and places it in a real-time setting, allowing for multiple concurrent chains of pipes and filters. Real-time performance is the attribute of concern in this ABAS.

3.1 Problem Description

The problem description offers an informal description of the situation the ABAS is intended to offer insight for, including the attribute-specific requirements to which the ABAS applies.[1]

Classical pipe and filter systems are used when a sequence of transformations is applied to a stream of data by a sequence of processes (or threads), producing a final output. Concurrent pipelines are needed when multiple streams are co-located on a single processor and where there are real-time requirements associated with the production of final outputs. In systems implemented in such a style, there are sometimes unintended performance consequences. The purpose of the Concurrent Pipelines ABAS is to provide a reasoning framework so that the performance consequences of architectural decisions are understood at design time.

For the Concurrent Pipelines ABAS, we consider a single processor on which multiple processes reside and are organized into sequences. Each process performs computations on its 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. The input data will be referred to as a message. The requirement then is to completely process each message with a specified bounded end-to-end latency—a deadline.

The analysis focus of the Concurrent Pipelines ABAS is how to reason about the effects of the process prioritization strategy on end-to-end latency.

3.1.1 Criteria for Choosing this ABAS

These criteria are meant to summarize situation(s) for which this ABAS is relevant by describing the style's topology and the attribute response behavior that the ABAS allows a designer to reason about and predict.

This ABAS will be relevant if

- your problem inherently has real-time latency requirements associated with the production of final outputs
- the topology you are using or considering consists of multiple processes arranged as concurrent pipelines

1. Text shown in italics in this section is descriptive and not part of the ABAS itself.

3.2 Stimulus/Response Attribute Measures

The important stimulus and the response that we want to reason about, control, and measure are characterized as follows:

- **Stimulus**: periodic or sporadic arrival of messages
- **Response**: worst-case latency associated with processing this message

3.3 Architectural Style

This section explicates the topology and key architectural decisions.

An example of the Concurrent Pipelines style is shown in Figure 4. Multiple messages arrive at the first process in each sequence (for example, process P11 and Pn1). In this style, messages are queued using a FIFO (first in, first out) queuing discipline and are processed when they reach the head of the queue. Each process vies for the processor using a fixed-priority, preemptive-scheduling discipline.

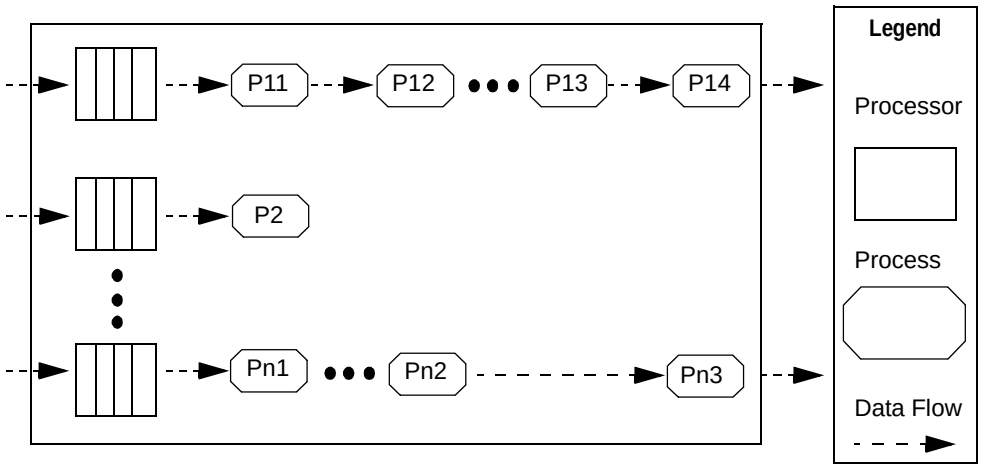

Figure 4: Concurrent Pipelines

You can think of each sequence of processes as a pipeline. Each input message is incrementally transformed by each process in the sequence. Hence, sequential messages of the same type might be in different stages of processing at the same time. As shown in Figure 4, there can be more than one pipeline on the processor (some of which might be trivial).

Systems are built like this to take advantage of the classical benefits of a pipe and filter architecture [Shaw 96]. Pipe and filter architecture benefits are

- The system can be easily understood as a sequence of data transformations.
- Each filter can be modified or replaced, in principle, without affecting any of the other filters.
- Filters can be reused elsewhere.

This ABAS can also arise in object-oriented systems. From a module perspective, the system might look like a set of interacting objects, each in its own thread, but from a process perspective, it could look like concurrent pipes and filters.

The architectural decisions that are important for performing analyses are listed in the table below. This table is based on the performance attribute characterization given in Chapter 2.

The architectural parameters of concern for this ABAS are those necessary for creating an analytic model of end-to-end, worst-case latency. Table 1 describes the performance architectural parameters and their values.

Performance Architectural Parameters
topology***:*** *pipeline(s)*
preemption policy: *priority-based preemption*
execution time for each process associated with processing each input: C_i
prioritization strategy: *sequence of priorities in the pipeline*
process scheduling discipline: *fixed priority*

Table 1: Architectural Decisions for the Concurrent Pipelines ABAS

3.4 Analysis

The goal of the analysis is to convey the relationships between topology and key architecture decisions and the stimulus/response behavior. A formal analysis is presented when possible; otherwise qualitative analysis is performed.

In the following sections, we present a formal analysis showing how to predict the end-to-end, worst-case latency in this ABAS given a knowledge of the parameters listed in Table 1. We also include a set of informal qualitative analysis heuristics, telling the designer what issues to be aware of in creating and assessing any design of a concurrent pipelines ABAS.

3.4.1 Reasoning

This section articulates the key points for how to think about this ABAS in terms of a formal performance analysis.

The analysis issue for this ABAS is how to reason about the incremental processing of inputs, where each increment can be made by a different process, each executing at its own fixed priority [Gonzalez 91]. To calculate the latency of a message traversing the i^{th} pipeline, you must determine the preemptive effects of the other pipelines. The key to determining these preemptive effects is to first identify the lowest priority process in the i^{th} pipeline. In brief, the following steps can be used to obtain an estimate of the worst-case latency for an input message using the i^{th} pipeline consisting of processes P_{i1}, P_{i2} ... P_{im}:

1. Determine the priority of the lowest priority process in the i^{th} pipeline, denoted by $LowP_i$.
2. Determine the set of pipelines whose lowest priority process has a priority greater than $LowP_i$. In other words, all of the processes in these pipelines have a priority greater than $LowP_i$. Denote this set as H for high.
3. Determine the set of pipelines that start with processes whose priority is greater than $LowP_i$ but eventually drop below $LowP_i$. Denote this set by HL, standing for starting higher and dropping lower.
4. Determine the set of pipelines that start with processes whose priority is lower than $LowP_i$ but eventually rise above $LowP_i$. Denote this set by LH, standing for starting lower and rising to higher.

Calculate the worst-case latency for the i^{th} pipeline by iteratively applying the following formula until it stabilizes.

$$l_{n+1} = \sum_{j \in H} \left\lceil \frac{l_n}{T_j} \right\rceil C_j + C_i + \sum_{j \in HL} C_j + \max_{j \in LH}(C_j) \qquad (1)$$

The above steps for calculating latency illustrate the sensitivity of the pipeline's latency to the priority of the lowest priority process in the pipeline under scrutiny (i.e., $LowP_i$) since the priority categories (i.e., H, HL, and LH) are determined by $LowP_i$.

3.4.2 Analysis and Design Heuristics

This section consists of qualitative questions and heuristics, which direct attention to the relationships between architectural decisions and the stimulus/response behavior of the style. The previous section provides a starting point for answering these questions and thus provides a basis for developing a skeletal analysis.

Even if one does not build a formal analytic model of the latency in a concurrent pipelines ABAS, a designer should keep in mind that the latency of pipelines is very sensitive to the prioritization strategy, in particular to the priority of the lowest priority process in the pipeline. Thus, when designing a system that employs such an ABAS, one should ask

- How does your choice of priority assignment impact latency?
- Is there another prioritization strategy that might reduce latency, in particular should the priority of the lowest priority process in each pipeline be changed?
- Is the architectural design flexible enough to accommodate reprioritization later in design?
- Is the effect of reallocating functionality to processes easily understood?

To realize the concurrent pipelines ABAS, one should consider the following:

1. What are desirable priority assignments?
   Answer: In many situations assigning higher priorities for shorter deadlines is a good strategy.

2. What if deadlines are beyond the end of the period?
   Answer: When deadlines are beyond the end of the period it is not sufficient to only examine the completion time of the first job as this job might not be the one with the longest completion time.

3.5 Summary

This example has shown the parts of an ABAS and illustrated the purpose of each part. The problem description delineates how and why you would go about using this ABAS—the kind of problem on which you need to get an engineering handle. The stimulus/response attribute measures tell you precisely what measurable attribute of the system about which you can reason. The architectural style tells you what key design decisions are fixed if you want to use this ABAS and what decisions you still need to make as an architect. The analysis section tells you, both formally and informally, how to think about this ABAS and to what key properties of the style need to be paid special attention.

4 Using ABASs for Design and Analysis

Imagine a handbook with a collection of ABASs, including several ABASs for performance, several for reliability, several for modifiability, and so on. To help you imagine such a handbook, we have included six ABASs in the Appendices, pages 31–72, of this report. You are strongly encouraged to turn to these ABASs as an adjunct to understanding this section. This is in the spirit of a handbook—one typically doesn't read a handbook cover to cover, but rather reads small sections on an as-needed basis to analyze the problem at hand.

Given a collection of ABASs, one natural question to ask is: how can multiple ABASs be used together to facilitate the design and analysis of an architecture? The purpose of this section is to suggest some answers to this question. A model problem will be used to help illustrate the use of ABASs.

4.1 Model Problem

The Sea Buoy problem: A collection of buoys is floating in the sea to acquire and maintain navigation and weather data, and to provide these data to air and sea traffic [Booch 96]. Each buoy collects and maintains air and water temperature, wind speed, and buoy location data. Wind-speed readings are to be taken every 30 seconds; air and water temperature readings are to be taken every 10 seconds; and buoy location readings are taken every 10 seconds. Temperature, wind, and location information is broadcast every 60 seconds using a radio transmitter attached to each buoy.

On demand, a 24-hour history of wind, temperature, and location information is broadcast in response to requests from passing vessels. Requests are received via a radio receiver on each buoy. This history transmission takes priority over the periodic broadcasts.

If sailors in distress can reach the buoy, they can flip an emergency switch to activate the continuous broadcasting of an SOS signal, which takes priority over all other transmissions until it is reset by a passing vessel. Some buoys are equipped with a red light that can be activated/deactivated by a passing vessel during sea-search operations.

Each buoy can have a different number of wind and temperature sensors, and each buoy might be modified in the future to support different sensor types.

The sea buoy problem has explicit performance and modifiability requirements. We will add a new availability requirement, which is clearly important to this problem even though this was not stated explicitly in the original description. A number of ABASs described in the Appendices of this report will be employed to create an initial design that satisfies the basic requirements of this problem. These ABASs include

- The *Synchronization ABAS,* which is relevant because most likely, mutually exclusive access to one or more data repositories will be necessary.
- The *Concurrent Pipelines ABAS,* which is relevant because there will be several pipelines of information in the system and it is important to reason about the performance of these pipelines. In particular, it is important to reason about how managing the performance of one pipeline can affect the others.
- The *Abstract Data Repository ABAS* which is relevant because it addresses modifications such as changes in sensors and sensor types that might result in data format changes or the introduction of new data types.

Our strategy in presenting this example is to show how to incrementally build a design by applying different ABASs to address different parts of the problem space. We will begin by addressing the need to have mutually exclusive access to shared data in the Sea Buoy problem while meeting performance goals (i.e., it is desired that access to the shared data does not become a performance bottleneck in the system).

4.2 Applying the Synchronization ABAS

The *Synchronization ABAS* (detailed in Appendix A, page 31) is a single-processor, client-server style augmented with real-time scheduling analysis. It highlights the key architectural decisions that will have an impact on performance and then offers quantitative and qualitative advice for reasoning about the impact of those decisions. The architectural decisions highlighted by the ABAS are

- the choice of processes and their relationships
- the process prioritization strategy
- the scheduling discipline
- the shared-data, store-locking policy

The *Synchronization ABAS* is directly applicable to the Sea Buoy problem. This is because sensor data must be stored and later retrieved for periodic or on-demand transmission, and so

we need to have a way for processes to share a data store. The wind sensor (W), air temperature (Ta), water temperature (Tw), and location (Loc) functions can each be mapped into their own processes, all of which write to or read from a common data store (DB) as shown in Figure 5.

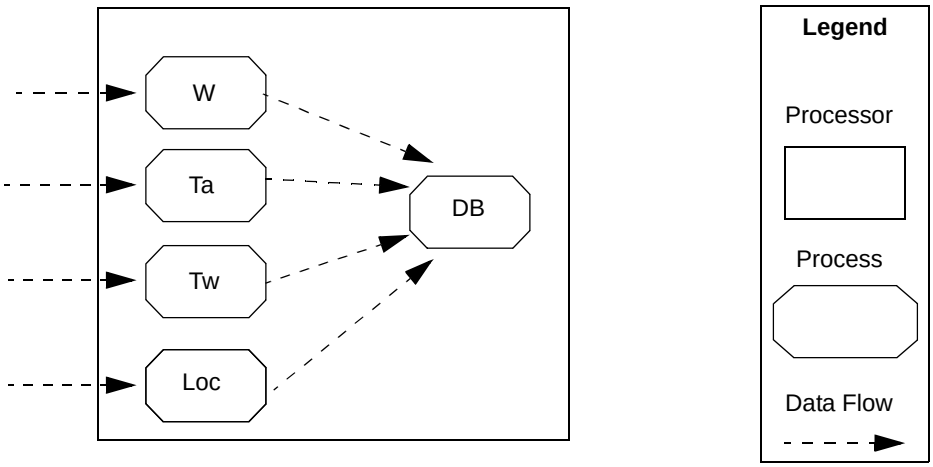

Figure 5: Initial Concurrency View for Sea Buoy

The *Synchronization ABAS* offers insight for how to reason about the impact of architectural decisions. In this case, the insight is packaged in the formula (shown below), which shows the effects of the various design decisions on latency. In this formula; C denotes execution time; T denotes period; and B denotes blocking time.

$$L_{n+1} = \sum_{j=1}^{i-1} \left\lceil \frac{L_n}{T_j} \right\rceil C_j + C_i + B_i \qquad (2)$$

Section A.4.1 explains how the terms in this equation reflect the topology shown in Figure 5.

Given that the processes with the shortest periods are to be assigned the highest priorities (as suggested by the ABAS), the impact of the architectural decisions embodied by this style on the latency of the wind-sensing process is captured by the following formula.

$$L_{n+1} = \left\lceil \frac{L_n}{T_{Loc}} \right\rceil C_{Loc} + \left\lceil \frac{L_n}{T_{Ta}} \right\rceil C_{Ta} + \left\lceil \frac{L_n}{T_{Tw}} \right\rceil C_{Tw} + C_W \quad (3)$$

The preemptive effect of high priority processes is captured by the terms with ceiling functions. The execution time of the wind-sensing process is accounted for by C_W. There are no blocking (B) terms representing the effect of lower priority processes on latency. Since the wind-sensor process is the lowest priority process (that we have included in the architectural design up to this point), B_i is equal to zero. This will change as we flesh out the design.

4.3 Composing Like-Attribute ABASs

Our strategy is to incrementally incorporate different aspects of the problem into the architectural design (the solution). However, this begs the question of how to *compose* multiple like-attribute ABASs—each of which addresses a small aspect of the design—to realize a complete solution. To facilitate composition, we focus our attention on the analytic underpinnings and let these guide our design. For example in this case, the latency for the wind-sensing process can be viewed as a function of execution times (C), priorities (P), and periods (T):[2]

$$Latency_{Wind} = f_{Wind}(C_{Loc}, C_{Ta}, C_{Tw}, C_W, P_{Loc}, P_{Ta}, P_{Tw}, T_{Loc}, T_{Ta}, T_{Tw})$$

Composing other aspects of the problem with the initial style discussed above involves recognizing the impact of other aspects of the problem on this function and vice versa. That is, to compose ABASs, we need to understand where decisions made in some other ABAS might affect terms such as C_{Loc}, C_{Ta}, and C_{Tw}. This illustrates how design via ABASs work; we consider not only the functionality and topology of the solution, but also how the measurable properties of the system will be affected by each design decision. Places where multiple ABASs affect a single property require special attention on the part of the architect [Kazman 99].

2. Here and in the following discussion, we will not show the details of the modified formulae, as they simply follow the pattern of Equation 3.

Reapplying the Synchronization ABAS

To begin illustrating how we compose ABASs, consider adding the broadcast (BC) and transmission (XMT) functions to the design presented in Figure 5. The resulting design is given in Figure 6.

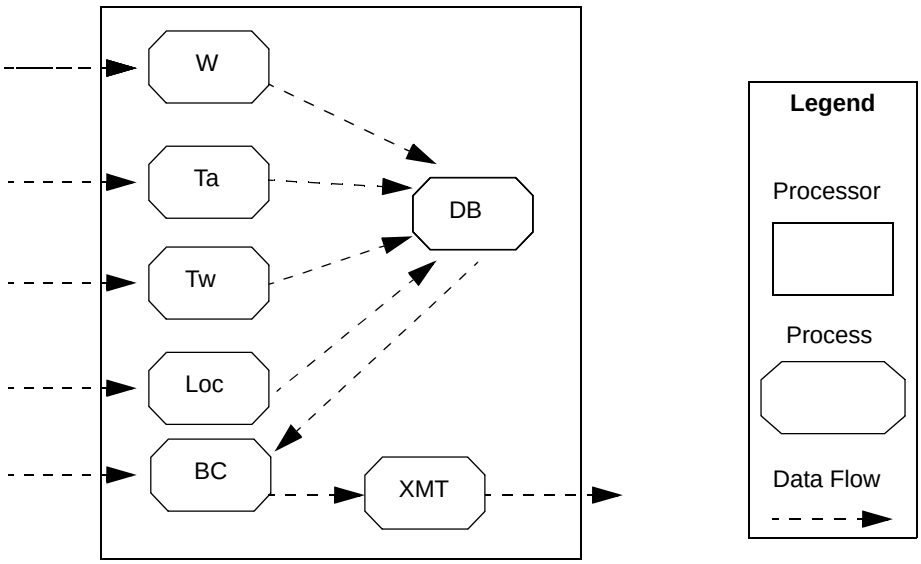

Figure 6: Concurrency View After Adding Broadcast and Transmission

First, from the point of view of the *Synchronization ABAS*, BC is a new client of DB. The reasoning associated with the *Synchronization ABAS* would suggest assigning the broadcast process a lower priority than the other periodic processes because of its lower frequency, which is dictated in the problem description. The first question is then: how does this affect $Latency_{Wind}$? Since the broadcast process' priority is lower than that of the other periodic tasks, $Latency_{Wind}$ does not depend on the execution time or period of broadcast. On the other hand, since broadcast accesses DB, the possibility exists that it can *block* the wind-sensing process and thus affect $Latency_{Wind}$. Therefore, the representation of $Latency_{Wind}$ is now modified to reflect that it is also a function of blocking due to broadcast:

$$Latency_{Wind} = f_{Wind}(C_{Loc}, C_{Ta}, C_{Tw}, C_{W}, P_{Loc}, P_{Ta}, P_{Tw}, T_{Loc}, T_{Ta}, T_{Tw}, B_{BC})$$

This brings us to an important question: how do the other periodic processes affect Latency-$_{Broadcast}$? The other processes preempt broadcast and thus $Latency_{Broadcast}$ is a function of their periods, priorities, and execution times and its own execution time:

$$Latency_{Broadcast} = f_{Broadcast}(C_{Loc}, C_{Ta}, C_{Tw}, C_W, C_{BC}, P_{Loc}, P_{Ta}, P_{Tw}, P_W, T_{Loc}, T_{Ta}, T_{Tw}, T_W)$$

Applying the Concurrent Pipelines ABAS

The broadcast function is not complete until the data is actually transmitted via the transmission process. The broadcast and transmission processes form a simple pipeline (as shown in Figure 6). We can draw on the Concurrent Pipelines ABAS discussed above for insight here as it offers the rule of thumb that the *effective* priority of the pipeline is strongly related to the lowest priority of all processes in the pipeline. This calls specific attention to the relative priorities of the broadcast and transmission processes. In particular, it tells us that assigning the transmission process a priority lower than that of the broadcast process can, in effect, lower the priority of the entire pipeline. This observation would be critical if the priority of the broadcast process were higher (to meet an early deadline, for example).

We still have to account for execution time needed to actually carry out the transmission of the data. $Latency_{Broadcast}$ is therefore also a function of $C_{BC} + C_{XMT}$:

$$Latency_{Broadcast} = f_{Broadcast}(C_{Loc}, C_{Ta}, C_{Tw}, C_W, C_{BC} + C_{XMT}, P_{Loc}, P_{Ta}, P_{Tw}, P_W, T_{Loc}, T_{Ta}, T_{Tw}, T_W)$$

There are several observations that can be made at this point. First, we have shown how ABASs can offer insight pertaining to architectural design decisions that have a key influence on requirements, in this case, latency requirements. Second, we have shown that a key to composing ABASs is to understand how the architectural decisions of one ABAS affect the stimulus/response behavior of the other ABAS, and vice versa. We have only illustrated composition for the relatively simple case of composing ABASs of the same attribute, which also use the same underlying modeling technique. In subsequent sections, we will discuss relaxing these restrictions.

Composing Different Analytic Approaches

The history function has not yet been folded into the design. The performance characteristic, which distinguishes this function is that it is event driven (or aperiodic). Up to this point, the underlying analytic approach used in the Synchronization and Concurrent Pipeline ABASs has

been Rate Monotonic Analysis (RMA) which is primarily a deterministic method. Assuming that the demands for historical information are stochastic, the history function might be more naturally treated using a stochastic technique such as queuing analysis.

Composing ABASs, in this case, hinges on being able to understand the interactions between the two analytic models employed: RMA and queuing analysis. A queuing analysis might need to be adjusted to account for preemption and synchronization. An RMA analysis might need to be adjusted to account for random bursts of aperiodic arrivals. A detailed discussion of these interactions is beyond the scope of this report.

The main point that we want to make here is that by revealing the analytic underpinnings of an architectural style (through formulae and/or a collection of heuristics), ABAS composition involves understanding the interactions at an analytic level. The analytic underpinnings help guide the designer to places of interaction between disparate modeling techniques.

ABASs Offer Clues for Extrapolating Beyond the Specific Situation

No matter how large (and seemingly comprehensive), a collection of ABASs will always be incomplete; there will always be situations that are not exactly covered by one or more ABASs in the current collection. However, ABASs are designed with rules of thumb and references for extrapolating beyond the specific model situation described. To illustrate, we now introduce the design constraint that the SOS, history and broadcast processes all use the transmission process.

This situation looks much like the situation discussed in the Synchronization ABAS. However, the original requirements state that the history function takes priority over the periodic broadcast function, and the SOS function takes priority over everything else. To accommodate the apparent need for a quick response to the SOS request, the SOS message might have to interrupt a relatively lengthy broadcast of the 24-hour history. This, in turn, means that the transmission function needs to be preemptable. However, the Synchronization ABAS treats the server process (in this case XMT) as a critical section. It is the critical section that results in the blocking terms above. The effect of interrupting is to reduce or eliminate the blocking term.

While the Synchronization ABAS does not deal directly with preemptable servers, it does highlight the relationship between critical sections in the server and blocking. The ABAS also highlights the relationship between the priority of the server process and blocking. This knowledge gives an architect a handle on how to make sensible design decisions. For example, the ABAS allows one to deduce that sending a long message in small chunks, where the transmission is preemptable between chunks, is a viable way to reduce the blocking of the SOS process to a level that is acceptable for the SOS function.

4.4 Composing ABASs of Different Attribute Types

Until now, we have exclusively focussed our design on performance considerations and the ABASs that aid us in achieving our performance goals. These ABASs have shaped the software architecture to the state that is represented in Figure 6 on page 17. In this section, we will consider how to go about composing ABASs that focus on different quality attributes and see what additional kinds of reasoning ABASs bring to bear on this more demanding problem.

Folding in Modifiability Requirements

One explicit modifiability requirement in the Sea Buoy problem description is the need to support adding new sensor types. The Data Indirection ABAS (detailed in Appendix B) and its sub-ABASs, the *Abstract Data Repository Sub-ABAS* (detailed Appendix C), and the *Publish/Subscribe Sub-ABAS* (detailed in Appendix D) are all potentially applicable here as they deal with the situation in which data producers and consumers interact with a data repository. Consulting the relevant ABASs, several considerations become apparent from the reasoning section. These considerations are as follows:

- Will the new sensors be producing data using the same or a different data format?
- Are the new sensors being added to or substituted for existing sensors?
- Is a new environmental parameter (such as air pressure) being sensed?

The various scenarios considered in the Analysis section of the Data Indirection ABAS and its sub-ABASs (see Section B.4) offer insights for considering these questions. If the new sensors are using the same data format, then the modification has a minimal impact. If a different data format is being used, the impact is potentially pervasive when using the Data Indirection ABAS, but is easily managed using the Abstract Data Repository or the Publish/Subscribe sub-ABASs. Likewise, if a new environmental parameter is being sensed the impact could also be pervasive.

Since in the case of the Sea Buoy problem, we want to plan for expansion of capabilities, we need to explicitly allow for new data types being stored in the repository. In addition, we have little control over the outputs of the sensors (since these are purchased “off the shelf”). As a result, we assume that the sensors will be producing data in different formats. Considering these scenarios, we find the Abstract Data Repository ABAS to be preferable to the Data Indirection ABAS for minimizing the impacts of the anticipated modifications. We do not need to use the Publish/Subscribe sub-ABAS, since we do not anticipate any temporal ordering issues in the changes to the Sea Buoy’s data producers and consumers. If such concerns did arise in the future, the Publish/Subscribe sub-ABAS could be substituted here. The resulting software architecture, once the Abstract Data Repository has been added, is depicted in Figure 7.

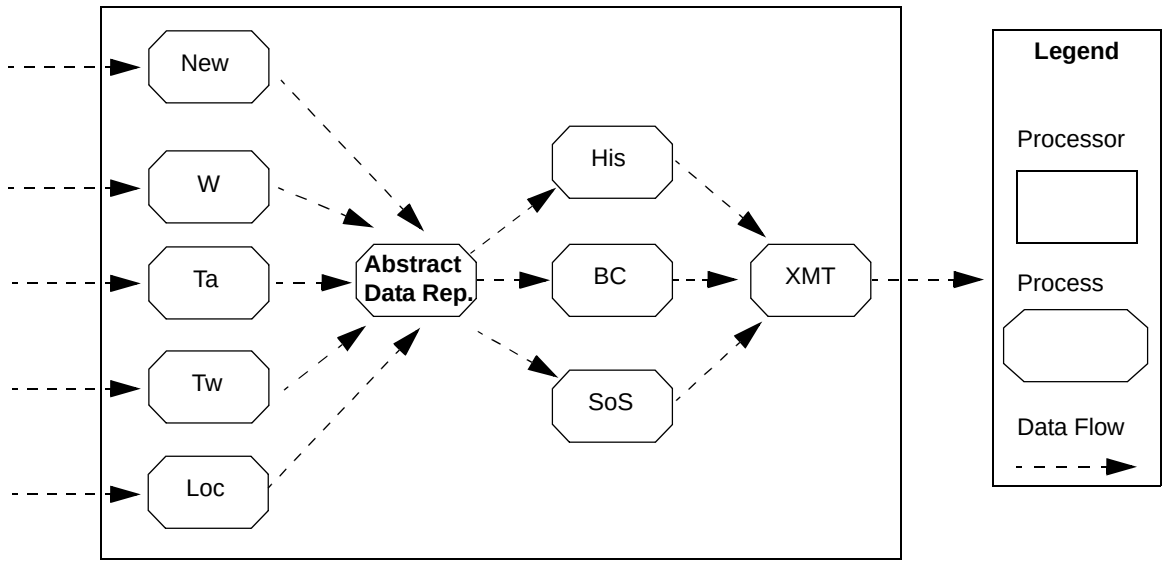

Figure 7: Concurrency View After Adding the Abstract Data Repository ABAS

While the use of the Abstract Data Repository ABAS would have favorable effects on the ripple effects of a new data format, introducing such a conversion might have an effect on the execution time of the process. We know that the latency functions discussed above depend on execution time. Adding new sensors will also create an additional execution time load. These are *tradeoff points*, architectural decisions that affect both modifiability and performance [Kazman 99]. Such architectural decisions have to be made carefully, with both modifiability and performance requirements in mind. ABASs give us a way to rigorously reason about these critical points in the architecture.

To notice tradeoffs—interaction across ABASs of different attribute types—you must

- be aware of how architectural decisions affect each quality attribute in isolation as exemplified by the discussion of latency functions in Section 4.3
- be disciplined about looking for situations in which the changes to an architecture from the point of view of one attribute affect the independent parameters of functions associated with other attributes. The analysis section of each ABAS gives guidance here by identifying the known sensitivities and tradeoffs.

Folding in Availability Requirements

While there were no explicit reliability requirements in the initial description of the Sea Buoy problem, it seems natural to assume that the availability of the SOS function is critical.

Appealing to the *Simplex ABAS* (detailed in Appendix F) reveals analytic redundancy as a strategy for achieving high levels of availability. To exemplify the notion of analytic redundancy, you can think of the relationship between power steering and mechanical steering in a car as analytically redundant. Both mechanisms have the same effect on the environment, that is, they change the direction of the wheels, but the mechanisms used and how they perform are different. Power steering is less reliable (since it depends on the car's power) but is easier to use; mechanical steering is more reliable (only depending on mechanical linkages), but is much harder to use.

Analytically redundant SOS mechanisms could be employed in this example, as shown in Figure 8. In this case, a separate hardware unit could be used to generate and transmit a simple SOS signal, the premise being that a relatively simple, specialized hardware unit is more reliable (and easier to verify) than the on-board processor. In addition to a highly reliable analytically redundant SOS component, the Simplex ABAS also calls for a highly reliable "Decision and Switch" component. This component implements rules for deciding if an output is valid and deciding what to do if an output is invalid. For the Sea Buoy problem, we have decided to simply broadcast both the simple and more comprehensive SOS signals, thereby eliminating the need for the decision and switch component. Note that the redundant SOS component also has its own transmission capability.

The simple SOS hardware serves as a backup to the more sophisticated signaling mechanism that runs on the on-board computer, which delivers a more comprehensive SOS message (potentially including position, weather information, time, etc.). Including this more comprehensive SOS mechanism is a tradeoff between performance and availability. The on-board SOS function will be invasive on the performance of the other Sea Buoy functions, as it runs at a high priority. The higher performance demands of the SOS transmission are presumably justified in their effects on the other components, but this assumption, now that it has been made explicit, needs to be checked with the system's stakeholders.

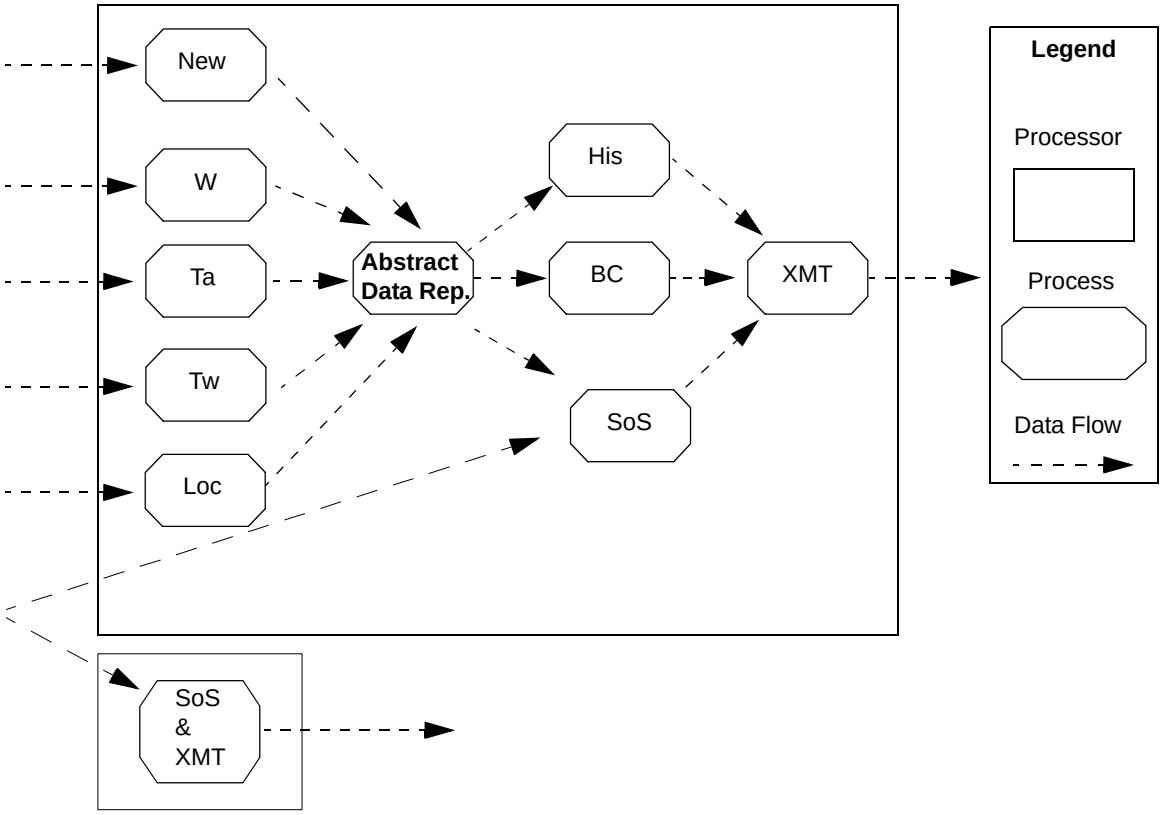

Figure 8: Concurrency View After Adding Simplex-Style Redundancy

4.5 Summary

Using ABASs in design causes one to think about the process of designing differently. ABASs do this by facilitating both the *generation* of design alternatives (by choosing ABASs based upon attribute-specific requirements) and the *discrimination* between design alternatives (based upon the reasoning associated with each ABAS). As illustrated by the above example, we choose the pieces of the design based upon matching both topological requirements (for example, the data store is a shared resource and this topology is inherent in the problem) and quality attribute requirements, which come from the system's stakeholders. An important part of using ABASs for system design and analysis is knowing how to compose them. This report does not give a cookbook; no such cookbook is possible at this point in the field's maturity, nor is it possible even in mature fields except in well understood specialized subdomains. Instead, it gives a way of thinking about design and the interactions of design decisions on different quality attributes.

5 ABASs in Analysis

Up to now, we have concerned ourselves with using ABASs to aid in the design of architectures for large, complex systems. However, we have also used ABASs extensively in analyzing existing systems as part of the Architecture Tradeoff Analysis Method (ATAM) [Kazman 99]. This should not come as a surprise to the reader, given our focus on analysis as the important distinction between existing architectural styles and ABASs. Since every ABAS is associated with an analytic model, whether formal or informal, we can use these models to understand the ramifications of architectural decisions that have already been made. ABASs can support two levels of analysis: a quantitative analysis in which formal reasoning and mathematical models are used, and a less formal qualitative analysis in which probing questions and rules of thumb are used. Qualitative analysis has proven to be very useful for the ATAM process, since time pressures typically forbid the building of detailed models.

The ATAM process consists of the following steps.

1. Present ATAM.
2. Present business drivers.
3. Present architecture.
4. Identify architecture styles.
5. Generate quality attribute utility tree.
6. Elicit and analyze architecture-styles.
7. Generate seed scenarios.
8. Brainstorm and prioritize scenarios.
9. Map scenarios onto styles.
10. Present out-brief.

Steps 5 and 6 are most relevant to the use of ABASs in analyzing existing architectures. In step 5, we elicit the quality attributes that are most critical to the system's success, and have the system's stakeholders prioritize these attributes and provide examples of them as scenarios. Using this information, we can then probe the architecture for the styles that are used to satisfy the attribute-specific requirements. Once these ABASs have been identified in the architec-

ture, the analytic models associated with them can be applied to understand the ramifications of architectural decisions on the quality-attribute goals.

For example, if a quality-attribute goal is for an architecture to be insulated from changes to the Object Request Broker (ORB) middleware, one possible way of realizing that goal in the architecture is to create a middleware abstraction layer. After that, have all of the application-specific code use the services presented by that layer, rather than directly accessing the interfaces provided by the ORB. If the system's architect identifies layering as a strategy used by the architecture to realize this modifiability goal, we can immediately make use of Layering ABAS (detailed in Appendix E). The analysis portion of this ABAS gives us a set of tools for asking questions of the architecture to determine the implications of a change to the middleware on the remainder of the system.

In this way, ABASs provide the foundation for an efficient architecture analysis technique by providing a set of pre-packaged analyses and questions for the architect, based upon both known solutions to commonly recurring problems and known difficulties in employing those solutions.

6 Conclusion

In this report, we have introduced the idea of an ABAS, pre-packaged units of architectural design and analysis. What separates ABASs from existing catalogs of architectural styles and design patterns is the emphasis on *analysis*. In each ABAS, we link analytic models (styles of reasoning) to topological architecture patterns. Because of our focus on analysis, the different responses to specific measurable aspects of quality attributes are the distinguishing characteristics between competing designs. ABASs give you, as an architect, a way to reason about design in direct relation to the quality attributes that you want to achieve.

Thus, architectural design using ABASs is different from ordinary design. Rather than designing first from principles or by limiting reasoning to topology we envision a process where a designer considers the problem's constraints, its inherent topology (for example, some computational environments are inherently distributed), and the quality attribute responses that must be controlled for the system to be successful. From these considerations, the designer can then look up appropriate ABASs from the handbook, indexing into the handbook by attribute responses and topology. The design and reasoning effort thus becomes one of applying the existing styles and their analytic models, rather than designing from scratch. There is still design work to do: combining ABASs requires careful thought and reasoning. Knowing how to adapt ABASs to similar but not identical environments also requires careful attention. The bulk of the design and analysis work is ready for the architect to reuse wholesale, and this, we believe, is an enormous benefit. We have already reaped this benefit in practice many times by using ABASs in analyzing software architectures.

In this report, we have attempted to show how ABASs can be used to facilitate both design and analysis. But our experience with using ABASs is still limited to a small number of design and analysis experiences. Our greater goal in writing the report is, on analogy with the design patterns community, to stimulate the community to generate a large number of ABASs covering every known quality attribute. The seven ABASs that we present here are just the beginnings of what we hope will be a larger effort that will support the creation of a true engineering discipline of architectural design.

Example ABASs

The purpose of this appendix is to offer a small but representative set of ABASs to illustrate the concept from the points of view of several different quality attributes. This is not meant to be a “write-only” appendix; it should be read with Chapter 4.

There are six ABASs in the appendices. The first one, Synchronization ABAS, is a performance ABAS concerned with multiple processes that need to synchronize to ensure mutually exclusive access to a resource. This is followed by four modifiability ABASs. The first is the Data Indirection ABAS which addresses a key style for affecting modifiability. This style uses either an explicit or implicit data repository to obviate the need for producers and consumers of data to have direct knowledge of each other. The second and third modifiability ABASs are variants of Data Indirection: the Abstract Data Repository ABAS and the Publish/Subscribe ABAS. The fourth modifiability ABAS is the Layering ABAS, which is a classical style for providing cumulative layers of abstraction on top of some base of functionality. The last ABAS in this appendix is an availability one, known as the Simplex ABAS, that uses a special form of redundancy to achieve high levels of availability.

Appendix A Synchronization ABAS

A.1 Problem Description

For the Synchronization 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 and 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—a deadline.

The interesting feature of the Synchronization ABAS is that this requirement must be met in the face of contention for a *shared resource*, for which the multiple processes must compete. The central property of this ABAS for reasoning purposes is: how contention for the shared resource is handled, and how this contention affects performance, in particular, end-to-end latency. The Concurrent Pipelines ABAS discussed in the body of this report is a related ABAS. The Concurrent Pipelines ABAS focused on multiple processes contending for a single processor using a preemptive scheduling discipline, whereas this ABAS focuses on multiple processes requiring mutually exclusive access to a shared resource such as shared data.

A.1.1 Criteria for Choosing this ABAS

This ABAS will be relevant if your problem inherently has real-time performance requirements and consists of multiple processes on a single processor that share a resource.

A.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.

A.3 Architectural Style

The synchronization style is shown in Figure 9 in a concurrency view mapped onto a hardware view. In this ABAS, there is a single processor and a set of processes with the associated known (or estimated) properties, listed in Table 2, that are transforming input streams into output streams. Some of these processes need to synchronize to share a resource controlled by S, the “server” process.

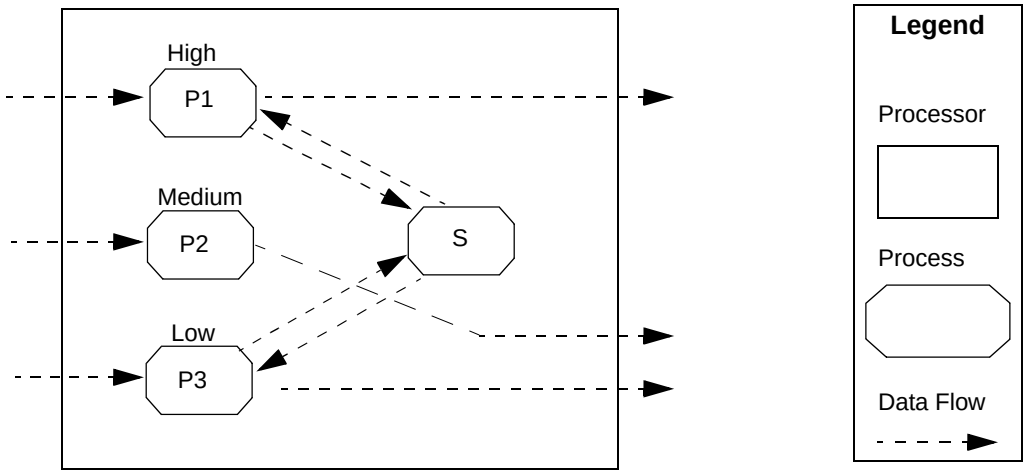

Figure 9: Synchronization ABAS

Table 2 describes the architectural decisions that are necessary for creating an analytic model of end-to-end, worst-case latency for this ABAS.

Performance Architectural Parameters
topology: *star*
preemption policy: *priority based*
execution time for each process associated with processing each input: C_i
period associated with each process: T_i
scheduling discipline: *fixed priority*
synchronization protocol including: • *the queuing discipline (e.g., FIFO or priority) for the server process* • *how the priority is managed during the critical section (e.g., the section of code during which other processes are locked out)*

Table 2: Architectural Decisions for the Synchronization ABAS

A.4 Analysis

In the following sections, 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 listed in Table 2; 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.

A.4.1 Reasoning

There are three types of time "experienced" by an arbitrary process under these circumstances: preemption, execution, and blocking time. Preemption time is the contribution to latency attributable to higher priority processes. Blocking is the contribution to latency due to low priority processes. Blocking time arises as a consequence of the shared resource topology. 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 worst-case latency for process i, assuming that processes 1 through i-1 are of higher priority, can be found by iterating the following formula until it converges (that is, the value of L_n remains the same for two consecutive iterations).

$$L_{n+1} = \sum_{j=1}^{i-1} \left\lceil \frac{L_n}{T_j} \right\rceil C_j + C_i + B_i \qquad (4)$$

This equation is a reflection of the architecture decisions shown in Table 2. 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. With each iteration, the formula accumulates the execution time associated with each of these preemptions, adds in the execution time of process i (C_i) and adds in the blocking time (B_i). As stated above, the blocking terms capture the effects of lower priority processes. The blocking time is directly affected by the synchronization protocol that is used.

Given an initial value of L_n for C_i, iterating until L_n equals L_{n+1} results in a determination of 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 of potential timing problems. The lack of convergence signals an unbounded latency. Equation 4 illustrates the potential sensitivity of latency to higher priority processes and blocking time.

A.4.1.1 Priority Assignment

One potential pitfall in the synchronization ABAS 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 processes 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 its deadline and yet the utilization of this set of processes is only 20%. The deadline monotonic [Leung 82] priority assignment is a strategy for avoiding this problem in the context of fixed-priority scheduling. This strategy assigns priorities as a monotonic function of deadline, with the shortest deadline receiving the highest priority.

A.4.1.2 Priority Inversion

This ABAS is a classical situation in which priority inversion and potentially unbounded priority inversion can arise. Consider the following situation. Assume that process P1 has a high priority; P2 has a medium priority; and P3 has a low priority (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 happen easily 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 by adding other medium priority processes. *This problem illustrates the sensitivity of the high priority process's latency to the priority of the server process*.

A.4.1.3 Blocking Time

While blocking time is a seemingly innocuous term in Equation 1, blocking is a significant and sometimes insidious contributor to latency. In general, blocking occurs whenever an architectural design permits a low priority process to execute when a higher priority process is also ready to execute.

In some cases, blocking is unavoidable. In other cases, such as with the priority inversion discussion, blocking can be managed much more effectively. Several common sources of blocking are listed below:

- **Critical section:** As discussed above, a critical section is a source of blocking. An improper server priority can result in an unnecessarily large amount of blocking. The key to circumventing unbounded 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 such prevention technique is to set the priority of process S to be at least as high as the highest priority client process. Another technique is to use a *priority inheritance protocol* [Rajkumar 91], which raises the priority of S to the highest priority process that is blocked waiting for the services of process S.
- **Deadlock:** Deadlock is an extreme form of blocking in which processing comes to a halt. It occurs when two or more processes need mutually exclusive access to two or more of the same resources. Deadlock is discussed in most books on operating systems.
- **FIFO:** A first-in-first-out queue is another common source of blocking. If a high priority process is stuck in a FIFO queue behind a lower priority process, the blocking time can be arbitrarily long.
- **Non-preemptable section:** Sections of lower priority processes that are non-preemptable can delay a higher priority process, since the high priority process is prevented from preempting if it is time for it to execute and the lower priority process is in its non-preemptable section.

- **Interrupt:** Strictly speaking, interrupts are not a source of blocking, but rather a source of preemption. However, often a lower priority thread is initiated by an interrupt. Since the interrupt is executing on behalf of the lower priority process, it can be viewed as a source of blocking to other higher priority processes.
- **Threads and processes:** Some operating systems support threads, which are lightweight units of concurrency that execute within a single, shared address space (whereas each process executes in its own address space). Sometimes in this situation, a two-level scheduler is used. In other words, processes are first scheduled, and then threads within the process are scheduled. A thread's high priority can be virtually ineffective if the thread resides in a process that has been assigned a relatively low priority.

A.4.2 Analysis and Design Heuristics

Even if one does not build a formal analytic model of the latency in a synchronization ABAS, 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
  - How does your choice of priority assignment impact latency?
  - Is there another prioritization strategy that might reduce latency?

- Sources of blocking
  - Has blocking been accounted for in estimating latency?
  - Are there sources of blocking in black-box components that have not yet been accounted for?

- Priorities used during the critical section
  - Does an unbounded priority inversion situation exist?
  - Can blocking time be reduced by using a different technique for managing the priority of a critical section?

Appendix B Data Indirection ABAS[1]

B.1 Problem Description

This ABAS is characterized by keeping the producers and consumers of shared data from having knowledge of each other's existence and the details of their implementations. This is accomplished by interposing an intermediary—a component and/or protocol—between the producer and consumers of shared data items. The general principle at work here is that modifiability is enhanced by reducing the data or control coupling between distinct components. In this ABAS, coupling is reduced by having the intermediary—typically a shared data repository—coupled to both the producers and consumers, and hence having them decoupled from each other.

B.1.1 Criteria for Choosing this ABAS

This ABAS will be relevant if you anticipate changes in the producers and consumers of data, including the addition of new producers and consumers. If these changes are frequent and pervasive enough to warrant concern about the cost of modification, this ABAS is relevant.

B.2 Stimulus/Response Attribute Measures

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

- **Stimuli**:
  - a new producer or consumer of data
  - a modification to an existing producer or consumer of data
  - a modification to the internals of the data repository
- **Response**: the number of components, interfaces, and connections added, deleted, and modified, along with a characterization of the complexity of these changes/deletions/modifications

1. Appendices C and D are sub-ABASs of this ABAS.

B.3 Architectural Style

Figure 10 shows the generic structure of this ABAS. Its topology is a star, with the repository located at the center and the producers and consumers at the periphery.

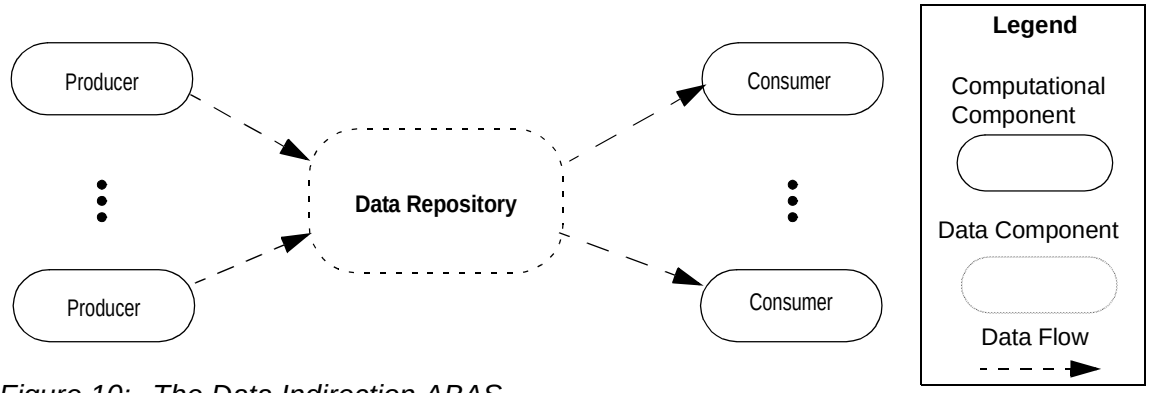

Figure 10: The Data Indirection ABAS

The data repository can be a location that is known to both producers and consumers (e.g., a file or a global data area) or it can be a separate computational component (a blackboard that is hosted in a separate process, potentially even on a separate computer). The only constraint on the repository is that it can hold data. The data repository is a persistent data store such as a shared data area in memory, a file, or a database.

In addition to the repository, there are some number of data producers and some number of data consumers. A single component may be both a producer and a consumer of data. The repository has a specified data layout—a file structure, or schema—and a set of data types that are known by all producers and consumers. This layout remains consistent during a single execution of the system. The producers place their data in the repository by virtue of the fact that they know the details of the repository's layout; the consumers similarly retrieve data from the repository. The issues of how performance and concurrency control are managed (e.g., the policy and mechanisms for determining who gets to update the repository and when) are outside the scope of this ABAS.

The components can be independent processes on the same processor or different processors. They could also be bundled together in a single process. Thus, there are no restrictions within this ABAS on the run-time packaging of the components. For those considerations and their implications, we would turn to a performance ABAS such as the Synchronization ABAS shown in Appendix A.

Table 3 discusses the architectural parameters and decisions of concern for this ABAS.

Modifiability Architectural Parameters
topology: *star*
persistence of data: *persistent*
client knowledge of data schema: *complete knowledge*
activeness of repository: *passive*

Table 3: Architectural Decisions for the Data Indirection ABAS

The activeness of the repository, persistence of data, and client knowledge of the data schema are design decisions that allow us to differentiate between several sub-styles of the Data Indirection ABAS including the Abstract Data Repository and Publisher/Subscriber ABASs described in Appendix C and Appendix D, respectively.

B.4 Analysis

To measure and more importantly, to *predict* the impact of the modifications, we have two choices of analytic techniques: walkthroughs or metrics. With specific metrics, we can assess the coupling and cohesion of a layered architecture which should predict the average transitive closure of changes and hence the average difficulty of changes. With walkthroughs, we can *directly* assess the effects of a set of anticipated changes on an architecture. In both cases, these techniques are an attempt to associate the stimuli (in this case, a set of changes to the architecture) with the difficulty of the changes. This difficulty is couched in terms of the amount of work needed to add, delete, or modify the affected components, connectors, and interfaces.

The goal of a modifiability walkthrough is to understand how well the style reacts to various types of modifications. That is, we want to know, for some representative set of typical changes to the system, whether the effects of those changes are localized or “ripple” to other components. Rippling is a consequence of implicit and explicit dependencies between components. Examples of such dependencies include the following:

- producers needing to know about the existence and names of consumers
- assumptions about the types of data and their relationships (e.g., data about a family includes a mother, father, and up to ten children; ages are represented by three digits and are positive integers)

- assumptions about the meaning of the data (e.g., temperature data is in degrees Fahrenheit versus Celsius or Kelvin)
- internal representation of data (e.g., data is stored in an array versus a linked list)

Because the clients in the Data Indirection ABAS have complete knowledge of the data schema, they are affected by changes made to the repository. Hence this ABAS is particularly vulnerable to rippling, as we will see in the next section.

B.4.1 Reasoning

In this ABAS, the producers and consumers of shared data are sharing assumptions about the data repository, via its schema/layout. That is, the producers and consumers need to agree upon the types, meanings, and organization of the data that they share, or else they cannot communicate just as humans cannot communicate unless they share a common lexicon, syntax, and semantics. This is a constraint of the style and a potential source of rippling. The data repository in this ABAS can, however, obviate the need for producers to know of the name, or even the existence, of particular consumers. Producers and consumers can simply exchange information via the data in the repository, which is the strength of this style.

For this ABAS, the architectural parameter of greatest interest for analysis is “client knowledge of data schema.” If the client has complete knowledge of the data schema, there is no abstraction being employed. If an abstraction interface has been defined that masks file layouts during reads and writes or via database access methods such as SQL queries, some aspects of the data schema (such as its layout in memory or the internal data types and structures employed) may be changed without changing the clients. We will discuss a different setting of this parameter in the Abstract Data Repository ABAS in Appendix C.

When we analyze instances of the Data Indirection ABAS, we want to understand how changes to the data repository, data producers, and data producers will ripple throughout the remainder of the system. We want to measure the extent to which these components are coupled.

While we cannot always measure coupling directly, we will suggest in our analysis here (and in all modifiability ABASs) ways of approximating appropriate measures of coupling. The analysis technique for this ABAS is to investigate a representative set of anticipated change scenarios [Kazman 94]. These scenarios form a palette of types of changes that can occur to this ABAS, and you can evaluate your own instances of this ABAS by determining the extent to which your anticipated changes fall into each category.

For the purpose of this analysis, we assume five scenarios. The following discusses each in detail:

1. adding a new consumer of data
2. adding a new producer of an existing data type
3. adding a new producer of a new data type
4. changing the internal representation of an existing data item
5. deleting an existing data type

We will consider each of these scenarios in turn.

Scenario 1: Adding a New Consumer of Data

Scenario 1 affects only one place in the architecture; it requires the creation of a new component (the consumer) and a new instance of an existing connector. The new consumer would simply need to directly access the data repository as other consumers do. The change does not affect any of the other consumers or any producers, as indicated in Figure 11, where new components are shown shaded.

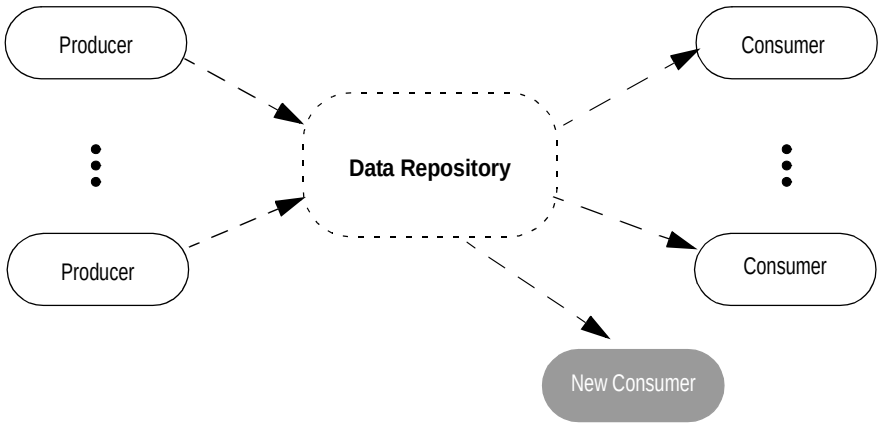

Figure 11: Adding a New Consumer

Scenario 2: Adding a New Producer of an Existing Data Type

On analogy with Scenario 1, Scenario 2 causes the creation of a single new component, as illustrated in Figure 12. Adding a new data producer of an existing data type only affects the new producer, which will access the data repository using the same techniques as other producers and consumers—that is, via detailed knowledge of the data repository's layout. Other producers and consumers of the data type are unaffected.

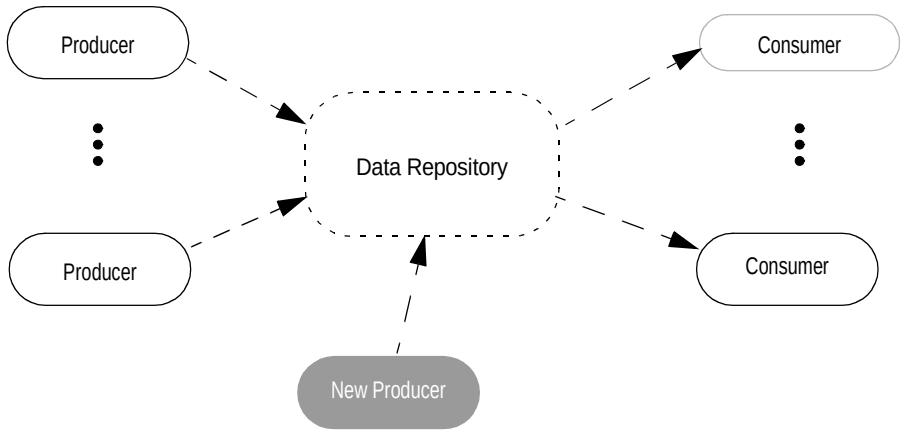

Figure 12: Adding a New Producer of an Existing Data Type to the Data Repository

Scenario 3: Adding a New Producer of a New Data Type

In this scenario, a new data type is being added to the repository. Assuming that this data type is being added because some consumer of the data needs it, three changes are needed: the new producer must be created, the repository must be modified to accept the new type, and either a new consumer must be created or an existing consumer must be modified. This is shown in Figure 13. Modified components are indicated with diagonal stripes.

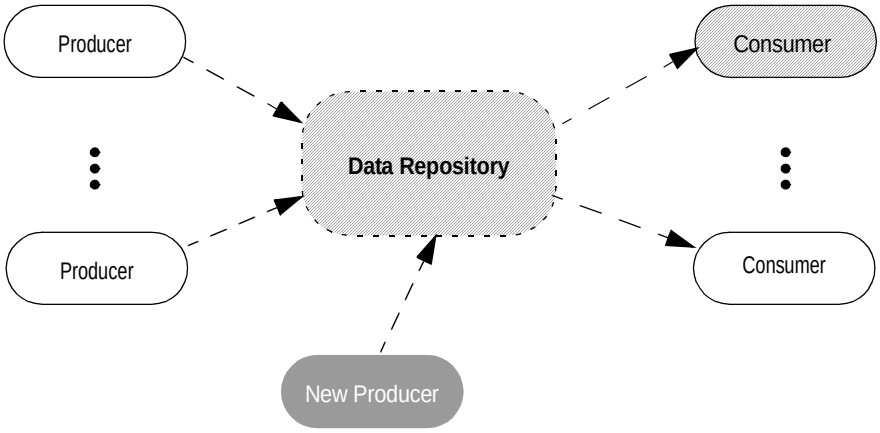

Figure 13: Adding a New Producer of a New Data Type to the Data Repository

We only show a single consumer being affected. However, if the new data is added in such a way that it affects the data access of other producers and consumers (for example, if data is accessed based upon a known position in an array and this new data changes some array indices), the change will ripple to additional producers and consumers. The key observation here is that while typically only the consumers that are *interested* in the data will be affected, if the data repository is not managed carefully (for example, if new array data is added anywhere in the array) then changes may have considerably greater rippling effects.

Scenario 4: Changing the Internal Representation of an Existing Data Item

Under the assumption that there is at least one producer and one consumer of any data type in the repository, then scenario 4—changing the internal representation of a data item—also causes three changes, as shown in Figure 14. As with scenario 3, we assume here that this change does not affect the layout of the other parts of the data repository.

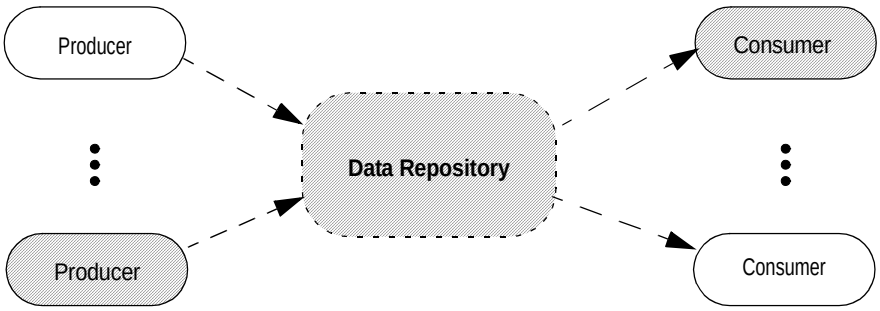

Figure 14: Changing the Internal Representation of a Data Item in the Data Repository

Scenario 5: Deleting an Existing Data Type

"Delete an existing data type" could have several meanings. One is that data of that type is no longer being produced. Obviously this affects both the producer and consumer of this type. However, this doesn't necessarily have to affect the data repository; it could simply continue supporting this as an unused data type.

"Deleting an existing data type" could also mean deleting it from the repository, affecting the data repository as well as selected producers and consumers. This is the version of the scenario that we have shown in Figure 15. This realization of the scenario is more in keeping with good software engineering practice, since otherwise over time the repository could contain many unused data areas, complicating its understandability and hence its maintainability.

This realization, while maintaining good software engineering practice, has the potential immediate problem that removing the data type may affect the layout of the entire repository and hence cause this change to ripple to other components which are neither consumers nor producers of the data type.

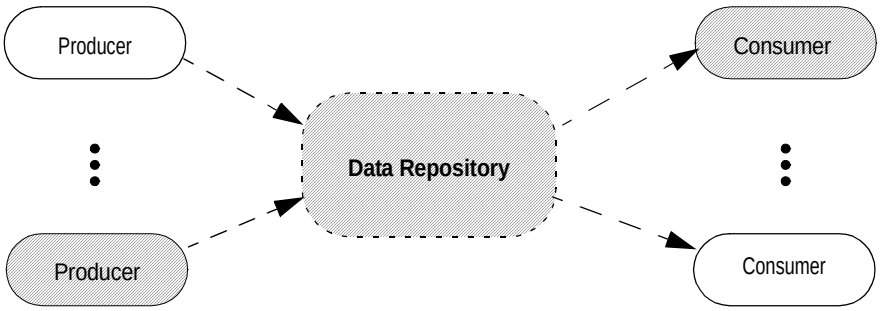

Figure 15: Deleting a Data Type

B.4.2 Analysis and Design Heuristics

You should choose this ABAS if you don't expect to add, delete, or change the data types used in the repository frequently. If, for example, your most frequently anticipated change is to add a new consumer of an existing data type, this ABAS will be simple to implement (since you have not had to go to the additional trouble of defining an abstraction interface) and will easily support your anticipated palette of modifications.

If you are adding new data to the repository, you should add it in such a way that it does not affect the access of existing data. This could mean, for example, adding new data to the end of an array, if arrays are used. Similarly, when removing a data type from the ABAS, the data type could be left in place as an unused field in the repository, or it can be removed, potentially causing ripple effects to other data types and their producers and consumers.

If you expect to change the data types relatively frequently, then you should consider a sub-ABAS such as the Abstract Data Repository or the Publish/Subscribe ABAS which, while more complex to implement, will shield you from the ripple effects that are inherent when producers and consumers have intimate knowledge of the data repository's layout.

Appendix C Abstract Data Repository Sub-ABAS

C.1 Problem Description

This sub-ABAS extends the Data Indirection ABAS by not only keeping the producers and consumers of shared data from having knowledge of each other's existence and the details of their implementations, but also by keeping the details of the shared data repository's implementation a secret from the producers and consumers. This secret is embodied in an abstract interface to the data repository. This abstract interface further reduces the coupling between the data producers and consumers above and beyond what the Data Indirection ABAS does.

C.1.1 Criteria for Choosing this ABAS

This ABAS should be chosen if you anticipate that the data producers will change the internal format of the data they produce. The consequences of such changes, if left unchecked, will ripple to the data repository, and hence to any consumers of the data. In addition, such changes might affect the layout of the data repository in non-local ways, thus affecting producers and consumers of other data if there were not an abstract interface. Using this ABAS will protect data producers and consumers from changes to each other and to the underlying repository. This comes at a slight performance cost, however, since the abstraction interface is a level of indirection away from the data itself, and so each data access typically incurs at least the overhead of traversing this extra layer of software.

C.2 Stimulus/Response Attribute Measures

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

- **Stimuli:**
  - a new producer or consumer of data
  - a modification to an existing producer or consumer of data
  - a modification to the internals of the data repository

- **Response**: the number of components, interfaces, and connections added, deleted, and modified, along with a characterization of the complexity of these changes/deletions/ modifications

C.3 Architectural Style

Figure 16 describes the architectural structure for the Abstract Data Repository sub-ABAS. In this ABAS there are some number of data producers and some number of data consumers. A single component may be both a producer and consumer of data. These producers and consumers know nothing of each other's existence, communicating solely through the "methods" that the abstract data repository's interface provides. These methods control the creation, deletion, modification, and retrieval of data from the repository. The producers send their data to a abstract data repository, but have no insight as to how or where this data is stored or retrieved. The repository holds this data for consumers who can access it via the provided methods, again, with total ignorance of the underlying layout and algorithms of the data in the repository. The abstract data repository stores the produced data until it is explicitly deleted.

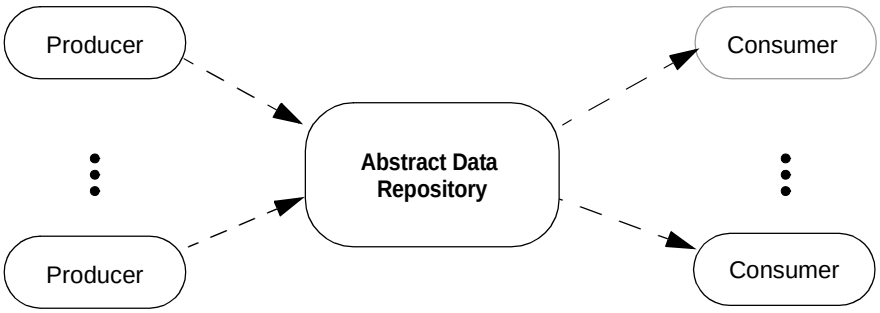

Figure 16: The Abstract Data Repository Sub-ABAS

The architectural style of this sub-ABAS inherits its architectural decisions from the Data Indirection ABAS. However, being a sub-ABAS, some of the architectural decisions are set differently. You can think of the relationship between ABASs and sub-ABASs as one of inheritance. Thus, the style is the same unless some parameter is explicitly overridden in the subtype.

Table 4 describes the architectural parameters and decisions of concern for the Abstract Data Repository sub-ABAS.

Modifiability Architectural Parameters
topology: *star*
persistence of data: *persistent*
client knowledge of data schema: *no knowledge*
activeness of repository: *passive*

Table 4: Architectural Decisions for the Abstract Data Repository Sub-ABAS

C.4 Analysis

As with the Data Indirection ABAS, the architectural parameter of greatest interest for analysis here is "client knowledge of data schema." Unlike the Data Indirection ABAS, the Abstract Data Repository sub-ABAS sets this feature as "no knowledge." The critical ramification of this ABAS feature is that data producers and consumers are kept ignorant of the specific ways in which the data repository is managing its data.

C.4.1 Reasoning

We will now consider how the Abstract Data Repository sub-ABAS reacts to the scenarios listed for the Data Indirection ABAS.

We will only indicate ways in which this sub-ABAS differs from the Data Indirection ABAS. In this case, the Abstract Data Repository sub-ABAS reacts identically to Scenarios 1, 2, 3, and 5. It reacts differently to Scenario 4.

Scenario 4: Changing the Internal Representation of an Existing Data Item

Unlike the Data Indirection sub-ABAS, this change has minimal effects on the Abstract Data Repository sub-ABAS because the producers and consumers do not directly access the repository. So, for example, a data producer can change how it is representing some data type, but this change will not ripple to any other component in the system, assuming that the existing interfaces to the repository are left unchanged.

The ramifications of this change are indicated in Figure 17. As you can see, only the data producer itself needs to be changed.

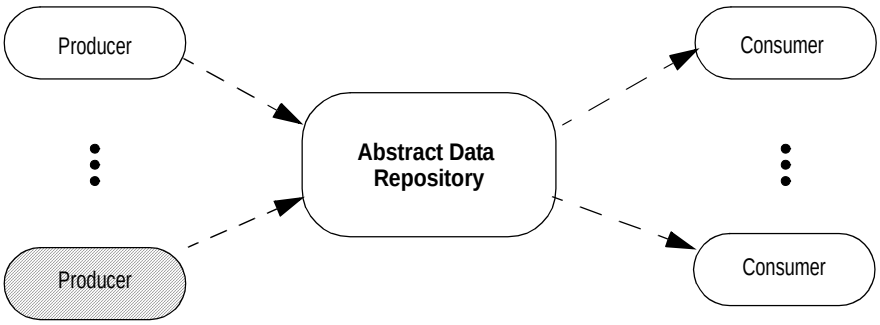

Figure 17: Changing a Data Item's Internal Representation in the Abstract Data Repository Sub-ABAS

C.4.2 Analysis and Design Heuristics

You should choose this ABAS if you expect to be frequently changing the producers and/or consumers of data, but not the temporal ordering of producers and consumers (that class of change is dealt with in the Publish/Subscribe sub-ABAS in Appendix D). For example, if you have many possible sources of a particular data item, or if you anticipate needing to interoperate with new sources of similar kinds of data over the system's lifetime, this ABAS will insulate you from the effects of these changes. This is because the sources of the data may change, but the changing specifics of how this data is collected and represented do not ripple through the data repository to the consumers.

There is typically a performance implication of using an abstraction interface, given that this interface adds a layer of indirection to the interaction between producers/consumers and the data repository. While this overhead is typically not large, in systems with tight performance deadlines, or where the repository is heavily used, it might be a significant factor and should not be overlooked.

Appendix D Publish/Subscribe Sub-ABAS

D.1 Problem Description

This sub-ABAS aids in automatically synchronizing the state of data producers and consumers. The data producers are called “Publishers,” and the consumers are called “Subscribers.” When a Publisher publishes a new piece of data, all subscribers are notified and automatically receive the data. That is, the repository does not simply passively accept requests to store and retrieve data but rather keeps track of which data consumers are interested in which data items and informs them when new relevant data has been deposited in the repository. This sub-ABAS extends the Data Indirection ABAS and the Abstract Data Repository sub-ABAS by not only keeping the producers and consumers of shared data ignorant of each other and of the details of the shared data repository’s implementation, but by making the repository *active* and the data in the repository *transient*.

D.1.1 Criteria for Choosing this ABAS

This ABAS should be chosen if you anticipate that all of the following conditions will be true:

- The data producers will change the format of the data that they produce.
- The number and identity of producers and/or consumers of a particular data item is unknown or is likely to change.
- The temporal ordering between producers and consumers of data is either unknown or subject to frequent changes.
- There are no complex temporal dependencies or tight, real-time deadlines associated with the production and consumption of data.

The third and fourth points are the crucial ones for this sub-ABAS. The Publish/Subscribe sub-ABAS dissociates producers and consumers of data in terms of their identities (as with the Data Indirection ABAS), their locations, the mutual knowledge of their existence, and the internal format of their shared data (as with the Abstract Data Repository sub-ABAS), and the temporal control between data producers and consumers.

This sub-ABAS keeps producers and consumers synchronized in terms of the state of their data, but this synchronization is not controlled tightly by the producers or consumers. Thus, this ABAS is typically inappropriate for applications where complex temporal dependencies exist (e.g., this data item must be processed by consumer B exactly 0.05 seconds after it is processed by consumer A), or where hard, real-time deadlines exist (e.g., all data from each frame must be processed in 0.1 seconds).

D.2 Stimulus/Response Attribute Measures

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

- **Stimuli**:
  - a new producer or consumer of data
  - a modification to an existing producer or consumer of data
  - a modification to the internals of the data repository
  - a change in the timing of data production and consumption

- **Response**: the number of components, interfaces, and connections added, deleted, and modified, along with a characterization of the complexity of these changes/deletions/ modifications

D.2.1 Architectural Style

The architectural style for this ABAS is shown in Figure 18. The component types are data producers (publishers) and consumers (subscribers). A single component may be both a producer and consumer. The connections between the components are shown as data flow, but in reality, this is often realized as both data and control flow (i.e., via a procedure call, remote procedure call, or remote method invocation). Although the subscription manager is shown as a separate component here, in practice it is frequently implemented in a distributed fashion, as a set of services that a data producer provides to data consumers. For example, in object-oriented systems, producer and consumer objects can inherit from a base class that provides basic subscription and notification services [Gamma 95].

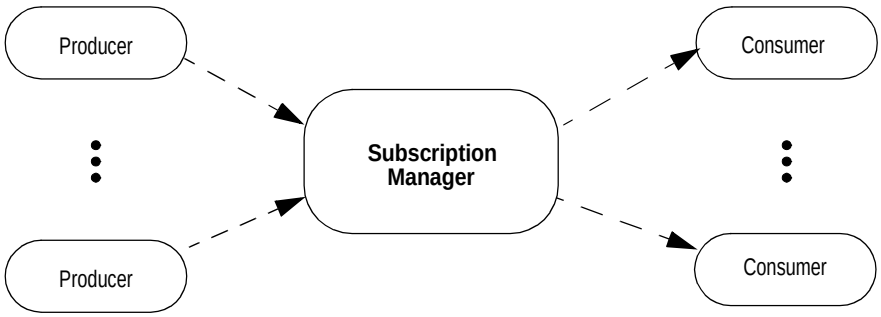

Figure 18: The Publish/Subscribe Sub-ABAS

The architectural style of this sub-ABAS inherits its architectural decisions from the Data Indirection ABAS. However, being a sub-ABAS, some of the architectural decisions are set differently. Table 5 describes the architectural decisions of concern for the Publish/Subscribe sub-ABAS. The changed parameters of note are the "persistence of data," the "client knowledge of data schema," and the "activeness of the repository."

Modifiability Architectural Parameters
Topology: *star*
Persistence of data: *transient*
client knowledge of data schema: *no knowledge*
activeness of repository: *active*

Table 5: Architectural Decisions for the Publish/Subscribe Sub-ABAS

D.3 Analysis

One of the salient features of this ABAS is that consumers are informed of changes to data in which they have registered an interest. Thus, there is a registration process whereby consumers subscribe to a named set of data, and it is this registration list that the subscription manager uses to inform the appropriate subset of the existing consumers. There may be multiple consumers interested in a particular data item in which case the subscription manager must have some rules to determine the order of notification. This notification may be synchronous or asynchronous. There may even be no consumer interested in a particular data item. The producer can thus remain completely unaware of if and how its data is being consumed. For anal-

ysis, this leads to a consideration of how a change in the ordering of data production and consumption will affect this ABAS.

D.3.1 Reasoning

We will now consider how the Publish/Subscribe sub-ABAS reacts to the scenarios listed for the Data Indirection ABAS.

We will, for the moment, only indicate ways in which this sub-ABAS differs from the Data Indirection ABAS. In this case, the Publish/Subscribe sub-ABAS responds the same to Scenarios 1, 2, 3, and 5 and responds differently to Scenario 4, in that changing a data item's internal representation affects only the producer of the data. These responses are identical to the Abstract Data Repository sub-ABAS.

We need to introduce a new scenario to distinguish the Publish/Subscribe sub-ABAS from the Abstract Data Repository sub-ABAS. This scenario is based upon the new stimulus that this ABAS responds to:

- a change in the rate of data production and consumption

We will now consider the implications of this stimulus on the Publish/Subscribe sub-ABAS.

Scenario 6: Changing the Timing of Data Production and Consumption

Consider a concrete example of this scenario: a process control system that includes a data producer that produces temperature reports. These reports might be received by two subscribers: one that controls a furnace and one that shows the current temperature to an operator. The response to this scenario is quite simple in the Publish/Subscribe sub-ABAS. The producer simply changes the times at which it produces data. The coordination of the production and consumption of data is handled by the subscription management service, which automatically propagates the changed data to all interested consumers. As a consequence, this scenario does not ripple—it affects no components beyond the data producer. This is indicated in Figure 19.

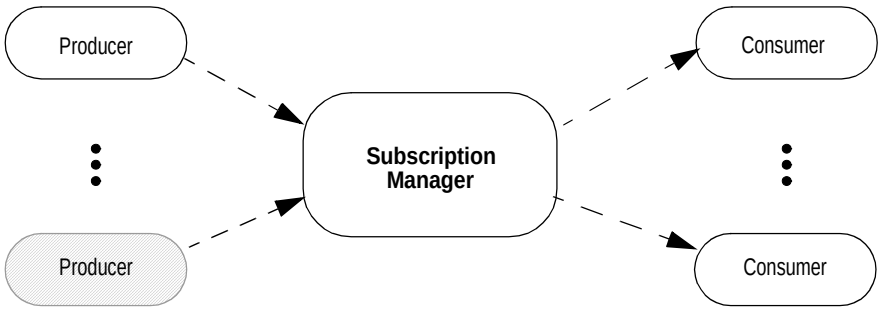

Figure 19: Changing the Timing of Data Production and Consumption in the Publish/Subscribe Sub-ABAS

On the other hand, this scenario would have significant ripple effects for the Data Indirection ABAS and the Abstract Data Repository sub-ABAS since in these ABASs you would have to change not only the rate of data production in the producer, but also all of the consumers. Furthermore, using these other ABASs you would have to somehow synchronize the producers and consumers to ensure data integrity. Otherwise the furnace controller and operator might be operating on old data or miss a data update.

D.3.2 Analysis and Design Heuristics

You should choose this ABAS if you expect to be changing the timing of data production and consumption frequently, or if this timing is not under programmatic control but rather is a reaction to external stimuli. In such cases, the use of the Publish/Subscribe sub-ABAS is appropriate. It allows the architect to plan for dynamically changing the timing of data production and consumption, add new consumers, or delete old consumers with virtually no effect on the remainder of the system.

This flexibility of course comes at a performance cost, which is twofold. First of all, there is a performance penalty to pay because the coupling between the data producer and consumer is now indirect, through the subscription manager (irrespective of whether this is implemented as a separate component). Secondly, if there are complex data dependencies (e.g., A sends a message to B which triggers sending a message to C) and these dependencies need to change (e.g., you now want to filter the message that A sends to B via component Q), these changes will be non-local; B will now have to subscribe to component Q and unsubscribe from component B; Q will have to subscribe to component A and provide a subscription interface for B. Furthermore, any cyclic dependencies or timing requirements on message deadlines may be difficult to reason about in this sub-ABAS.

Appendix E Layering ABAS

E.1 Problem Description

Layered architectures exist to provide cumulative levels of abstraction on top of some base functionality. These layers exist to hide implementation details and provide for modifiability at varying levels of abstraction. Layered architectures are used, for example, to provide portability layers for software systems that must run on different operating systems and hardware platforms, and to provide a common abstraction for communications. Many complex systems use layers as its software architecture or as a significant part of its architecture.

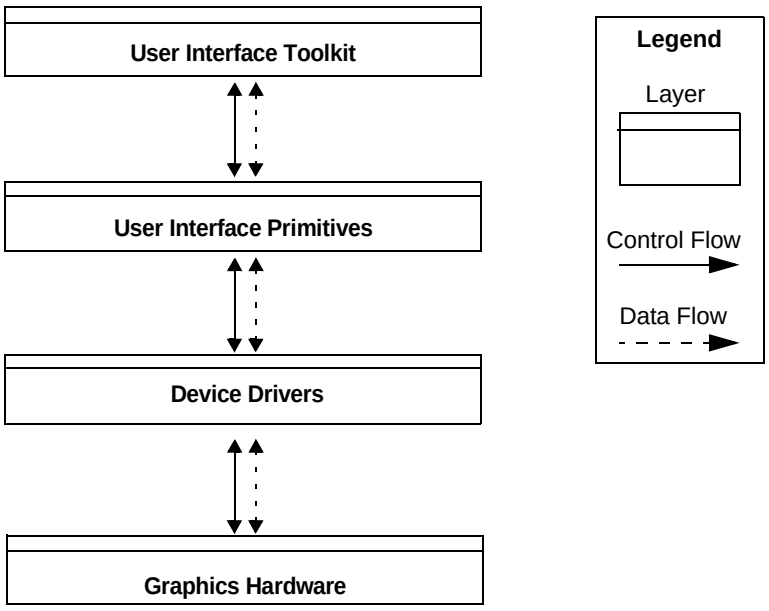

Figure 20: A Typical Representation of a Layered System

A typical example of a layered system, as shown in Figure 20, is found in graphics and user interfaces where the bottom layer knows how to physically draw onto a specific graphical output device or receive input from an input device, and the top layers consist of high-level interfaces that a programmer can use to create a user interface. Various intermediate layers exist to provide portability or intermediate abstractions for more fine-grained control over the user

interface. We say that this system is strictly layered because, as is indicated by the connections, every layer communicates only with its immediate neighbors.

Another common example of a layered system is found in communication protocol stacks, where each successive layer knows less about the details of the underlying communications hardware and provides higher level, more powerful abstractions for the programmer.

E.1.1 Criteria for Choosing this ABAS

This ABAS will be relevant if your problem inherently has distinguishable broad categories of functionality that

- are internally highly coherent
- are stable with respect to changes (that is, the categories do not change often, even if their internals do change)
- depend upon each other in predictable ways
- do not have cycles of dependencies
- have low coupling with other categories, and in particular, are typically only coupled with at most two other categories

E.2 Stimulus/Response Attribute Measures

Layering exists to isolate some parts of a system from others. Layers are inserted wherever changes are perceived to be independent. So, for example, layers are typically used to hide communication or database protocols and hide operating system or user interface toolkit implementation specifics. As a consequence of this use of layering, the stimuli of interest to layering is a modification to the system, and the measurable aspect of a layered system that the layering is expected to improve is the consequence of the change to the system, in terms of the added, changed, or deleted components, interfaces, and connections. We want to measure both the numbers of changed components, interfaces, and connections, and the magnitude of each change.

We characterize the important stimulus and its measurable, controllable responses as follows:

- **Stimulus**: a change to a layer in the software
- **Responses**: number of layers affected and number of components, interfaces, and connections added, deleted, and modified, along with a characterization of the complexity of these changes/deletions/modifications

E.3 Architectural Style

The layered architectural style consists of sets of components that implement distinguishable broad categories of functionality. Each of these sets is organized into a layer. Elements of each layer can, in the strictly layered style, only interact with elements in their layer or an adjacent layer. The elements within a layer are usually packaged uniformly, and are most commonly objects or procedures. Less commonly a layer may be constructed of processes, threads, filters, or some other component type. Elements of a layer may communicate with elements in an adjacent layer via data connections (e.g., sockets, HTTP), but are more commonly via connections that transfer both data and control (e.g., procedure call, RPC, method invocation).

Attempts to *enforce* layering within an architecture are typically made through means such as naming conventions (e.g., all functions defined by the X windows library, Xlib, begin with the letter "X"), code ownership (the graph layout layer is owned by a single development group, and only they can change it), and design conventions (the graph layout layer cannot directly call the window system, but must instead call a virtual toolkit layer). For example, the "operating system" layer seen in many architectural diagrams is only a layer by virtue of code ownership; few designers or developers have access to its source. So, designers must treat it as a sealed layer.

Table 6 describes the architectural parameters and decisions of concern for the ABAS necessary for creating an analytic model of modifiability.

Modifiability Architectural Parameters
topology: *layers*
connectivity: *to adjacent layers only*
knowledge of data schema: *no knowledge*

Table 6: Architectural Decisions for the Layering ABAS

E.4 Analysis

To measure and, more importantly, to *predict* the impact of the modifications, we have two choices: walkthroughs or metrics. With metrics, we can assess the coupling and cohesion of a layered architecture which should predict the average transitive closure of changes and hence the average difficulty of changes. With walkthroughs, we can *directly* assess the effects of a set of anticipated changes on an architecture.

In both cases, these techniques are an attempt to associate the stimuli, in this case a set of changes to the layered architecture, with the difficulty of the changes. This difficulty is

couched in terms of the amount of work needed to add, delete, or modify the affected components, connectors, and interfaces.

E.4.1 Reasoning

The point of the layering ABAS is to minimize the effects of changes to one layer of the software when a portion of another layer changes.

While we cannot always measure coupling directly, we will suggest, in our analysis, ways of approximating appropriate measures of coupling. The analysis technique for this ABAS (at least for the modifiability attribute) is to investigate a representative set of anticipated change scenarios. When examining the ABAS's response to these scenarios, what we are really doing is examining the transitive closure of change propagation.

For the purpose of this analysis, we assume the following three scenarios. The following discusses each in detail:

1. changing the internals of a component in a layer with no side effects
2. changing an existing interface
3. adding a new interface and exposing this functionality to an upper layer

Scenario 1: Changing a Component's Internals

One kind of modification (the most desirable kind) affects only the internals of a single component within a single layer, as shown in Figure 21. It is clear from this representation that the transitive closure (TC) of such a change = 1, because it does not affect any interface and cannot propagate beyond a single component.

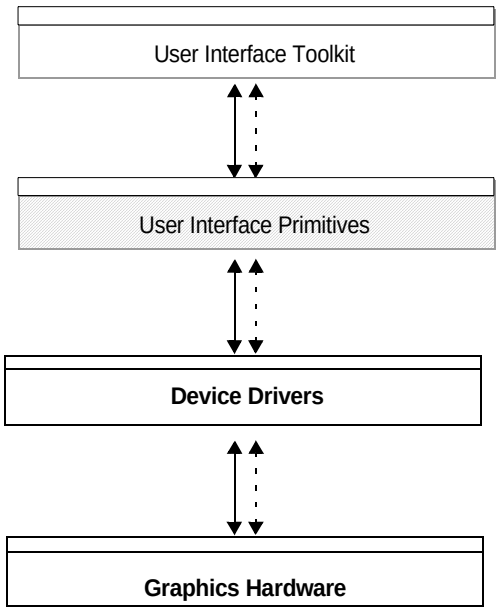

Figure 21: Changing a Component's Internals

Scenario 2: Changing an Existing Interface

In another kind of modification to a layered system, the proposed modification requires that some functionality be added that changes an existing interface or creates a new interface. For example, consider the case where the user interface primitives were previously only able to draw polygonal regions, and now a new version of the primitives is available that can draw arbitrary spline paths.

In this case, the modification will affect some component internals in the User Interface Primitives layer (to add the new functionality) and will affect that component's interface (to "expose" the new functionality). This is depicted in Figure 22. In this case, TC = 3; that is, the component itself needs to be changed, and its interface needs to be changed to expose the new functionality, which might require changes in the User Interface Toolkit layer.

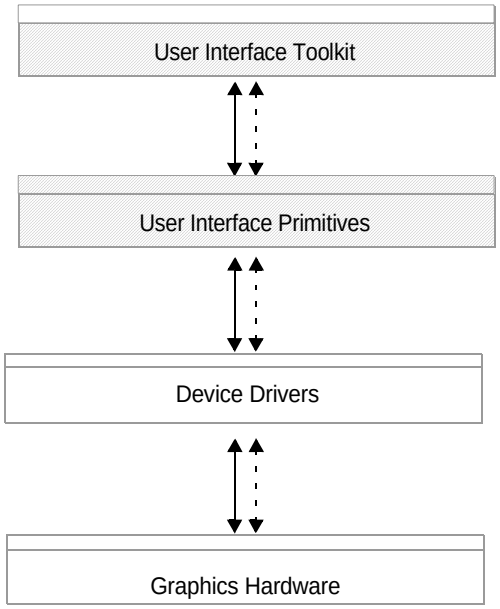

Figure 22: Changing a Component's Interface

Scenario 3: Exposing a New Interface

Additionally, the uppermost layer might want to expose the new spline capability to the user, in which case it needs to be changed as well, increasing the TC to 4, as shown in Figure 23.

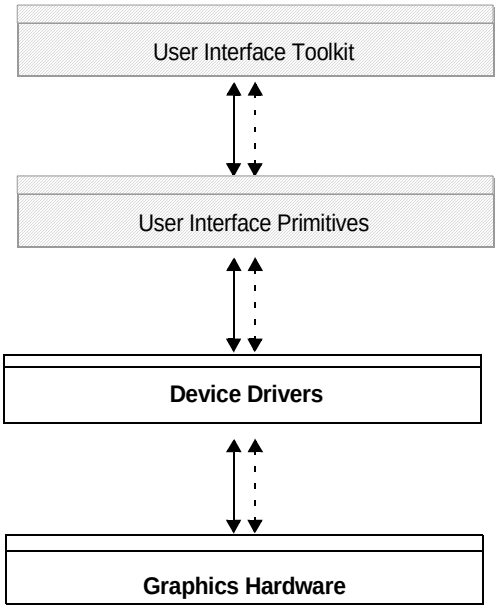

Figure 23: Changing and Exposing a New Interface

E.4.2 Analysis and Design Heuristics

Layering is a technique that is primarily used for portability. It works simply because it isolates some parts of the system from others. There are a few common design considerations to keep in mind when creating a layered system:

- can the system's performance afford the number of layers that you have proposed? Each layer comes at a cost of a performance penalty to be paid in terms of the overhead of the additional calls made when traversing the layers.
- will the layering be strict or is it acceptable to "bridge" layers? Layer bridging is often done for performance reasons in places where the overhead of traversing additional layers makes it difficult to meet performance goals. Bridging layers defeats the purpose of layering: changes at lower layers can easily be non-local and hence compromise the modifiability and portability of the system.
- will the layering permit downward calls only or will it permit upward calls as well? Strictly layered systems typically only permit downward calls. Allowing calls in both directions increases the coupling of the layers, which decreases the system's modifiability.

Appendix F Simplex ABAS

F.1 Problem Description

The Simplex [Sha 96] ABAS focuses on the problem of software reliability and availability 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 of how to take advantage of redundancy to increase reliability while avoiding "common mode" software failures.

To illustrate the problem that Simplex is addressing, consider "the update paradox," [Sha 96]. 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, 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. The first problem is how to introduce redundancy to ensure the proper level of reliability/availability without introducing common mode failures. (Even components that have been implemented by different groups and hence have different implementations can suffer from common mode failures.) The second problem is how to upgrade a system without compromising its reliability/availability? In general, the Simplex ABAS offers insights for using redundancy to manage availability/reliability.

F.1.1 Criteria for Choosing this ABAS

This ABAS will be relevant if your problem

- has the need for online upgrade
- has high availability requirements, and in particular the requirement to avoid common mode failures
- can run in one or more degraded modes, one of which can be implemented via a simple, "trusted" algorithm

- if you are considering using redundancy to manage reliability/availability

F.2 Stimulus/Response Attribute Measures

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

- **Stimulus**:
  - a fault “arrival” at the system

- **Responses:**
  - the levels of degraded service
  - the reliability/availability 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.

F.3 Architectural Style

Simplex is an architectural style that 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 24. 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.

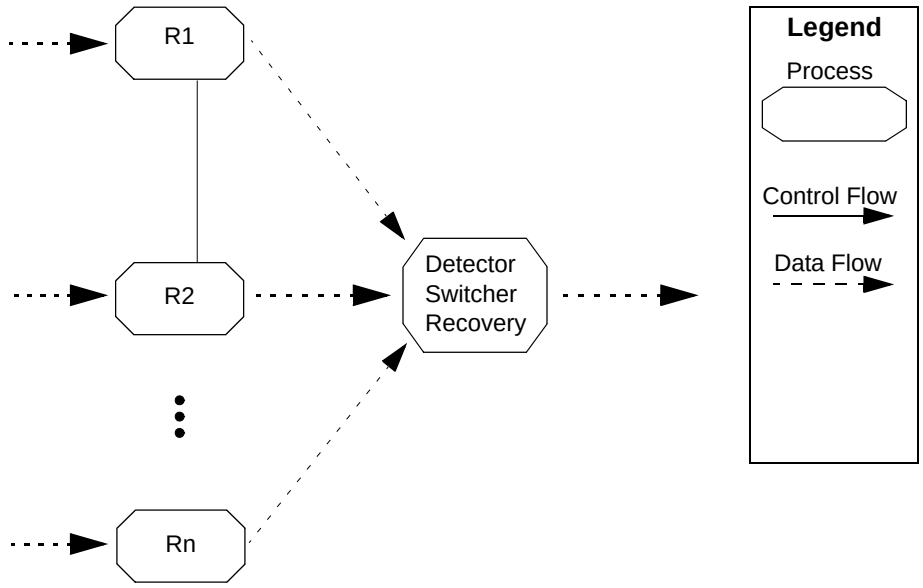

Figure 24: A Redundancy-Specific Architectural Style

The Simplex style, as shown in Figure 25, 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. 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 their implementation, the algorithms used, or the output produced.[2]

In Figure 25, the "leader" component, the other redundant components (R1 and R2), and the "safety" component are analytically redundant.

2. You can think of the relationship between power steering and mechanical steering as analytically 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. Manual steering provides the "safety" component in such a system.

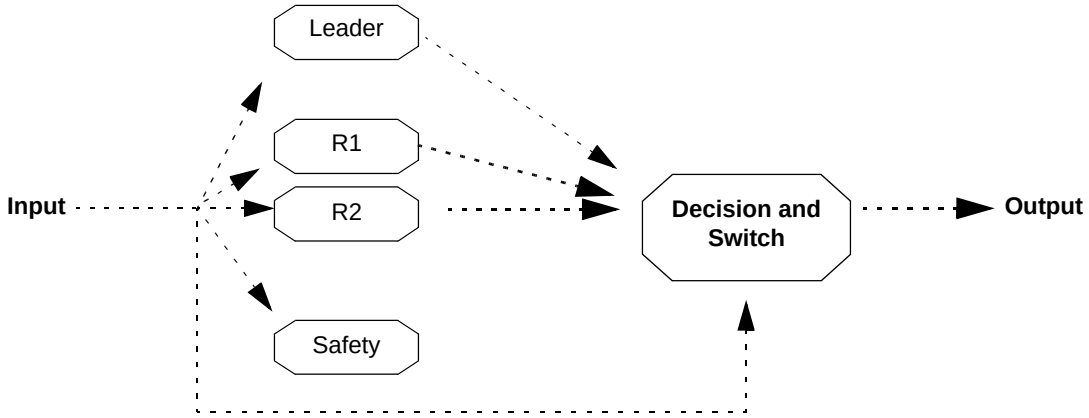

Figure 25: The Simplex Architectural Style

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 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 component 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 ABAS should address these 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.

Simplex Architectural Parameters
Failure rates of the components (leader, safety, decision/switch, etc.): λ_i
Repair rates of the components (leader, safety, decision/switch, etc.): μ_i
Number and type of redundant components: *m*
A characterization of the level of service provided by each component: *optimal/degraded*
Voting mechanism: *comparison each other and safety controller*

Table 7: Architectural Decisions for the Simplex ABAS

In most cases, it is necessary to estimate the failure and repair rates of the various components. We will assume, in the analysis that follows, that the failure rates for the decision/switch component and the safety component are very low in comparison to the failure rates of the other components.

F.4 Analysis

To model the availability of this style, you have to estimate the failure and repair rates of the components to calculate the availability of the system. Reliability growth models.

F.4.1 Reasoning

Modeling a Simpler Problem

Before discussing the analysis of the Simplex style, we'll take a look at a similar style, the *majority voting* style. This is the style that we used in the problem description to illustrate the update paradox. For this style, there are three redundant components.[3] At least two of the three 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, as defined, can be in only one of three states: three working components, two working components, or failed. If F failures per year occur and a component

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

repair takes on the average 1/R years, the Markov model shown in Figure 26 can be used to calculate the availability (that is, the proportion of time that the system is not in the failed state).

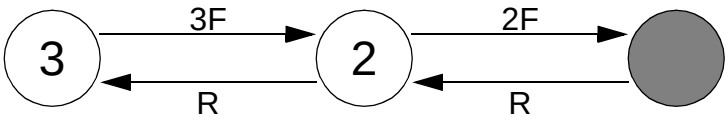

Figure 26: A Markov Model for the Majority Voting Style

The representation of a Markov model in Figure 26 can be viewed as a state diagram. State “3” represents the state in which three components are working; state “2” represents the state in which two components are working; and the gray state is the failed state. The transition arrows are labeled with failure (F) and repair (R) rates. Since each component fails independently with an average rate of F, three 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 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 the 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 highly reliable safety controller, and a monitoring and decision unit. The Simplex style preserves the total number of active components, as compared with the majority voting style, but allocates functions to components differently

depending on their states, and ultimately the components have different failure properties. The Markov model for this style is shown in Figure 27.

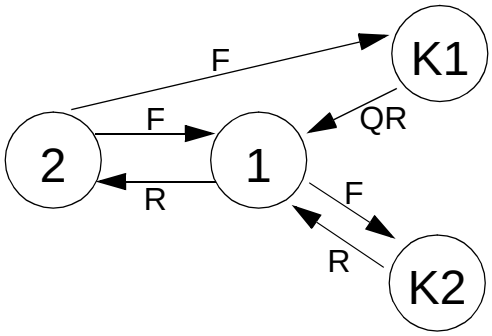

Figure 27: A Markov Model for the Simplex Style

The system starts in state “2” with two high performance controllers, the outputs of which are compared. If they agree, we assume that they are 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 (that is, the output will not place the system in a state which can not be controlled by the high performance controller), 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 state in which the safety component is made to be 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).

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 27 closely approximates the model in Figure 28, below. The availability properties of the model shown in Figure 28 are better than for majority voting (shown in Figure 24) 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 Simplex uses a semantic check of the output for failure detection.

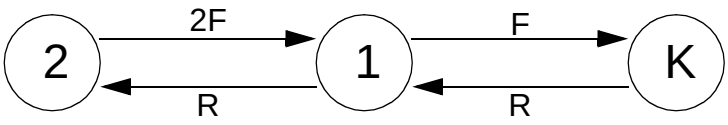

Figure 28: An Approximate Markov Model for the Simplex Style

F.4.2 Analysis and Design Heuristics

There are several key heuristics embodied in the Simplex architecture. Rather than consider a system to be in one of two states, working or failed, Simplex uses several analytically redundant levels of service. Analytically redundant levels help to circumvent common mode failures by using a very highly reliable albeit less precise backup to a very precise but possibly less reliable level of service. Also, the simplex style exploits the possibility that repair from a reduced level of service could be more efficient than repair from a completely failed state. This quick repair allows for a relatively high level of availability.

The Simplex approach circumvents the upgrade paradox by substituting a different failure detection criterion; majority voting uses “majority” as the correctness criterion, whereas Simplex applies correctness heuristics (based on principles of control theory) to the output values themselves. As long as the output values can pass the Simplex correctness criterion, a “minority” value can still be accepted.

References

[Booch 96] Booch, G. "Object-Oriented Development." *IEEE Transactions on Software Engineering 12*, 2 (February 1996): 211-221.

[Briand 99] Briand, L.; Daly, J.; & Wuest, J. "A Unified Framework for Coupling Measurement in Object-Oriented Systems." *IEEE Transactions on Software Engineering*, 1999.

[Buschmann 96] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; & Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns* Wiley & Sons, 1996.

[Gamma 95] Gamma, E.; Helm, R.; Johnson, R.; & Vlissides, J. *Design Patterns* Addison Wesley, 1995.

[Gonzalez 91] Gonzalez Harbour, M.; Klein, M.; & Lehoczky, J. "Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority,"116-128. *Proceedings of IEEE Real-Time Systems Symposium*, IEEE Computer Society Press, 1991.

[Kazman 94] Kazman, R.; Abowd, G.; Bass, L.; & Webb, M. "SAAM: A Method for Analyzing the Properties of Software Architectures," 81-90. *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994.

[Kazman 99] Kazman, R.; Barbacci, M.; Klein, M.; Carriere, S. J.; & Woods, S. G. "Experience with Performing Architecture Tradeoff Analysis," 54-63. *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.

[Klein 93] Klein, M.; Ralya, T.; Pollak, B.; Obenza, R.; & Gonzalez Harbour, M. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems* Kluwer Academic Publishers, 1993.

[Klein 99] Klein, M.; Kazman, R.; Bass, L.; Carriere, S. J.; Barbacci, M.; & Lipson H. “Attribute-Based Architectural Styles,” *Software Architecture* 225-243. *Proceedings of the First Working IFIP Conference on Software Architecture*, San Antonio, TX, February 1999.

[Leung 82] Leung, J. L. T. & Whitehead, J. “On the Complexity of Fixed Priority Scheduling of Periodic, Real-Time Tasks” *Performance Evaluation, 2,* 4, 237-250.

[Rajkumar 91] Rajkumar, R. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Boston, MA: Kluwer Academic Publishers, 1991.

[Sha 96] Sha, L.; Rajkumar, R.; & Gagliardi, M. *A Software Architecture for Dependable and Evolvable Industrial Computing Systems* (CMU/SEI-95-TR-005, ADA301169). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1996. http://www.sei.cmu.edu/publications/documents/95.reports/95.tr.005.html

[Shaw 90] Shaw, M. “Prospects for an Engineering Discipline of Software” *IEEE Software 7,* 6 (November 1990): 15-24.

[Shaw 96] Shaw, M. & Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline* Prentice Hall, 1996.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1999	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Attribute-Based Architectural Styles	C — F19628-95-C-0003
6. AUTHOR(S)	
Mark Klein, Rick Kazman	

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-99-TR-022
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
HQ ESC/DIB 5 Eglin Street Hanscom AFB, MA 01731-2116	ESC-TR-99-022

11. SUPPLEMENTARY NOTES

12.a DISTRIBUTION/AVAILABILITY STATEMENT	12.b DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (maximum 200 words)

An architectural style is a description of component types and their topology. It also includes 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 engineering artifacts because they define classes of designs along with their associated known properties. They offer experience-based evidence of how each class has been used historically, along with qualitative reasoning to explain why each class has its specific properties.

Attribute-Based Architectural Styles (ABASs) build on architectural styles to provide a foundation for more precise reasoning about architectural design by explicitly associating a reasoning framework (whether qualitative or quantitative) with an architectural style. These reasoning frameworks are based on quality attribute-specific models, which exist in the various quality attribute communities (such as the performance and reliability communities).

Architectural styles, and hence ABASs, are powerful because they provide a designer with the concentrated wisdom of many preceding designers faced with similar problems. In this report we exemplify the use of ABASs in both design and analysis. We argue that ABASs provide the groundwork to create an engineering discipline of architectural design—to make design a predictable process rather than an ad hoc one.

14. SUBJECT TERMS	15. NUMBER OF PAGES
architectural style, component types, topology, ABAS	82
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL