A BASis (or ABASs) for Reasoning About Software Architectures

Mark Klein, Rick Kazman, Robert Nord
Software Engineering Institute, Carnegie Mellon University
Pittsburgh, PA, U.S.A. 15213

+1-412-268-7615
{mk, kazman, rn}@sei.cmu.edu

ABSTRACT

This paper discusses the use of Attribute-Based Architectural Styles (ABASs)—architectural styles accompanied by explicit analysis reasoning frameworks—in design. The paper has two main objectives: to convince readers that ABASs provide a basis for insightful reasoning about a software architecture's ability to meet its quality attribute goals; and to demonstrate the utility of ABASs by showing an example of how ABASs are used to design an industrial system architecture entirely via ABASs. In the process of designing this architecture, we show excerpts from our growing ABAS handbook and argue for why ABASs help us in designing architectures efficiently and predictably.

Keywords
Architecture analysis and design, quality attribute models, architectural styles

1 ATTRIBUTE-BASED ARCHITECTURAL STYLES

Architectural Styles

An architectural style (as defined by Shaw and Garlan [10] and elaborated on by others [1]), is a description of components, connectors, topology, and some constraints on their interaction. Architectural styles also provide an informal description of their strengths and weaknesses. Architectural styles are important engineering artifacts because they define classes of designs along with their associated known properties. They offer experience-based evidence and qualitative reasoning about how each design class is used. “Use the pipe and filter style when reuse is desired and performance is not a top priority” is an example description that is found in definitions of this style.

What are ABASs?

In [7] and [8] we introduced the notion of an Attribute-Based Architectural Style (ABAS) to offer a foundation for more precise and efficient reasoning about architectural design. We accomplish this goal by explicitly associating a *reasoning framework* (qualitative or quantitative) with an architectural style, which shows how to reason about the design decisions comprising the style. These reasoning frameworks are based on quality attribute-specific models, which exist in the various quality attribute communities (such as the performance, modifiability, and reliability communities).

ABASs are powerful because they provide a reuser with the concentrated wisdom of many preceding designers faced with similar problems. Using ABASs will allow an architect to employ the collected knowledge of the architecture design community and the various quality attribute communities in much the same way that object-oriented design patterns have given novice designers access to a vast array of experience collected in the object-oriented design community [2].

ABASs are an important step in bringing predictability—the hallmark of a mature engineering discipline—to architecture design. ABASs promote a disciplined way of doing design and analysis of software architecture based on reusing known patterns of software components with predictable properties. We will emphasize this point throughout this paper and exemplify it in a concrete design example.

ABASs are not only attribute-based but, unlike existing architectural styles, are also attribute *specific*. This is a form of separation of concerns. When we design or analyze using ABASs we consider a single quality attribute at a time, because each ABAS is associated with only one attribute reasoning framework. However, in many situations, separation of concerns inevitably leads to a subsequent need for composition of concerns. Later in the paper we address this, showing how to design an architecture by combining several ABASs.

The Sections of an ABAS

An ABAS description consists of 4 major sections:

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 the components and connectors, patterns of data and control interactions (topology), and any constraints on the style.

4. *Analysis* - a description of how the quality attribute models are formally related to the architectural style, along with

heuristics of how to reason about the style.

Our experience has shown that ABASs provide a number of benefits to engineers designing and analyzing complex architectures [6]. They provide:

- a way of reasoning about the architectural style with respect to a quality attribute such as modifiability, usability, availability, performance, etc.
- a package consisting of an architectural style along with an analytic model and a checklist of attribute-specific questions
- an analytic characterization that concisely tells you whether this style is appropriate for the problem that you face (in much the same way that algorithms are characterized as $O(n^2)$ or $O(\log n)$)
- a way of dividing and conquering an architectural design or analysis problem

In this paper we will report on a series of small experiments that offer evidence of these benefits.

Approach

One step on the path of putting ABASs to practical use is to compare a design generated completely from ABASs to a design generated in practice. To prove the utility of ABASs in design we would need to run a controlled experiment where we designed a system two ways, one with and one without ABASs. This is exceedingly difficult to do and to control in practice. What we did instead as an intermediate step is to “re-design” an existing system primarily from the requirements. We also had access to some limited design documentation, but we relied upon the requirements and ABASs to generate our own design and compared it to the existing design at several intermediate stages. Our goal was to understand how ABASs can be used to guide the design and to illuminate critical design decisions and tradeoffs.

2 CASE STUDY

The goal of this section and the next is to demonstrate how ABASs guide and clarify architectural reasoning. In this section we will describe the requirements for a health care patient monitoring system.[1] In the next section we will show how to use ABASs to construct an architectural design which addresses those requirements. We will then use this “ideal” design to give us insight into the actual design.

This system is an embedded, real-time monitoring system, discussed in [4]. It is a stand-alone bedside unit that obtains and displays a patient’s vital signs, or sends them to a central unit for display. The bedside unit can be transported along with a patient, so physical size and cost limitations impose severe hardware constraints. The system has different configurations to produce a set of medium to high-end patient monitors.

The primary function of the system is to acquire data and make it available to the user for viewing. This functionality can be seen in the scenario depicted in Figure 1.

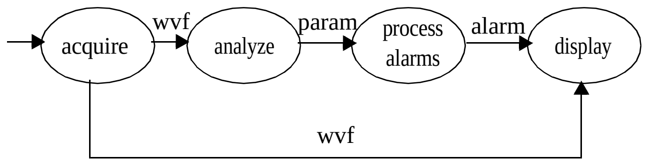


Figure 1: Producer Consumer Processing Scenario

The scenario shows how raw data is acquired, processed, and displayed on the screen. Data flow between entities is indicated by arrows labelled with the type of data. The waveform (wvf) data is first acquired. The acquired data is both displayed on the screen and processed to produce key parameter (param) values. These values are further processed to produce visual and audio feedback as well as alarm information for the clients, and are displayed on the screen.

In addition to these functional requirements, the architecture was influenced by the need of the monitoring system to handle a flexible and growing set of requirements, to run on different hardware and software platforms, and to meet real-time performance requirements. These are as follows.

Modifiability requirements: adding/modifying producers and consumers of information including, but not limited to: changes to smoothing/filtering algorithms, alarm sensing logic, and trend tracking; new data formats; different displays; relocation of application entities to other processors.

Portability requirements: the system must be able to run on multiple hardware platforms; use different types of devices (displays and sensors); run on different operating systems; and accommodate different graphics primitives.

Real-time performance requirements: Raw data is sensed every 10 ms. and must be displayed on display device 1 within 20 ms. of being acquired. Many different types of waveform calculations are performed. Some are performed once every 100 ms. and other are performed once every 300 ms. The 300 ms calculations are performed using reused legacy software assets. The results of the 300 ms. calculations must be displayed on the patient monitor as a waveform with an associated beep within 50 ms. of completing the calculation. Certain detectable conditions should cause an alarm to be triggered. The alarm should manifest on the patient monitor within 500 ms. of completing the waveform calculation. The results of the 100 ms. calculation should be sent (but not necessarily received) to a remote site within 20 ms. of the completion of the calculation.

3 APPLYING ABASs

We will now show how we chose and used ABASs to create an initial design for the patient monitoring system. In each case we started with a portion of the functional and quality requirements, used these to index into our handbook[2] of ABASs for one that addresses some portion of the require-

1. Although this is a real system, many of its details have been modified to protect the company’s proprietary interests.

2. The handbook is continually growing, but an early version of it can be found at [8].

ments, and instantiated that ABAS to meet the needs of this system. We will discuss combining the ABASs and making tradeoffs among them in Section 4.

Addressing Portability: The Layering ABAS
As stated above, the system's requirements state that it must be able to run on multiple hardware platforms; use different types of devices (displays and sensors), run on different operating systems; accommodate changes in the underlying graphics primitives.

From our handbook we determine that the Layering ABAS has role in satisfying each of these requirements. Extract 1 shows a portion of the Layering ABAS. This extract illustrates a portion of the reasoning that a user of the ABAS handbook would go through in choosing this ABAS:

Problem Description: 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

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, to hide operating system or user interface toolkit implementation specifics.

- **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.

Extract 1: Choosing the Layering ABAS

The ABAS[3] tells us that layering is a technique that isolates some parts of the system from others. In the case of the patient monitoring system it will hide implementation specifics that underlie key interfaces to devices, communication protocols, the operating system, and graphics primitives.

The patient monitoring application satisfies many of the criteria listed in Extract 1. Application entities need an interface to graphics, to various devices and to a communication service which provides location and name independence. Each of these collections of services is highly coherent. Each collection of services uses lower level primitives and/or the operating system in a predictable manner. Dependency is unidirectional. The ABAS would lead us to consider a layering style as shown in Figure 2. We have not shown the data and control connections in this figure, but we begin with the ideal assumption that each layer accesses only its adjacent lower layer.

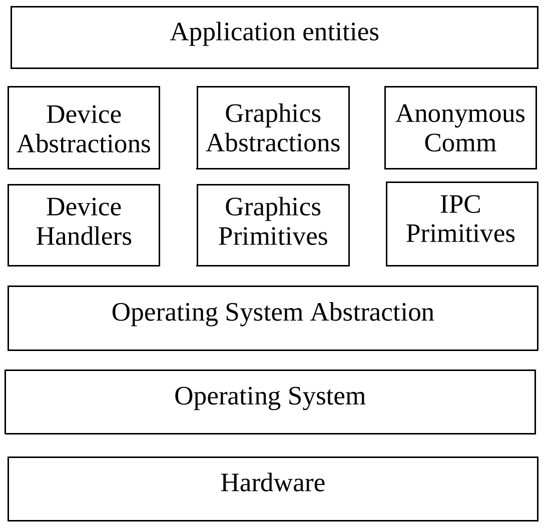


Figure 2: Layering Style for Portability

Layering is not a new topological concept. A benefit of the Layering ABAS is not that it introduces layering, but rather that it offers a succinct characterization of the *criteria* under which a layering style might apply. But an even more significant benefit provided by an ABAS is the association of quality attribute-based reasoning with the topological architectural style. Extract 2 illustrates the reasoning associated with the Layering style.

The Layering ABAS is a modifiability ABAS. The reasoning framework for all modifiability ABASs (and portability is a form of modifiability) is *scenario-based*: we posit classes of probable modification scenarios and then trace the propagation of changes due to each posited modification to understand their implications. When modifications are confined to the internals of a specific layer (scenario 1 in Extract 2), then everything from that layer's interface up through the top layer is immune to the modification. In our case, since the modifications are confined to the internals of the third layer in Figure 2 (Device Handlers, Graphics Primitives and IPC Primitives) or the fourth layer (Operating System Abstraction layer), then the desired portability will be achieved.

The analysis of other scenarios in the layering ABAS show that the transitive closure of other modifications can be larger than 1. For example, if application entities (in Figure 2) communicate with one another by directly using the internals of the IPC Primitives then application entities are susceptible to the ripple effects of modifications that affect the internals of the IPC Primitives. This "layer bridging" com-

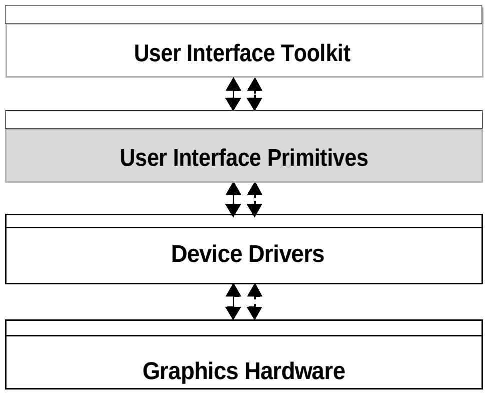
3. Note that this extract is a condensed portion of the actual ABAS, which is typically 5-10 pages in length.

Analysis
The analysis technique for this ABAS 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:

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

Changing a Component's Internals
One kind of modification (the most desirable kind) affects only the internals of a single layer, as indicated below by the shading. It is clear from the representation below that the transitive closure of such a change = 1, because it does not affect any interface and cannot propagate beyond a single component.



Extract 2: Analyzing a Layering ABAS

promises portability.

How Did the ABAS Help?
Notice that the ABAS isolates and highlights key design decisions that result in the desired modifiability properties (locality of changes). Identifying and documenting key design decisions, and understanding the relationship between theses design decisions and desired behavior is central to predictability in engineering. Understanding this relationship sometimes only requires qualitative heuristics. However, the level of sophistication of the analysis is not the point here. The point is that design decisions within an ABAS framework are accompanied by an associated analysis and a set of assumptions. These help to ensure the desired properties are highlighted and adhered to during development.

In the current practice of architectural design even simple heuristics such as avoiding layering bridging are often ignored. The result of such decisions are at first seemingly innocuous. Cumulatively, however, they create an architectural rigidity that eventually turns into an insidious and costly problem.

Our goal is to create an engineering handbook of ABASs, which we hope will foster a routine software engineering practice in which such heuristics are in the forefront of every designer's mind. This is a theme we will later reiterate. For now let us move on to addressing modifiability.

Addressing Modifiability: The Publish/Subscribe ABAS
The modifiability requirements for the system center around the addition, modification, and deletion of producers and consumers of information including, but not limited to: changes to smoothing/filtering algorithms, alarm sensing logic, and trend tracking; new data formats; different displays; relocation of application entities to other processors. Although it is not an exact fit, the Publish/Subscribe ABAS seems well suited to dealing with these requirements. Extract 3 shows the criteria for choosing this ABAS.

In this ABAS, data producers (publishers) and consumers (subscribers) communicate via an *abstraction* of shared data. The paradigm is that they are sharing state rather than explicitly communicating. When new data is published all interested subscribers receive it. The ABAS dissociates publishers and subscribers of data in terms of their identities, their locations, the internal format of their shared data, and the temporal control between them. In this way it helps to satisfy the modifiability requirements of our system: it is easy to add or change publishers and subscribers of data because they do not depend on each other directly. Subscrib-

Problem Description: 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.

Stimulus/Response Attribute Measures
- **Stimuli**:
  - a new producer or consumer of data
  - a modification to an existing producer or consumer of data
  - 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.

Extract 3: Choosing the Publish/Subscribe ABAS

ers only depend upon the state that they “share” with publishers.

Adding new subscribers of existing published data is easy: they just subscribe to the existing data. The effects of changing data formats is limited if the data producer does not expose the changed data formats to its consumers, i.e., it maintains its public interface to the data. For example, in the case of our system alarms may be internally stored and calculated in a variety of ways, but as long as these differences are masked by the publishers, the remainder of the system is unaffected.

We can now use this ABAS to organize the system’s communicating entities according to the information that they share, and relegate the details of how their communication is accomplished to the Publish/Subscribe mechanism, as shown in Figure 3 (where the grey lines represent Publish/Subscribe communication).

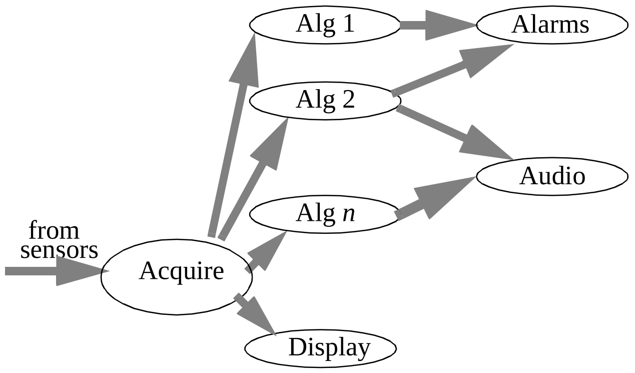


Figure 3: Publish/Subscribe Style for Modifiability

The analysis associated with the Publish/Subscribe ABAS asks the designer to consider the effects of future modifications on the structure of the system, as shown in Extract 4. With respect to these scenarios, this ABAS appears to be perfectly suited to the modifiability requirements, because each of the changes is localized to a single publisher or subscriber.

However, the Analysis and Design Heuristics ask the user of this ABAS to be wary of two *performance* implications: the added overhead of the indirection that is part of this mechanism, and the caution that meeting timing requirements may be difficult. For the moment, however, the design of the system can proceed since we are only concerned initially with the quality attribute of modifiability in this ABAS. Interactions among the ABASs will be dealt with separately in Section 4.

The Publish/Subscribe mechanism adds overhead to the communication between entities in the form of additional calls (since this mechanism is an additional layer on top of IPC) and copying the message *n* times, once for each subscriber. While proposals do exist for real-time Publish/Subscribe mechanisms [9], they are not yet commercially available and so we must deal directly with this performance challenge. This highlights a design trade-off between time to market, cost, and performance (since using an off-the-shelf non-real-time Publish/Subscribe mechanism results in quick time to market at a low cost but with no performance guarantees).

Analysis
While we cannot always measure coupling directly, there are ways of approximating coupling via scenarios [5]. These scenarios form a palette of change types, 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 six change scenarios:

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
6. changing the timing of data production and consumption

Analysis and Design Heuristics
There is a performance penalty to pay in this ABAS 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).

. . .

Furthermore, any cyclic dependencies or timing requirements on message deadlines may be difficult to reason about in this sub-ABAS.

Extract 4: Analyzing the Publish/Subscribe ABAS

Addressing Performance: The Concurrent Pipelines ABASs

The performance requirements for the patient monitoring system express the following processing periods and deadlines:

- Raw data is sensed every 10 ms. and must be displayed on display device 1 within 20 ms. of being acquired.
- Many different types of waveform calculations are performed. Some are performed once every 100 ms. and other are performed once every 300 ms. The 300 ms. calculations are performed using legacy software.
- The results of the 300 ms. calculations should be displayed on the patient monitor as a waveform with an associated sound within 50 ms. of completing the calculation.
- Certain conditions detectable from analyzing the waveforms should cause an alarm to be triggered. The alarm should manifest on the patient monitor within 500 ms. of completing the waveform calculation.
- The results of the 100 ms. calculation should be sent (not necessarily received) to a remote site within 20 ms. of the

completion of the calculation.

The first design driver to consider here is that there are several periodic activities with several processing stages, where each stage has a real-time deadline. Given that the cost constraints of the patient monitoring system force us to entertain a uniprocessor solution to this problem, the Concurrent Pipelines ABAS—as partially illustrated in Extract 5—appears to give us the reasoning framework needed to reason about meeting the stringent real-time performance requirements.

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

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

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

Extract 5: Choosing the Concurrent Pipelines ABAS

Following the style of the ABAS, we can configure the system as a set of concurrently executing pipelines, with the required periods (pd) and deadlines (dl), as is shown in Figure 4. Note that waveform data is acquired every 10 ms. from a single source (a set of sensors) and is subsequently used by the 20 ms. Display process and the 100 ms. and 300 ms. period analysis algorithms. In effect, the Acquire process needs to operate at three different rates, but it is forced to be a single function because it must acquire date from the sensors. The Concurrent Pipelines ABAS assumes that each pipeline has its own initiating process. This is clearly not the case here. What to do?[4]

We will initially bridge the difference between the inherent nature of the problem and the form that the ABAS leads us to by assuming that each pipeline has its own initiating process. Hence we can realize the style as shown in Figure 4. In this figure rounded rectangles are processes and lines indicate data transfer between the processes.

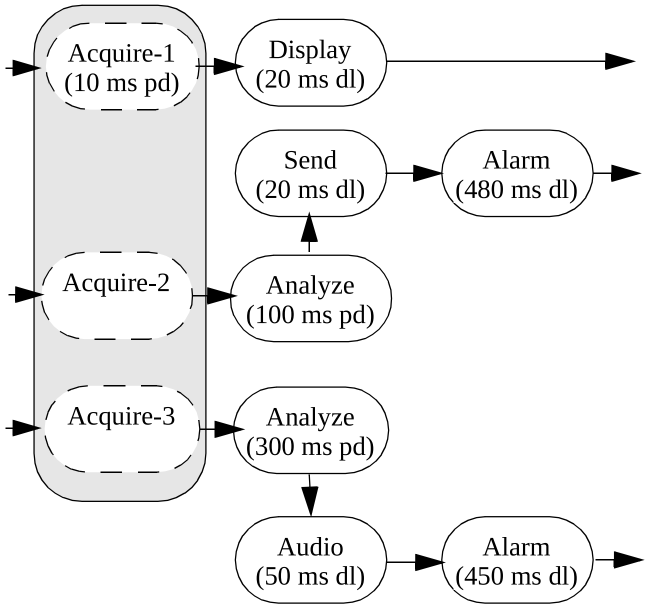


Figure 4: Design Using the Concurrent Pipelines Style to Predict Performance

There are several points to notice about this design. The Acquire function has been allocated to three virtual processes, one for each pipeline. The shaded area indicates the one physical process which is actually acquiring the data within which reside the virtual processes (processes shown bordered by dashed lines). For now we have three pipelines and can directly use the Concurrent Pipelines reasoning model. We will return to this design decision later.

Since the alarm function must be completed 500 ms. after the algorithm processing, but must also wait for the completion of Send (with a 20 ms. period) and Audio (with a 50 ms. period) in the second and third pipelines respectively, the deadlines for the Alarm processes were changed to 480 and 450 ms.

Extract 6 from the ABAS handbook offers guidance for analyzing the Concurrent Pipelines style's performance characteristics. In particular, the ABAS shows how to reason about the effects of priority assignments on latency and provides a formula for calculating (and hence predicting and control-

4. It should be noted that this kind of question is not atypical: ABASs will not always precisely fit design or analysis situation at hand. Having to map your problem into a model that is analyzable by an ABAS, or having to extrapolate beyond what the ABAS offers will be a common situation in practice. In such situations the engineer has a choice: to adapt the design so that it is amenable to routine analysis, or to extrapolate beyond the style as given to cover the existing problem. Each ABAS offers guidelines to help in these situations along with references to works that offer more comprehensive treatments of the problem space.

ling) the worst-case latency of the system.

First, consider the 10 ms. deadline associated with the Acquire process. Assume that the publication of data introduces on average 2-3 ms. latency. In the worst-case Acquire-1, Acquire-2, and Acquire-3 are publishing during the same 10 ms. period in which case Acquire-1 is in danger of missing its 10 ms. deadline. This alerts us the possibility of Acquire-1 incurring significant delays due to Acquire-2 and Acquire-3. Secondly, even if we can eliminate the delays due to Acquire-2 and Acquire-3, if Acquire-1 publishes data every 10 ms. it will use up about 30% of the processor's resources.

Analysis
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 [3]. 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. (Treat equal priorities as greater.)
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 converges.

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

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 $LowP_i$.

Analysis and Design Heuristics
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.

Extract 6: Analyzing Concurrent Pipelines

We can make two design decisions based upon this simple exercise: 1) Acquire-1 must use a less expensive data transfer mechanism, and 2) we must prevent Acquire-2 and Acquire-3 from significantly delaying Acquire-1, since it has the shortest period. To treat the first issue we will use the less expensive IPC mechanism instead of the Publish/Subscribe mechanism. To treat the second issue we will place Acquire-1 in a separate process whose priority is higher than the priority of the process that houses Acquire-2 and Acquire-3. This is consistent with the one of the Analysis and Design Heuristics in the ABAS (in Extract 6) which suggests that priorities should be a function of deadlines: the shorter the deadline, the higher the priority.

The next question is how to assign priorities to each process. The ABAS provides us with several guidelines:

- the effective priority of a pipeline is the lowest priority in the pipeline
- shorter deadlines are generally accorded high priorities

These guidelines suggest the following priority structure for the three pipelines (larger numbers are higher priorities):

1. Acquire-1 (30) --> Display (27)
2. Acquire-2 (20) --> Analyze-100 (20) --> Send (20) --> Alarm (18)
3. Acquire-3 (20) --> Analyze-300 (19) --> Audio (15) --> Alarm (13)

There are a few points to note about the prioritization structure. Pipeline 1 has priorities that are higher than the priorities of the other pipelines. Consequently it cannot be delayed by the other pipelines. Since we have only separated data acquisition into two processes, Acquire-2 and -3 are left as cohabiting the same process and consequently must have the same priority. Consequently pipeline 3 can potentially delay pipeline 2. Analyze-300 in the third pipeline has a priority that is consistent with its 300 ms deadline; a priority that is lower than Acquire-3, but higher than Alarm in pipeline 2. Consequently it can delay pipeline 2.

How Did the ABAS Help?
These observations (and even a more quantitative analysis) are enabled by the Concurrent Pipelines ABAS. The ABAS provides a framework for reasoning about this architectural style thereby providing rationale for design decisions, and just as importantly, a framework within which the design can evolve (by changing priorities, for example) while maintaining predictable performance behavior. These guidelines provide a similar benefit as being able to characterize an algorithm as $O(n^2)$ or $O(\log n)$.

4 COMPOSING ABASs FOR DESIGN

In this section we will compose a complete architectural design from the ABAS-inspired pieces of architecture discussed in the previous section. During this phase we move from the idealized design world of pure ABASs into the real world of design limitations and tradeoffs.

The System's Architectural Design
A complete architectural representation, from the perspective of its concurrency view as mapped onto hardware, is shown in Figure 5 (where empty rectangles indicate separate

computational resources such as a CPU or network) and shaded rectangles with horizontal lines indicate queues). This design now includes a number of key design decisions worth elaborating.

Key Design Decisions

The Acquire and Display processes correspond to the Acquire-1 and Display processes in Figure 4. This pipeline represents the greatest challenge to system performance, since it executes every 10 ms. The Display process is responsible for interacting with the Display-1 device (direct data reads/writes to system resources such as queues are indicated by thin black arrows). The Display process is likely to incur delays due to interacting with the device which it has been given an additional 20 ms. to accomplish. However, if device interactions have not been completed by the time Acquire is ready to read data again, Acquire should not be delayed. Consequently the Display function has been allocated to its own process to take advantage of the additional 20 ms.

The Acquire and Display processes communicate via IPC, rather than using the Publish/Subscribe connectors that connect the other processes. The black dashed arrow in Figure 5 indicates the place in which IPC was used instead of Publish/Subscribe. This is a key deviation from the Publish/Subscribe style in that it is the only portion of the communication between system's software entities that does not use the Publish/Subscribe mechanism.

The Distribute process corresponds to Acquire-2 and Acquire-3 in Figure 4. Its role is to wake up every 100 ms, read data from memory that it shares with Acquire, and publish data for the 100 ms. and 300 ms. algorithms. Distribute has a lower priority than Acquire to ensure that its processing does not delay Acquire.

We entertained the idea of allowing the Analyze process to directly read data from shared memory (that is, without using Distribute as an intermediary that publishes data), but this would compromise the ability to move application entities to another processor in the future. Note we are assuming that a COTS published/subscriber service would provide location transparency whereas a home-grown abstract data type (ADT) mechanism would not, for reasons of cost and time to market.

As is shown in Figure 4, the alarm processing is carried out within two separate processes at two separate priorities. Both processes call the same reentrant alarm processing procedure. Alarm processing is not combined with the Analyze processes since Alarm's deadline is longer than the periods of the Analyze processes and thus could conceivably cause the Analyze processes to miss deadlines. Alarm processing is carried out within two processes (as opposed to a single process) to avoid non-preemptable access to the alarm processing, which could result in unnecessary delays to the processing of the 100 ms waveforms.

The 20 ms. deadline Send function was collapsed into the 100 ms. Analyze function, resulting in a process that we now call Analyze & Send. The 50 ms. deadline Audio function was collapsed into the 300 ms. Analyze function, resulting in the Analyze & Audio process. In both cases, no additional benefits are derived by putting these functions their own processes and so these were collapsed in the name of simplicity, and to save the overhead of the additional processes.

Key Design Tradeoffs

This design now contains several other tradeoffs that need to be paid attention to. Layer-bridging: as we described above, if Acquire-1 used the Publish/Subscribe mechanism to publish data every 10 ms. it would use up about 30% of the pro-

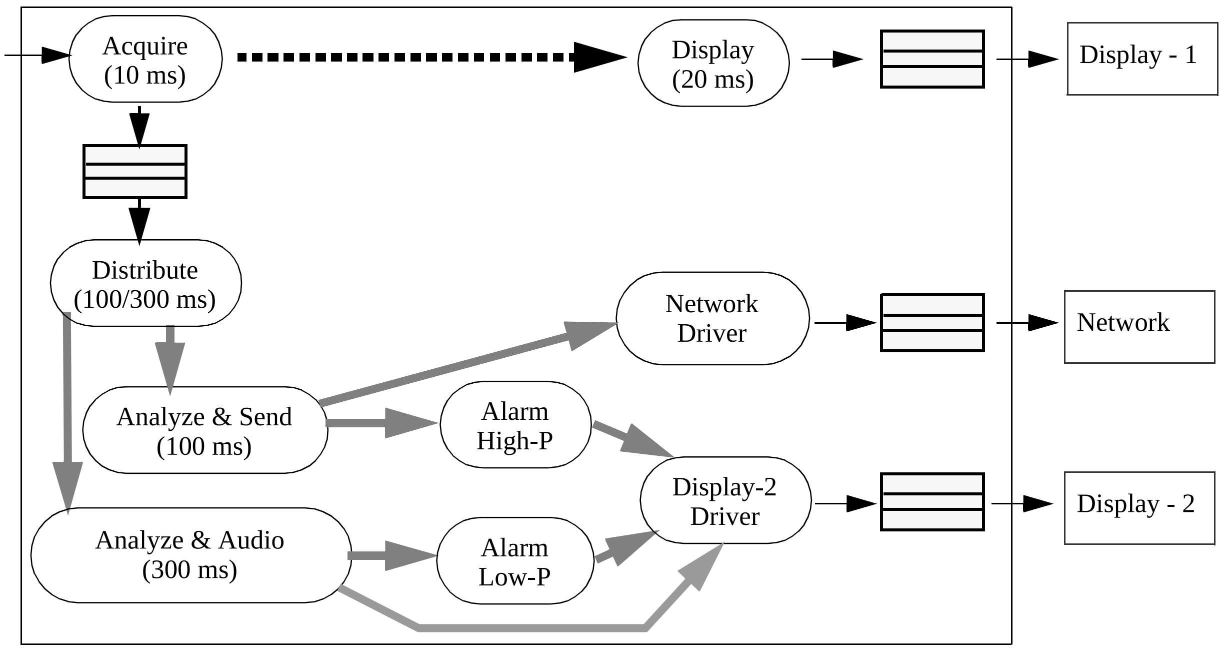


Figure 5: The Concurrency/Hardware View of the Patient Monitoring System Architecture

cessor's resources. This is a trade-off between portability and performance. By using the IPC Primitives layer directly, changes to that layer will ripple into application entities. We feel that this is an acceptable tradeoff because the performance arguments are compelling, and because we are controlling the harmful effects of the layer bridging by strictly limiting the use of IPC to this one data path.

Another trade-off favoring performance over modifiability occurs where the alarm function is split.

In using a Publish/Subscribe mechanism in favor of an ADT mechanism we sacrificed some performance in favor of modifiability.

5 RESULTS

We began this paper by arguing for the modularity and composability of the ABAS-oriented approach. ABASs provide individual styles associated with formal and heuristic analyses. An architect indexes into these by the stimuli to be processed, the responses to be controlled and by topology.

What Does ABAS-based Design Buy?

In using each ABAS an analytic framework is invoked, telling the architect how to control the responses. In addition, the ABASs' key design decisions are highlighted (typically realized as key analysis questions) so that the architect is alerted to the need to pay particular concern to these areas.

A design method should help in dividing and conquering a complex problem: dividing a problem into manageable chunks and then conquering the composition of those chunks. Dividing is typically easy. Combining and conquering is where the problems lie. ABASs guide us in dividing and helps to conquer by highlighting the key interactions among the styles and their analyses.

Consider the results of the experiment that we performed on the patient monitoring system: the chosen set of ABASs divided the problem for us into individually reasonable (but idealized) solutions. Our layering was unbridged. Publish/subscribe was used everywhere. Pipelines didn't interact. The conquering however didn't occur until those individually reasonable solutions are combined. Doing so forced us to wrestle with the tradeoffs that are found in the real world of design. The individual ABASs pointed us to the likely locations of these trade-off points and told us how to reason about them to ensure a solution that meets all of the requirements.

While we had access to some limited architectural documentation of the original design, we have now created a design for the patient monitoring system working entirely from the requirements and from ABASs. This is a real system, not simply an academic exercise. The result of our ABAS-based design is extremely close to what was finally implemented as judged by a designer who was involved with the original architecture design.

So what is the point of this exercise? The point is that by applying ABASs we were able to design *efficiently*, minimizing both *risks* and *costs*. And we designed with confidence, knowing that we were using proven techniques with analyzable properties.

Comparing to the Actual Design

So how well did we do, compared with the actual architecture of this system that was implemented? The short answer is that we did very well, finding all of the serious quality attribute issues in the architecture that the company had to face.

We will first address efficiency. We were able to create the broad outlines of the architecture (for a system of roughly 1,000,000 lines of code) in 1-2 hours, simply by matching ABASs to the requirements and looking at the likely problems in composing them. We completed the architecture in less than one week of work, including considering other architectural alternatives, such as the use of the Abstract Data Repository ABAS for data sharing, in place of the Publish/Subscribe ABAS. This is not to say that every design detail was chosen and documented in one week but rather that all key architecture decisions were made and analyzed within this time.

The system, as we've portrayed it, appears simple. That is due in part to us abstracting the problem, but it is primarily due to the way in which we look at the problem; specifically it is due to the way that ABASs divide up the problem into separately analyzable and solvable pieces. By cutting to the essence of key aspects of the problem we are able to turn a complex problem into a simple one.

ABASs also helped us minimize both risk and cost. By applying ABAS reasoning to the architecture we were able to pinpoint critical quality attribute problems, such as the need to split data acquisition into two processes. We predicted other problems as well, such as the need to bridge the layering scheme and bypass the Publish/Subscribe mechanism to achieve the desired latency characteristics. And we chose a set of processes and their priorities which will allow us to meet the system's real-time requirements.

As a result, we designed with confidence. Architectural design is a huge risk in any project: the wrong design can doom a project, despite heroic efforts by the implementors.

Another benefit of ABASs is that they facilitate communication among stakeholders and make the justification of architectural decisions more concrete. Architecture is a social activity: an architect must be able to communicate an architecture, including its benefits and pitfalls, to a wide group of stakeholders. Failure to get "buy-in" on a architecture can be just as devastating to a project's eventual success as a poor architecture.

6 CONCLUSIONS

In this paper we introduced the notion of an Attribute-Based Architectural Style; which is a relatively small capsule of design knowledge comprising an architectural style strongly coupled with an attribute-specific analysis style and a set of analysis questions and heuristics. ABASs are a style of reasoning about a specific quality attribute (such as performance or modifiability) closely tied to an architectural style. The goal is to reveal the consequences of the design decisions embedded in the style. Our contention is that a collection of ABASs covering a suitably large number of patterns and attributes can serve as the key building blocks for designing systems with predictable quality attribute behavior.

Why do we believe that ABASs can serve as key building blocks for designing systems? Over the past two years we have created a collection of ABASs and used them for design and for analysis in conducting architect evaluations [6]. They revealed key design decisions efficiently; revealed design alternatives quickly; focussed design discussions at the right level of abstraction; highlighted tradeoffs between attributes; provided the basis for documenting design rationale; and facilitated a separation and then composition of concerns. We have now used ABASs in a dozen analysis and design exercises with industrial systems. We grant that this is limited experience from which to draw conclusions, but we are (in politician speak) cautiously optimistic.

We have a number of reasons to be cautiously optimistic about ABASs. First of all, we feel that this type of documented and analyzed design is crucial for the formation of an engineering discipline of software, as is suggested by Shaw [11]. In this work, Shaw states that any mature engineering discipline contains a handbook that documents a set of “unit operations”. These patterns of design, accompanied with analyses, form the basis for all routine design and analysis within the engineering discipline.

ABASs are the basis for exactly the type of information that is needed in a software engineering handbook. They are very similar in nature to some of the key contributions to software engineering over the past twenty years. These key contributions explicitly or implicitly link design with analysis. That's exactly what ABASs do for architecture. Over the past twenty years some of the key contributions to software engineering have to do with techniques for increasing our understanding of and managing quality attributes. Parnas' articulation of the notion of information hiding was relating a style of design to modifiability. Knuth's compendium of algorithms connects detailed design with performance analysis (complexity).

The architectural styles and the analysis that we have documented are not original, but linking them together as composable capsules is original. Moreover, using these capsules as a vehicle for simplifying architectural design and analysis holds promise for introducing a new mind-set for software engineering, a mind-set which focuses attention on analyzing the ramifications of the architectural decisions in a modular and methodical manner.

7 REFERENCES

1. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley & Sons, 1996.
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns*, Addison Wesley, 1995.
3. Gonzalez Harbour, M., Klein, M., Lehoczky, J., “Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority”, *Proceedings of IEEE Real-Time Systems Symposium*, 1991, 116-128.
4. Hofmeister, C., Nord, R., Soni, D., *Applied Software Architecture*, Addison-Wesley, 2000.
5. Kazman, R., Abowd, G., Bass, L., Webb, M., “SAAM: A Method for Analyzing the Properties of Software Architectures," *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994, 81-90
6. Kazman, R., Barbacci, M., Klein, M., Carriere, S. J., Woods, S. G. “Experience with Performing Architecture Tradeoff Analysis”, *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999, 54-63.
7. Klein, M., Kazman, R., Bass, L., Carriere, S. J., Barbacci, M., Lipson H. “Attribute-Based Architectural Styles,” *Software Architecture: Proceedings of the First Working IFIP Conference on Software Architecture*, San Antonio, TX, February 1999, 225-243.
8. Klein, M., Kazman, R., “Attribute-Based Architectural Styles”, CMU/SEI-99-TR-22, Software Engineering Institute, Carnegie Mellon University, 1999
9. Rajkumar, R., Gagliardi, M., Sha, L., “The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation”, *Proceedings of the IEEE Real-time Technology and Applications Symposium*, June 1995.
10. Shaw, M., Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
11. Shaw, M., “Prospects for an Engineering Discipline of Software”, *IEEE Software*, 7(6), November 1990, 15-24.