Layers, Decisions, Patterns, Styles, and Architectures

Robert W. Schwanke
Siemens Corporate Research, Inc.
rws@scr.siemens.com

Abstract

The pattern-composition diagram *and the* attribute/decision graph *summarize an architecture and its rationale, respectively. This paper introduces, defines, and discusses these notations, with an example of a reference architecture for a broad class of real-time systems. The design patterns organizing the infrastructure of the example system would be highly recommended for most large systems today, and are therefore nominated as "foundational architectural patterns".*

1. Introduction

Current software architecture research is drawing upon and extending the work of the design patterns community to catalog standard architecture styles so that they can be widely disseminated, refined, and put into practice [4], [8], [19]. The Software Engineering Institute is collecting *attribute-based architectural styles* (ABASs), which are architectural styles designed to promote specific system attributes [12]. Bachmann *et al* [1] have proposed describing architectures by using ABASs to decompose a system, in an approach reminiscent of programming by stepwise refinement. Klein *et al* [14] have demonstrated this technique in a design exercise. Bosch has offered a methodology and process for software architectural design based on styles and patterns [3].

Such approaches needs effective ways to present the structural relationships between the different styles or patterns used and to summarize *why* each pattern was chosen and how it depends on other choices. The *pattern-composition diagram* merges the conceptual view and the module-layer view of a software architecture [11], annotating its elements with the patterns used, to give an overview of both the application and infrastructure architecture of a system as a composition of patterns. The *attribute/decision graph* is a notation, reminiscent of a decision tree, that summarizes the justification for each major design decision (often a pattern) by showing the sequence of required attributes and other design decisions that justify it.

These notations were created out of necessity, for a paper describing the *real-time event flow* architecture style [16]. This style is designed to promote real-time analyzability in systems that process real-world events. To show that the style is broadly useful, its description had to deal with variations in requirements and corresponding variations in choice of patterns. Both notations support *optionality* to help address this need.

This paper begins by briefly outlining the common requirements of real-time event flow systems. It then introduces the two notations, using the real-time event flow style as an extended example. It reflects upon the example to draw lessons about the roles of styles and patterns in architecture, about the *process* vs. the *product* of architectural design, and about reusable compositions of patterns and styles. Readers desiring further information about the real-time event flow architecture style should consult the earlier paper [16].

2. A Word About "Events"

The term *event* has many uses, with different meanings. In this paper, an event is a measurable phenomenon occurring at a specific time, or a data record associated with such a phenomenon. System-internal messages resulting from an event may also be considered events, if they are at least implicitly associated with the time of the event, and require timely processing. Some systems process events "as fast as possible", for example by using an interrupt-driven approach. Other systems process events "as predictably as possible", for example by using a pure polling approach and minimizing conditional execution. Many systems combine these two approaches. Without minimizing the differences, this paper refers to both types as event flow systems.

3. Requirements

The real-time event flow style is intended for systems that have most of the following requirements.

- *Event processing domain*. The system receives data from hardware sensors and also, typically, sends data to hardware actuators. (Pure instrumentation systems, without control, are included.)

- *Real time*. It must respond to external events within specified time limits and specified "jitter" (variation in response time). It may also have soft-real-time, non-real-time, and graceful-degradation requirements.
- *Software product line.* The software should support a fairly large number of systems with significant differences in functional requirements, which are nonetheless targeted for a coherent market and built out of substantial shared software assets [7]. This definition includes a software product that supports a "solutions business", in which a separate organization creates extensively tailored, "one-off" installations.
- *Evolvable*. The design must accommodate future changes in functionality and support technology.
- *Scalable*: The processing capacity requirements span a wide range of performance points.
- *Portable*: The product should be easy to move to a different operating system and hardware platform.
- *Dynamically reconfigurable*: Event processing steps must be reconfigurable at run-time.
- *Fault-tolerant*: failures of individual hardware and software elements should not bring the whole system down. MTBF for subsystems should be comparable to the MTBF for the physical equipment.
- *Available*: no scheduled down-time for the entire system, and bounded recovery time. Availability must be considered in two contexts: failure recovery and reconfiguration. Therefore, this paper treats this requirement in conjunction with those other two requirements, rather than separately.

4. Pattern-composition Diagrams

The term *software design pattern* was first popularized by Gamma *et al* [8], triggering a flurry of publications defining and naming new design patterns. This paper uses the term to also include architectural patterns [4] and well-known design structures that show promise to become standard patterns.

The *pattern-composition diagram* gives an overview of the system architecture, showing its decomposition into subsystems and layers, its conceptual views, and where various design patterns occur. (Because layering is the most common design pattern used, this paper sometimes refers to this type of diagram as a "layer diagram", where that usage is unambiguous.)

4.1. Event Flow Layers, Concepts, and Patterns

The real-time event flow architecture is summarized in a pattern-composition diagram in Figure 1. The following paragraphs survey the architecture "top-down", briefly discussing its design patterns in three groups. Some of the design patterns are discussed more extensively in later sections, as needed.

Foundational patterns. These patterns are used to build up a "comfortable" programming environment that addresses requirements found in most systems.

- *Platform Layer Pattern.* This pattern is a specialization of the Layer pattern [4] in which the lowest of a set of layers is a platform for the remaining layers. The platform layer provides the run-time environment and other common infrastructure shared by all products in the product line, and does not depend on particular application functionality.
- *Portability Layer Pattern*: This pattern is also a specialization of the Layer pattern, in which the lowest of a set of layers (in this case, the set of layers contained in the Platform layer) is a portability layer. The portability layer encapsulates many COTS infrastructure technology choices, including computers, operating systems, and distributed communication services (e.g. DCOM, CORBA). It also provides certain operating system services and distributed communication services in a portable way. The internal structure of a portability layer is largely dictated by the dependencies among COTS technologies.
- *Product Line Component Bus*: This pattern organizes the application as a collection of components connected by a software component bus. This bus implements domain-specific or product-line-specific types for components, connectors and containers, as well as any run-time mechanisms needed to manage their execution. It is typically built on top of a COTS component bus (cf. Section 9). Notice that this pattern implements the relationship between the *conceptual view* and the *module view* of Hofmeister *et al* [11], by effectively providing an interpreter for conceptual-view models.
- *Component Configurator*[15]: The configurator manages the application's components and connectors. Its functionality has two parts: policy and mechanism. The mechanism is implemented in the component bus, providing the capability to load, unload, suspend, and resume individual components and connectors. The policy part is a subsystem/layer, underlying the current configuration, which decides when to reconfigure and assures that the changeover takes place in the proper sequence and minimizes the time that the system is unavailable.

Core patterns. These patterns define the essence of the style: non-blocking components and one-way connectors processing real-time events under the control of a rate-based scheduler.

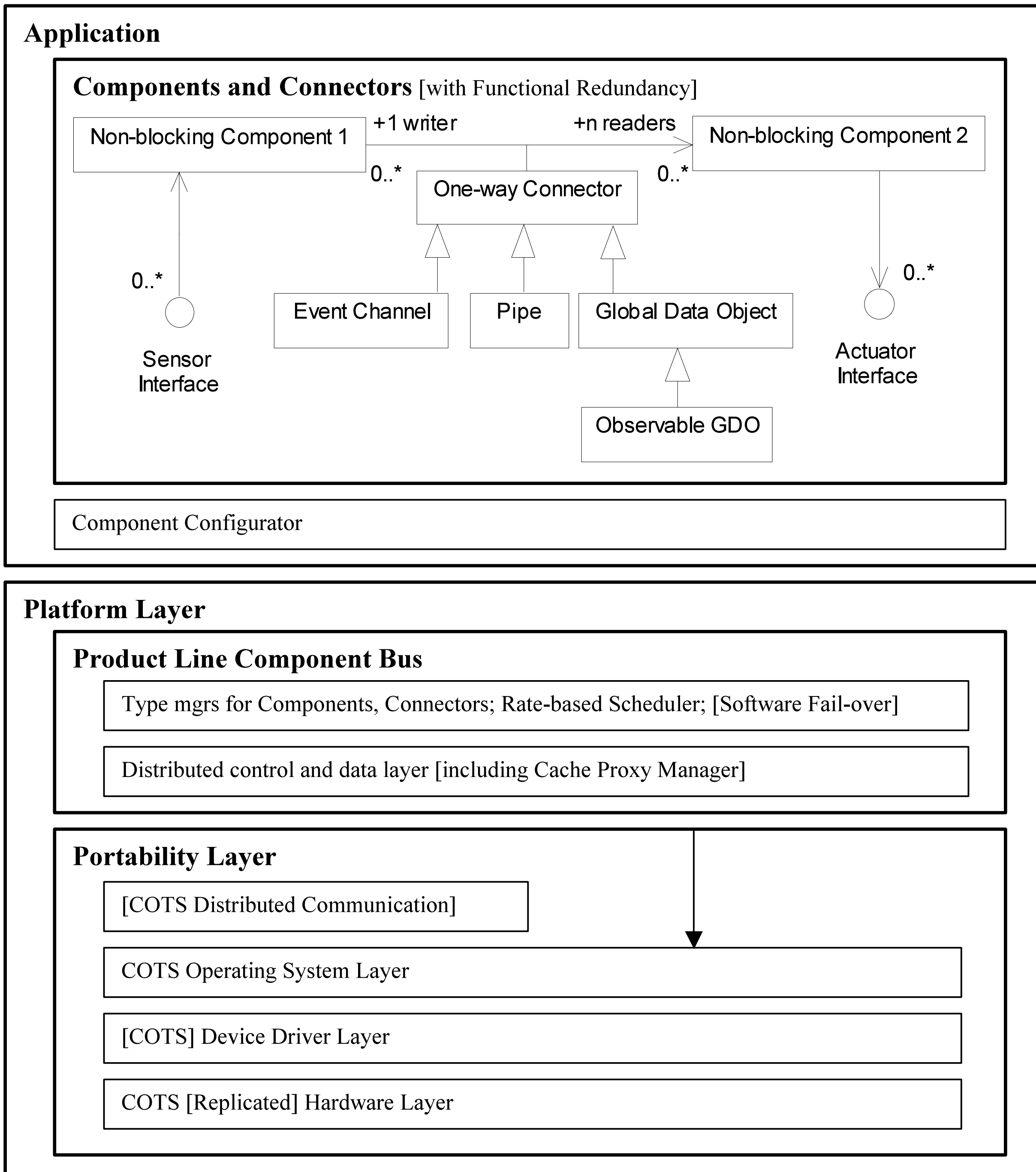

Figure 1 Pattern-composition Diagram

- *Non-blocking components*: Each component processes incoming event-messages one at a time, without blocking (except between events).
- *One-way connectors*: One-way connectors are first-class design elements in the conceptual architecture. They provide communication paths between components and decouple the control flows of the components. A one-way connector sends data from a single writer-component to one or more reader-components, guaranteeing delivery but providing no reply. Like components, connectors can have their own states, but their implementations are part of the component bus and may therefore be distributed across the system. Sub-patterns of one-way connectors:
  - *Pipes* [19]: streams of data that have a single writer and potentially multiple readers. Each reader receives all the data in the stream.

- *Event Channel*: a loosely coupled, easily distributed variant of Publisher-Subscriber [4].
- *Global Data Objects* (GDOs): A global data object, as defined in HealthyVision [11] and this architecture, is a real-time specialization of the publish-subscribe pattern. It is a distribution-transparent object with one writer and multiple readers. The writer posts changes to it as needed, and each reader *pulls* the information it needs, when needed. When no readers are subscribed to a GDO, write operations on it are suppressed to save time and space. Although global variables are normally considered harmful, GDOs are helpful because: having a single writer makes them one-way connectors; they appear explicitly in the architecture; and, the component bus can take care of synchronization. GDOs are a particularly nice way to communicate between components that execute at different rates.
- *Observable GDOs*: reader-components may register to receive notifications when global data objects change, according to the Observer pattern [8].

- *Rate-based Scheduling*: the components and connectors are scheduled for execution according to specified event arrival distributions, periodicity, latency, importance (different from priority), and other real-time parameters. The schedule may be computed statically, dynamically, or some combination thereof. One well-known rate-based scheduling method is Rate Monotonic Analysis [13]. Gill has recently added some interesting techniques [10].

Optional patterns. These patterns are used to address additional attributes often required of real-time event flow systems.

- *Cache Proxy*: this pattern, a variant of the Proxy pattern [4], is useful for selectively replicating distributed data, especially when there is a single writer and when the readers pull the data.
- *Hardware Replication*: the hardware may be completely replicated, so that every computation is performed multiple times, concurrently, to increase fault tolerance.
- *Functional Redundancy*: the system engineer for a particular application designs redundant application functionality and maps it to the execution architecture in such a way that failure of a software or computing component will not prevent the system as a whole from performing its required functions. (This technique requires platform support to avoid cascading failures.)
- *Software Fail-over*: the component bus may include a software fail-over scheme to increase fault tolerance. Sharp and Roll [18] have described a fail-over strategy that takes advantage of rate-based scheduling, the cache proxy pattern, and a component configurator.
  - The configurator installs multiple copies of each component, on different processors, but only activates one of each.
  - Connector data is replicated to all Proxies, even those read by inactive components.
  - A failure detection algorithm, such as a heartbeat algorithm, detects component or processor failure.
  - When a component fails, the configurator selects a backup component, makes sure it has the right inputs, and activates it.
  - The rate-based scheduler compensates for lost time.
  - Each component should be scheduled conservatively enough that it has time to fail, be reconfigured, and execute again before its hard deadline.

4.2. Syntax and Semantics

The primary syntax of these diagrams is that of nested layers, for example as used by Hofmeister *et al* [11] and Clements and Nord [5]. Each layer is a subsystem, represented by a horizontal rectangular box. Each box is a scope boundary, meaning that the box imports the resources it needs and exports some of the resources it implements. Each layer may import resources provided by lower layers. Nested boxes represent decomposition. Layer-bridging may be permitted, forbidden, or dependencies may be explicitly specified with arrows. Labels are uninterpreted, but are conventionally used to describe the functionality and to identify the design patterns that are used in the architecture. [Bracketed] text in labels denotes optional items.

In addition, each layer box can contain a subdiagram drawn in a different syntax, suitable to the design pattern used to decompose that box. In the example in Figure 1, the Components and Connectors layer is elaborated using a conceptual view notation [11] rather than a layer notation. It shows sensors and actuators connected to non-blocking components, which are interconnected by one-way connectors. These connectors may be event channels, pipes, global data objects, or observable GDOs. There can be any number of connectors between components, but each connector has one writer and n readers. Although not *always* needed in an architecture overview, if the system includes such a components-and-connectors style, showing it in the overview illuminates the relationship of this style to the rest of the architecture. Further discussion of component-and-connector notations, and of relationships between views, is beyond the scope of this paper.

Although the real-time event flow example does not require it, other subsystems could be elaborated using other pattern-related notations.

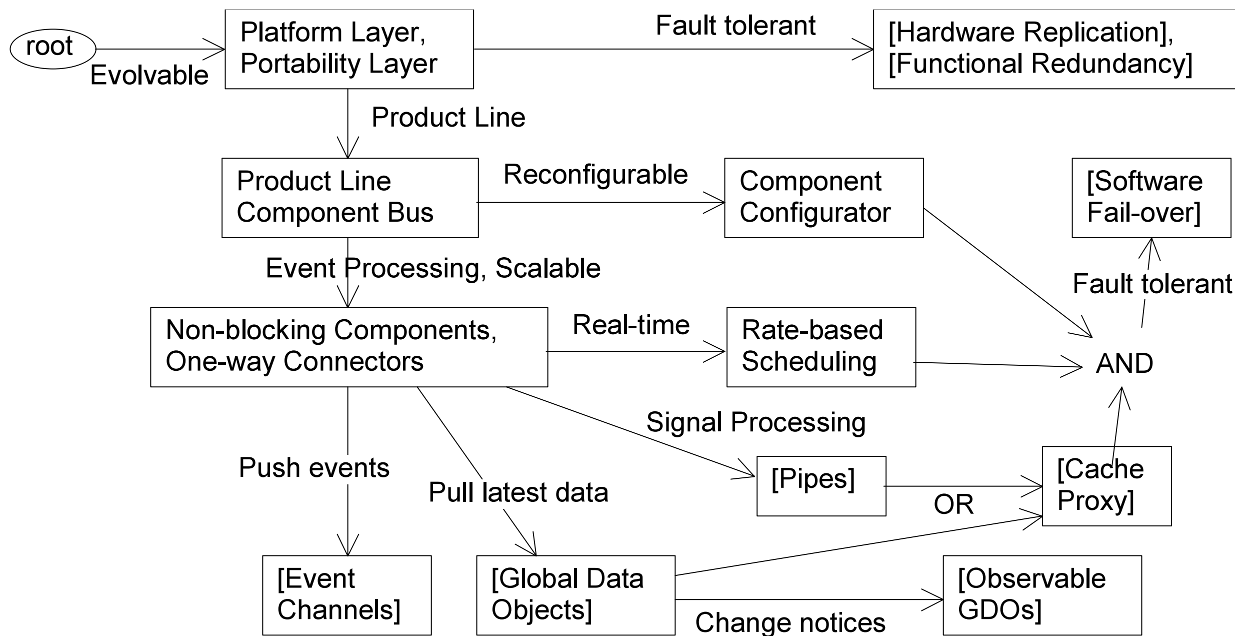

Figure 2 Attribute/Decision Graph

4.3. Options and Configurations

The pattern-composition diagram represents variations in the architecture in several ways. [Square brackets] delimit optional parts of the architecture. Where the whole subsystem name is bracketed, that subsystem is optional. Where only part of the name is bracketed, multiple alternative variants of the name are intended. For example, "[COTS]" means that the subsystem might be COTS or might be developed in-house.

In the conceptual view sub-diagram, the components-and-connectors scheme defines a potentially large class of individual products that could be built, by building a collection of specific components and configuring them. The relations and cardinality constraints restrict how the various components and connectors can be hooked together. The text of the architecture specification can of course impose additional constraints, such as scale constraints and deep semantic constraints, for example in the form of a *decision model* [20].

5. Attribute/decision graph

The *attribute/decision graph* summarizes the rationale for a chosen architecture, showing the principal dependencies among required system attributes and the design decisions that promote those attributes. It summarizes the content of the requirements traceability matrix and the matrix of dependencies between design decisions, but does so creatively, to emphasize what the architect thinks is important.

Each edge of the graph represents a refinement of the design, in which zero or more additional attributes are addressed by adopting one or more additional decisions. Although it looks somewhat like a decision tree, it is not: an individual realization of the event flow architecture may span most or all of the entire graph, rather than being restricted to a linear path through the graph, as would be true in a decision tree.

5.1. Event Flow Attributes and Decisions

Figure 2 relates the attributes and the design patterns of the real-time event flow architecture. Informally the graph should be read as follows:

- To make the system *Evolvable*, it has a *Platform Layer* and a *Portability Layer*.
- Because the system is also a *Product Line*, it has a *Product Line Component Bus*.
- To also make it dynamically *Reconfigurable* (and/or Available), it has a *Component Configurator.*
- Because it is also doing *Event Processing* and/or needs to be *Scalable,* it uses *Non-blocking Components* and *One-way Connectors*.
- To also meet hard *Real-time* requirements, it uses *Rate-based Scheduling*.
- To *Push events* it uses *Event Channels*.
- For *Signal Processing* messages, it uses *Pipes*.
- To *Pull latest data*, it uses *Global Data Objects.*
- To send *Change notices*, it uses *Observable GDOs*.
- To distribute data for *Pipes* and/or *Global Data Objects*, it uses the *Cache Proxy* pattern.
- Systems that use *Cache Proxy*, *Rate-based Scheduling*, and a *Component Configurator*, and must be *Fault-tolerant,* can use the *Software Fail-over* scheme.

- *Fault-tolerant* behavior can be achieved with *Hardware Replication* and/or *Functional Redundancy*.

5.2. Syntax

The attribute/decision graph is a directed, acyclic graph with root, such that every node in the graph is reachable from the root.

The nodes are design decisions; in this paper they are mostly design patterns. The edges are labeled with desired system attributes. Nodes and edges can both have multiple labels. Edges can be unlabelled.

Two or more out-going edges from a node can be marked "XOR" or "OR", indicating whether the alternatives are mutually exclusive or not, respectively. Unmarked outgoing edges are treated as "inclusive-OR" by default. Similarly, two or more incoming edges to a node can be marked "AND" or "OR", indicating whether the node represents an "AND-join" or an "OR-join". "OR-join" is the default.

5.3. Semantics

Each node in the graph represents a derivation step in the architecture, in which a design decision was made based on a set of required attributes and preceding design decisions. However, the node depends on more than just its immediate incoming edges and predecessor nodes; it potentially depends on *all* preceding nodes and edges along *all* paths from the root to the node.

For example, in Figure 2, the rationale for the Component Configurator comes not only from the reconfigurability requirement and the decision to use a product line component bus, but also from the product line and evolvability requirements and the decisions to use a platform layer and a portability layer.

To be more precise requires some formal notation. The subgraph consisting of all nodes and edges transitively preceding a certain node, X, is the *backward slice* of X, denoted "P^*_X", where P suggests "predecessor" and "*" is the transitive closure. P^*_X includes *X* itself. Such a subgraph, (which, in these graphs, always includes the root), corresponds to a set of assertions about attributes and decisions, e.g. "Requires(attribute)" and "Implements(decision)". An edge from node A to node C with label B is denoted $A \rightarrow_B C$. The rationale for node C based on the edge labeled B is denoted as "$\text{Rationale}_B(C)$" (note the subscript, B).

Using this notation, consider four cases: single predecessor, OR-join, AND-join, and unlabelled edge.

Single predecessor. Consider a labeled edge, $A \rightarrow_B C$, where A is the only node immediately preceding C. The edge $A \rightarrow_B C$ means, "Design decision C is justified by the assertions corresponding to P^*_A plus the additional requirement B." More formally,

$$\text{Assertions}(P^*_A) + \text{Requires}(B) \supset \text{Rationale}_B(C)$$

OR-join. Consider two labeled edges, $A \rightarrow_B C$ and $J \rightarrow_K C$, either labeled "OR" or unlabeled ("OR" is the default.) Together, they mean that the node C has two rationales:

$$\text{Assertions}(P^*_A) + \text{Requires}(B) \supset \text{Rationale}_B(C)$$

and

$$\text{Assertions}(P^*_J) + \text{Requires}(K) \supset \text{Rationale}_K(C)$$

(Note the different subscripts on the two rationales.)

In the example system, the Cache Proxy pattern is OR-joined from Global Data Objects or Pipes, with no additional required attributes. Using either of these two patterns in a Scalable design implies "implementing large, single-writer objects in a distribution-transparent manner", which is the rationale for the Cache Proxy pattern.

AND-join. Consider the same two labeled edges again, but this time suppose that where the edges meet node C, they are marked "AND". This AND-join, "$A \rightarrow_B C$ AND $J \rightarrow_K C$", means that the (single) rationale for C derives from both predecessors and the merge of their backward slices. More formally,

$$\text{Assertions}(P^*_A \cup P^*_J) + \text{Requires}(B + K) \supset \text{Rationale}_{B,K}(C)$$

In the example system, the Software Fail-over node is AND-joined from three other nodes. The rationale for the fail-over mechanism includes the fact that it can be implemented cheaply in systems that already use Cache Proxy, Rate-based scheduling, and a Component Configurator. Therefore, the diagram shows its rationale as derived from the conjunction of all three mechanisms.

Unlabelled edge. The unlabelled edge $A \rightarrow C$ means, "Design decision C is justified by the assertions corresponding to P^*_A, without additional requirements." More formally,

$$\text{Assertions}(P^*_A) \supset \text{Rationale}(C)$$

Meanings of identifiers. Note that the identifiers appearing in the attribute/decision graph all obtain their meanings from the natural language text accompanying the diagrams. It is tempting to argue about the rationales embedded in this graph, independent of the rest of the architecture. This is nonsense, of course, both because the locations of the design patterns in that diagram add constraints, but also because the attribute/decision graph is itself only a summary of the design rationale of the architecture. So, for example, the assertion "Implements(One-way Connectors)" actually means that the system imple-

ments them in the manner and context in which they are described in the pattern-composition diagram and associated text.

5.4. Optionality

As in the layer diagram, some of the labels in the attribute/decision graph are [square-bracketed], denoting optionality. However, because each node must be reachable from the root, an optional node may only be omitted when doing so will not make other pattern nodes unreachable. For example, in Figure 2, both “Global Data Objects” and “Observers on GDOs” are marked as optional, but it would be impossible to have “Observers on GDOs” if there are no “Global Data Objects”, so the “Global Data Objects” pattern cannot be omitted unless the “Observers on GDOs” is also omitted.

To fully support variation points and variation parameters, the notation could perhaps be extended with an exclusive-OR split construct, where the alternative outgoing edges would be labeled with predicates involving variation parameters from a *decision model* [20]. Further speculation along these lines is beyond the scope of this paper.

6. Examples

The event flow architecture was inspired by three very different event flow applications with very similar architectures: Safety Vision [11], Healthy Vision [11], and Bold Stroke OFP [6] [17]. Each system has an architecture that can be derived from the common architecture by omitting some of the optional patterns. The following table compares them. The interested reader could photocopy the earlier figures 3 times (once for each system) and mark them up accordingly, to see the differences.

Although I did not have access to the actual rationales for these systems, the omitted attribute requirements and design patterns are all compatible with the attribute/decision graph of the common architecture, with one exception: Bold Stroke OFP has a dynamic component configurator even though it has no explicit requirement for dynamic reconfigurability. This design decision was made because the fault-tolerance requirement is met using a software fail-over scheme that needs dynamic reconfigurability to minimize the time the system is unavailable.

6.1. SafetyVision

This power plant control system [11] compiles code from function block diagrams to create control systems. The function blocks are non-blocking components and the signals are one-way connectors. The components are collected into containers (manually grouped in the design) and executed on virtual machines that make them completely portable to other execution platforms without recompilation. Containers exchange signals via one-way telegrams, which are read according to the global processing schedule rather than being read immediately on arrival. Since the system is required to behave deterministically, all components are periodic and poll all input signals and execute all code in every period, no matter how many times the input signals have or have not changed. This gives the signals GDO semantics, without observers.

A single master schedule is used across all distributed processors, synchronized by a global clock. Within a container, a topological sort of the components determines the execution order. Fault tolerance is achieved with functional redundancy in the design.

System Attributes and pattern	**Safety Vision**	**Healthy Vision**	**Bold Stroke OFP**
Domain	Power Plant	Patient Monitor	Mission Avionics
Evolvable	yes	yes	yes
Platform Layer	yes	yes	yes
Portability Layer	yes	yes	yes
Product Line	yes	yes	yes
Component Bus	yes	yes	yes
Reconfigurable		yes	
Configurator		yes	yes
Event Processing	yes	yes	yes
Scalable	yes	yes	yes
One-way Connectors	yes	yes	yes
Event Channel			yes
Signal Processing		yes	Pre-processor[1]
Pipes		yes	
GDOs	yes	yes	yes
Observable GDOs		yes	
Hard Real-time	yes	yes	yes
Rate-based Scheduling	Manual	Static RMA	Static and Dynamic
Cache Proxy			yes
Fault Tolerant	yes	yes [2]	yes
Functional Redundancy	yes		
Hardware Replication			
Software Fail-over			yes

[1] A separate computer pre-processes signals.
[2] Used fast reboot to recover from fault.

6.2. HealthyVision

This is a patient monitoring product line [11] that performs a combination of signal processing and event/alarm processing. Some of its inputs are waveforms (e.g. heart monitors), whereas others sample simpler sensors (e.g. pulse, oxygenation, temperature).

The system has two classes of real-time latency constraints. Waveforms must be updated on the display screen with high enough frequency and low enough jitter to appear to move smoothly across the screen. Alarms and numeric sensor values change much less frequently, and jitter is less critical.

This is the project that coined the term *global data object*. Most components are connected by GDOs, with observers. All execution is event-driven, rather than being periodic. However, all time-critical event streams are periodic, anyway, so the receiving components also wind up being periodic.

Signal processing requirements are stringent enough that it uses a hardware signal processor as well as one to three general-purpose CPUs. Waveforms are transmitted to the display screen via a path that uses special purpose pipes. These pipes use a shared-memory implementation mechanism and a push protocol for maximum performance. Waveforms are also transmitted via GDOs to other components that have less stringent latency requirements.

6.3. Bold StrokeOFP

This Operational Flight Program for jet fighters [6][17][18] is apparently the first commercial user of TAO, the ACE ORB that prioritizes its communication using rate-based scheduling to meet real-time latency requirements. Although it has no requirement for functional reconfigurability, it uses an ACE-based partitioning configurator to support its fail-over mechanism.

7. Variation by Extension and Contraction

In addition to the optionality already representable in the layer and decision diagrams, it is interesting to note what kinds of extension and contraction are easy to do in the architecture.

The attribute/decision graph is the first place to look for extension and contraction. The graph can be extended by adding labels, nodes or edges, and can be contracted by making a mandatory node optional, or by deleting any node that has no successors. (Deleting a node includes, of course, deleting its incoming edges.) In addition, "semantic contraction" can be achieved by replacing a label with a less-specific label, as long as doing so does not compromise the rationales of subsequent nodes. For example, the label "Rate-based Scheduling" could be rewritten as "[Rate-based] Scheduling, or just "Component and Connector Scheduling".

Decision graph changes would require corresponding changes to the layer diagram. Existing boxes on the diagram can be refined to add more subsystems or patterns, and leaf-level boxes can be deleted. Label changes on the decision graph would have to be reflected in corresponding rewrites of labels in the layer diagram.

8. Patterns, styles, and architectures

The patterns used in the example architecture can be usefully classified into three groups: core, optional, and foundational patterns. This classification illuminates the differences between a pattern, a style, and an architecture.

8.1. Core Patterns of the Style

The seven core patterns, together, define the essence of the style: non-blocking components and one-way connectors, subject to rate-based scheduling, make an event flow system analyzable with respect its real-time requirements. "One-way Connectors" is a "pure virtual pattern", with at least four sub-patterns: pipes, global data objects, event channels, and observable GDOs. Although each individual sub-pattern is optional, any realization of the style has to use at least one sub-pattern.

The difference between a style and a set of patterns is the synergy among the patterns. In this case, rate-based scheduling would not achieve real-time analyzability unless the components are non-blocking and the connectors are also subject to scheduling. A non-blocking component can only use connectors with non-blocking writes, such as one-way connectors. The implementation of one-way connectors depends on whether the components are cyclic or event-driven. If the components and connectors are scheduled, messages (particularly inter-processor) can be sent in batches, for efficiency. Thus it is the synergistic combination of patterns, plus the desired attribute (real-time analyzability) that makes the real-time event flow style more than a set of patterns.

8.2. Optional patterns to specialize the style

Two optional patterns can be used to specialize the style: cache proxy, and software fail-over. Although the cache proxy pattern could be used in any system where distribution transparency is desired, it is most useful for objects that have a single writer (subject to reconfiguration) and readers that pull. It is also well suited to scheduled communication, since that simplifies synchronization of the master and proxy copies of the data. Thus, the cache proxy pattern takes advantage of properties of the core pattern.

Similarly, the software fail-over pattern builds on the cache proxy, rate-based scheduling, and component configurator patterns, composing these capabilities with some additional functionality to achieve fault tolerance. Although fault tolerance can also be achieved by hardware replication, independent of the event flow style, the fact that the real-time event flow style already supports the cache proxy approach and rate-based scheduling makes the software fail-over approach attractive. Therefore, it seems that these patterns do not represent new styles composed with the base style, but simply add-on patterns.

8.3. Foundational patterns to support the style

The remaining seven patterns of the architecture are all relevant to the event-flow style, but not dependent on it. The platform layer, portability layer, and product line component bus patterns are pre-requisites for implementing the style. (It is hard for this writer to imagine a good alternative way to implement it.) The component configurator must be implemented with style-specific semantics in order to safely carry out dynamic reconfiguration. Hardware replication and functional redundancy are "negatively" relevant as alternatives to the software fail-over pattern.

However, the main requirements that these patterns address are domain-independent ones, which real-time event flow systems share with many other kinds of systems. Evolvability, product lines, fault tolerance, and reconfigurability are significant issues for most large systems being designed today, whether or not they are acknowledged as key requirements. This part of the architecture suggests three conjectures:

- An architecture is often organized around one principal, usually domain-specific, style, but uses additional patterns to support other generally-desirable, domain-independent attributes.
- Some attributes are so commonly required that one can begin to identify "foundational" architectural structures to realize them. For example, "Layering" is a design pattern, but "Platform Layer" and "Portability Layer" are foundational architectural structures.
- Where an architecture employs several of these foundational structures, they are likely to appear in the same relationship to one another, making their composition a candidate for a handbook entry.

Further investigation is needed to explore these conjectures.

9. Component Models and Containers

The product line component bus used in the real-time event flow architecture raises questions about how different product line component models need to be from currently-available COTS component models.

The example systems highlight several attributes that can distinguish component models, including component size, threading model, communication mechanisms, scheduling, and dynamic reconfiguration. COTS component models, aimed at mass markets, typically adopt a one-size-fits-all philosophy. In contrast, a product line component model can be tailored to the needs of a much smaller market. This is why it is necessary to build a product-line-specific component bus that implements the desired component model on top of COTS technologies.

9.1. Real-time vs. COTS Component Models

This section contrasts the typical characteristics of real-time component models with current COTS offerings, e.g. DCOM, CORBA, and Enterprise Java Beans. As of this writing, the Real-Time Standard for Java (RTSJ) does not have a true component model, and so is not included in the comparison.

Component size. Real-time function blocks are often as small as a few hundred lines of code, whereas COTS components are typically expected to be much larger.

Threading model. Real-time components are event-driven and are forbidden from using system services that may block, whereas COTS component models allow for component blocking.

Communication mechanisms. Real-time components use a variety of one-way connectors. Current COTS component buses provide remote method invocation, event broadcast, and sometimes event channels.

Scheduling. Real-time schedulers take advantage of the connection topology and the real-time specifications of components and connectors to construct efficient schedules. COTS component buses assume the worst case: dynamically changing topologies, with component blocking, and aperiodic execution.

Dynamic reconfiguration. Real-time systems may need reconfiguration without interruption of service, requiring a careful dance between the scheduler and the configurator. COTS component buses define connectionless protocols (e.g. RMI) and connection objects (e.g. EJB Session Beans), but may not support non-stop, real-time reconfiguration.

9.2. Containers: Component Model Adapters

A large, distributed real-time system can still take advantage of COTS component technologies by defining container object types that adapt a COTS component model to the needs of the product line.

A container is a component satisfying a COTS component model that provides the infrastructure specified for a

more specialized component model, such as for a product line. For example, a container could be a CORBA 3.0 component that executes a sub-graph of a real-time event-flow application. Within that sub-graph, the components might be statically scheduled to execute sequentially in a single thread, with dynamic scheduling only affecting the order in which different containers are executed. If the sub-graphs are well chosen, communication between containers can be efficiently "batched", using the COTS communication mechanisms.

9.3. Deployment

A real-time event flow system deployment is can be conveniently described by a binding of components to containers and containers to computing resources. The deployment may be entirely static, for example when functional redundancy is used to achieve fault tolerance. However, when dynamic reconfigurability is required, the deployment description may only specify the number and configuration of execution elements, perhaps augmented by constraints on permissible bindings of components to containers and containers to execution elements. The remaining bindings are established by the component configurator at load time and reconfiguration time.

10. Architecture Design Methods

The following comparisons sample current work in pattern-oriented software architecture methods.

10.1. Bosch's *Software Architectural Design*

Jan Bosch recommends an iterative architectural design process [3] that creates an initial, functionality-driven architecture and applies a sequence of transformations to it to improve its non-functional attributes. If the real-time event flow architecture had been developed according to this process, the sequence might have gone something like this:

1. Real-time event flow style: non-blocking components and one-way connectors with rate-based scheduling.
2. Platform Layer architecture pattern
3. Portability Layer architecture pattern
4. Product Line Component Bus
5. Event Channels, Global Data Objects, Pipes, and/or Observable GDOs.
6. Cache Proxy
7. Component Configurator
8. Software Fail-over, Functional Redundancy, and/or Hardware Replication.

However, that would be rewriting history, or as someone has described it, "shortening the path behind me." I do not know the actual design process for any of the real examples. Furthermore, my experience reflects a less orderly process than such a sequence would seem to indicate. An architect, faced with a set of required attributes and a set of architectural styles and patterns that address them, must solve the puzzle of finding a way to make the styles and patterns work together. While I agree with the basic nested looping structure of Bosch's process, in practice the inner loop actually addresses a *set* of under-supported attributes with a *set* of additional design patterns. The transformation that takes place may actually discard a previously-chosen pattern in favor of ones that together do a better job.

Although pattern-composition diagrams show two partial orderings among design decisions (decomposition and dependency), these do not necessarily reflect the order in which the decisions were made, as the hypothetical sequence above illustrates. Similarly, attribute/decision graphs show a partial ordering among attributes and decisions, but these are logical dependencies, not temporal orderings.

Bosch's initial sub-process to create a functionality-driven architecture amounts to selecting and specializing a components-and-connectors style [19] or a conceptual style [11]. However, where he regards only the components as archetypes, I would regard the connectors as archetypes as well, in that there are significant choices among connector types for the same component types, depending on properties of the data being transmitted. Also, where he regards connectors as first-class design entities only if they are implemented as first-class objects, I would regard one-way connectors as first-class design entities, independent of their implementation, because they affect topology, schedulability, and performance analysis.

10.2. The *Architecture Based Design Method*

Bachmann et al. [1][2] address the architecture based design (ABD) method to the problem of creating a conceptual architecture that satisfies the principal architectural drivers found in the requirements. They note that the drivers will generally lead to a "dominant architectural style" for each design element. The real-time event flow *architecture* reflects this insight, in that the real-time event flow *style* is the dominant style for the layer sub-titled "conceptual view". Furthermore, the nested layers of the pattern-composition diagram appear to conform to the recursive decomposition structure of the ABD method.

However, the ABD decomposition structure is intended to fall entirely within the conceptual view, addressing user-visible functionality. The ABD method addresses infrastructure only to the extent of identifying properties for the component templates that will make them fit the platform interface, whereas Figure 1, for example, covers the infrastructure architecture as well as the conceptual

architecture. Incidentally, despite the ABD method being described as a recursive *process*, it appears to be an *iterative*, *opportunistic* process that produces a recursive *design structure*.

10.3. The SEI ABAS-based Approach

Klein *et al*. [12] recommend synthesizing an architecture by combining several attribute-based architectural styles. They have demonstrated the technique with a "re-design" exercise [14] on the patient monitoring product, "HealthyVision", later used in the present study.

Because the real-time event flow architecture uses fifteen patterns, compared to the three styles used in the re-design exercise, we can see the importance of documenting and analyzing the dependencies among attributes and patterns, as well as the difficulty of fitting the patterns together without compromising the attributes they promote. We can also see that design patterns need not be composed "from scratch" for each project, but that at least the foundational patterns and the conceptual architectural style have standard compositions that can be reused.

11. Conclusions

The real-time event flow architecture example shows that the pattern-composition diagram and the attribute-decision graph are useful for summarizing an architecture, for showing the relationships between the design patterns and conceptual style used in the application and their support in the platform, for describing simple variations, for comparing similar architectures, for speculating about extensions and contractions of the architecture, and for distinguishing the structure of the architecture from the structure of the process that produced it. The architecture exhibits a single dominant style, augmented by optional patterns that specialize it and foundational patterns that support it. The decision graph shows that there is an ordering among the patterns that gives guidance in applying them.

12. Acknowledgements

Many thanks to Rod Nord for his extensive suggestions.

13. References

[1] Bachmann, F., L. Bass, G. Chastek, P. Donohoe, F. Peruzzi. The Architecture Based Design Method. CMU/SEI-2000-TR-001, Pittsburgh: Carnegie-Mellon University, 2000.

[2] Bachmann, F., L. Bass, M. Klein. An Application of the Architecture-Based Design Method to the Electronic House. CMU/SEI-2000-SR-009, Pittsburgh: Carnegie-Mellon University, 2000.

[3] Bosch, J. *Design and Use of Software Architectures.* New York: ACM Press and Addison-Wesley, 2000.

[4] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture: A System of Patterns*. Chichester, UK: John Wiley & Sons, 1996.

[5] Clements, Paul, and Nord, Robert. "Documenting a Layered Software Architecture", in ISAW4 ???.

[6] Doerr, B. S., D. Sharp, "Freeing Product Line Architectures from Execution Dependencies", in [7], pp. 313-330.

[7] Donohoe, P., ed. Software Product Lines: Experience and Research Directions. Boston: Kluwer Academic Publishers, 2000. Proceedings of the First Software Product Lines Conference (SPLC1), August 28-31, 2000, Denver, Colorado, USA.

[8] Gamma, E. R., R. Helm, R. Johnson, J. Vlissides. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.

[9] Georg, G., S. Seidman. "The Use of Architecture Description Languages to Describe a Distributed Measurement System", in *7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems,* Edinburgh, Scotland, 3 - 7, April, 2000.

[10] Gill, C., D. Levine, D. C. Schmidt, F. Kuhns, "The Design and Performance of a Real-Time CORBA Scheduling Service", International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware, guest editor Wei Zhao, 2000.

[11] Hofmeister, C., R. Nord, D. Soni. *Applied Software Architecture*. Reading, MA: Addison-Wesley, 2000.

[12] Klein, M., R. Kazman, *Attribute-Based Architectural Styles*. CMU/SEI-99-TR-022, Pittsburgh: Carnegie-Mellon University, 1999.

[13] Klein, M. H., T. Ralya, B. Pollack, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Boston: Kluwer Academic Publishers, 1993.

[14] Klein, M. R. Kazman, R. Nord. *A BASis (or ABASs) for Reasoning About Software Architectures.* In preparation.

[15] Rosa, F. A., A. R. Silva. "Functionality and Partitioning Configuration: Design Patterns and Framework" in *IEEE Proceedings of the Fourth International Conference on Configurable Distributed Systems*. Annapolis, Maryland, USA, May 1998.

[16] Schwanke, R. W. "Toward a Real-time Event Flow Architecture Style". *IEEE Int'l Conference and Workshop on Engineering of Computer-based Systems*. April 2001.

[17] Sharp, D. "Component-Based Product Line Development of Avionics Software", in [7], pp. 353-370.

[18] Sharp, D., W. Roll. Correspondence with the author, September, 2000.

[19] Shaw, M., and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice Hall, 1996.

[20] Weiss, D. M., and Lai, C. T. R. *Software Product-Line Engineering*. Reading, MA: Addison-Wesley, 1999.