An Architectural Style for Object Oriented Real-Time Systems

Jose L. Fernandez
Arquitectura e Innovación Tecnológica. Caja de Madrid, Las Rozas, 28230 Madrid, Spain. fernandezjl@acm.org

Abstract

*Domain specific* software architectures are one of the most relevant products of domain modeling. A new architectural style, with its corresponding architectural elements and constraints, is proposed. The style matches the constructive principles of the object oriented and real-time communities, such that design decisions can be evaluated based on mathematical analysis of real-time behavior previous to testing activities. The style is applied in a specific domain, real-time data acquisition and alarm monitoring of industrial processes. Its application to other object oriented systems with concurrent paths of sequential processing, is also possible.

*Keywords*: software architecture, real-time systems, architecture characterization, architecture notation.

1. Introduction

Domain analysis is the key to systematic reuse. It is through domain analysis that knowledge and experience about a family of systems or domain, is transformed into generic specifications, designs and architectures, representing the different abstractions of the domain model.

Domain specific software architectures are one of the most important products of the domain models. The software architecture describes the overall structure of a system. Structural properties include overall organization and global control structure; protocols for communication, synchronization, and data access; assignment of functional requirements to design elements; physical distribution; composition of design elements; scaling and performance. A domain specific software architecture supports also the implementation of the commonalties and variability of a family of systems in a particular domain.

Unfortunately, current object oriented notations and methodologies do not support adequately the design of architectures for real-time systems. This is due to the lack of modeling of concurrent activities and the lack of design information necessary to evaluate the impact of design decisions on real-time behavior.

Previous considerations imply the necessity of developing an architectural style to be used in real-time object oriented architectures. As defined by Perry, an important aspect about an architectural style is that it encapsulates important decisions about the architectural elements and emphasizes important constraints on the elements and their relationships [14]. The similarities with building architecture are obvious.

A new style for concurrent object oriented architectures is proposed. It can be used when individual paths of execution are required to be concurrent and multiple processes may be positioned along the path to control the action. Then each process controls a segment of the path sequentially. So we call the style PPOOA, ‘Pipelines of Processes in Object Oriented Architectures.’

The proposed style is described using constructive elements such as components, coordination mechanisms and sequences. Constraints on architectural elements are represented by guidelines. The guidelines not only represent the semantics of the style, they are also helpful to the novice software architect using the style. A graphical notation considering the architectural elements and their interactions is also proposed.

The style represents a new approach in developing real-time systems. It allows the evaluation of behavior through the use of a mathematical technique called Rate Monotonic Analysis [12]. The rigorous modeling of sequences and activities is fundamental for this evaluation. Contrary to other approaches [15], we emphasize the notion of “activity” instead of “state”, allowing the direct evaluation of real-time behavior.
This paper describes the proposed style and its application to a specific domain, real-time data acquisition and alarm monitoring of industrial processes. Section 2 briefly represents the foundations of the style based on previous research of the real-time and object-oriented communities. Section 3 is the description of the architectural elements identified for the style. Section 4 discusses the constraints of the style. Section 5 describes the architecture characterization and which are the architectural views emphasized in the style. Section 6 gives samples of the application of the style to the example domain. Section 7 presents conclusions and discusses future directions.

2. Foundations of the Style

There are several research activities in the 00 and Real-Time arena which constitute the foundations of the architectural style presented. The foundations of the style are:

- Domain Analysis. The “Features Model” produced by FODA [11] is used to develop a model of the properties of the coordination mechanisms proposed as connectors in the architectural style.

- Behavior models. “Use Cases” [10] are the basis of the requirements modeling; “Stereotype” concept [19] help in the construction of the vocabulary of components; and “timethread” [21] is the basis of defining sequence as causality flow of activities.

- Explicit concurrency. Contrary to other 00 approaches, concurrency is modeled explicitly. The explicit model describes concurrency external to the object structure. The parallelism sits on the top of the object structure rather than being integrated into it [4]. The explicit model requires two abstraction levels components and sequences. Coordination mechanisms are essential for it. This model is well supported in Ada and Java.

- Rate Monotonic Analysis (RMA). RMA is a mathematical approach that helps ensure that a real-time systems meets its performance requirements. Real-time properties of the style components and sequences are identified based on RMA [12].

- Architectural Description Languages. The approach followed in UniCon [16] is specially useful in the interface description of the components of the style.

3. Architectural Elements

Shaw defines an architectural style as a family of architectures constrained by component/connector vocabulary, topology, semantic constraints and analysis methods supported [17]. Styles can be different because they use different vocabularies, topologies or constraints. The vocabulary of component/connector is frequently the principal aspect characterizing the style.

The vocabulary of the proposed style consists of components, coordination mechanisms (connectors) and sequences. Each of these architectural elements is described below.

The proposed style and its emphasis in modeling of explicit concurrency implies the identification of components not defined in some 00 architectures. These components, called processes and controllers control concurrent activities. Other components identified are typical of 00 architectures. A distinctive aspect of the proposed architecture concerns connectors. The role assigned to the coordination mechanisms used to synchronize or communicate independent sequences of activities is critical. It is due to their impact on blocking and real-time behavior. A taxonomy of coordination mechanisms was developed using the features model proposed in FODA. Fourteen coordination mechanisms were classified [6], although only six are considered candidates for the style, and are described below.

The modeling of sequences of activities with a textual notation is also critical for the analysis of real-time behavior using mathematical techniques. The notion of sequence presented here is not the same as it is in Unified Modeling Language. In UML, a sequence is the graphical representation of the interactions among objects in a single execution of a system. It emphasizes its focus on the time-ordered sequence of “transactions” [1]. In the proposed style, a sequence models a causality flow of “activities”. The reason of this difference is to allow the real-time performance analysis by the mathematical techniques of RMA.
3.1 Component

In architecture, a component defines a locus of computation. Examples are filters, databases, objects and Abstract Data Types [17].

The object orientation and explicit concurrency supported by PPOOA style enable us to define the following components as constructive elements of the style:

- **Definition Component.** It encapsulates commonly used types and constants.
- **Algorithmic Component.** This component encapsulates the algorithm that performs calculations or transforms data from one type to another. The data representation is not included in the component. An example in the domain modeled here, is the alarm monitoring component.
- **Domain Object.** It is a component implementing the behavior providing some of the services supported in the domain problem. It encapsulates operations and states for an entity of the domain; hardware and GUI dependencies are isolated. It correlates directly to a concept in the user’s domain.
- **Structure.** Also known as Abstract Data Type, is a component which encapsulate and provide a complex data type abstraction and operations to allow users to manipulate their own instances of the type. Examples are list, stack, ring, and others.
- **Process.** Component which encapsulates the control of each concurrent system activity which executes either periodically or aperiodically. Typically this component implements one thread and does not provide external operations. The aperiodic process provides an activation operation. Communication among processes is supported by coordination mechanisms.
- **Controller Object.** It is a component responsible for initiating and directly managing a cycle of action. It encapsulates the conditional branching logic or state machinery necessary to fire the events or messages that start some sequences of activities. Controller objects may be multithreaded, and thus are the most difficult for evaluation of their real-time behavior.
- **Subsystem.** A subsystem is the component which clusters other components of the architecture based on the minimizing coupling and visibility criteria. For this purpose subsystem component implements the facade pattern [9] to support synchronous communication between subsystems. Asynchronous communication between subsystems is directly supported by coordination mechanisms.

Each component of the architecture has specific attributes describing their real-time properties. The assignment of values to these attributes, allows the evaluation of the design decisions previously to testing. As mentioned before, RMA is the basis of real-time attributes definition. Characteristic attributes are:

- Execution time of each operation supported by the component.
- Blocking time of each operation supported by the component (optional, implementation dependent).
- Priority (optional, necessary for concurrent components).
- Execution period (applied to periodic processes).
- Activation pattern (applied to aperiodic processes and controllers).

The interface for synchronous communication to each component is modeled with the following notation (Figure 2).

<table>
<thead>
<tr>
<th>INTERFACE IS</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
</tr>
<tr>
<td>TYPE</td>
</tr>
<tr>
<td>SIGNATURE</td>
</tr>
</tbody>
</table>

**Figure 2. Interface Description Notation**

Asynchronous communication is supported by coordination mechanisms, described below.

Other aspects described for each component are those related to implementation. One is the necessity of an independent thread of execution for the component. Composition constraints are also critical.

3.2 Coordination Mechanisms

Synchronous communication between components is supported by their interfaces. Asynchronous communication, specially between process components is supported by coordination mechanisms.

A coordination mechanism provides the capabilities to synchronize and/or communicate components. Synchronization is the blocking of a process until some specified condition is met. Communication is the transfer of information between processes. So coordination encompasses synchronization and communication.

Experiences in the domain let us consider for the style the following coordination mechanisms:
• **Bounded Buffer.** The bounded buffer also called buffer, is a temporary data holding area. Data producers and consumers make calls to send and get fixed-length messages from the bounded buffer. Since the producer and consumer can run asynchronously, the buffer has a bounded capacity for storing data.

• **General Semaphore.** The general semaphore is a non-negative integer value used as a synchronization mechanism.

• **Mailbox.** Mailbox is a data structure used by two or more processes to pass variable length messages among them via the “send” and “receive” operations. The mailbox is considered here with null storage capacity.

• **Signal.** A process can send a signal to another process or to a group of processes as a means of coordinating those processes. Signals differ from other mechanisms in that the receiver may either wait for a signal to arrive or may receive a signal asynchronously while it is carrying out another activity.

• **Rendezvous.** The rendezvous is a synchronous and unbuffered coordination mechanism supported in Ada, that allows two processes (Ada tasks) to communicate bidirectionally.

• **Transporter.** A transporter can be considered to be an active data mover. The transporter makes calls to get and send messages from and to the coordinating processes.

The Ada 95 protected type is not a coordination mechanism at the same implementation level as these mechanisms. Protected type is the synthesis of conditional critical region and monitor concepts. Protector type simplifies the construction of higher level (e.g. bounded buffer) race-free coordination mechanisms [18].

Each coordination mechanism proposed in the style has specific values for the attributes describing the real-time properties. We used the FODA features model to represent the hierarchy of coordination mechanism properties. Thus the coordination mechanisms proposed in the style are classified according these properties.

As shown in figures, there are three main groups of features within the coordination mechanisms domain:

1. **Identification.** Those features that describe the way in which the coordinating processes that want to synchronize or communicate refer to each other (figure 3). When using direct naming, a process that wants to coordinate with another must explicitly name the recipient partner. On the contrary, when using indirect naming, the coordinating processes use some kind of intermediate object, so coordinating partners identities are hidden.

2. **Synchronization.** Those features that describe the exclusion and prioritization properties of the coordination mechanisms (figure 4). Exclusion describe the properties that guarantee the blocking of processes. Priority is concerned with the ordering of the client processes requests.

3. **Communication.** Those features that describe the information transfer capabilities of the coordination mechanism (figure 5). These capabilities include the storage capacity of the coordination mechanism for buffering either events or messages, the data flow direction and the length of the information transmitted.

![Figure 3. Identification features](image-url)
Since the services supported by each coordination mechanism are domain independent, they are completely described using a notation similar to that of the components interface. For the sake of brevity, coordination mechanism services are not described here. It can be found elsewhere [7].

Other aspects described for each coordination mechanism are those related to implementation. One is the necessity of an independent thread of execution in the implementation of the mechanism. As was mentioned previously, the use of Ada 95 protected types to implement the mechanisms precludes an independent thread of execution. A problem to consider is priority inversion. Design practices and priority assignment avoid this problem.

3.3 Sequences

A sequence is a causality flow of activities triggered by an event. The term causality flow means cause-effect chains that cut across many components of the system architecture.

A sequence is not the same as a thread of control, in the sense of either the path taken by the program counter during execution or the direction of flow of messages. For example, in communicating components through polling, causal-flow is in opposite direction of the thread of control.

In general there may be many sequences in progress through a system at any time that may interact and interfere with each other. In the proposed style, the interaction can be synchronous or asynchronous either using components or coordination mechanisms.

Sequences are triggered by events. Events can be external to the system, internal (for example an alarm detection) or time triggered. An important consideration is the distinction between internal events. An internal event is considered different from a previous one if either its arrival pattern or response time requirement are different from the starting event of the ongoing sequence. So this internal event is the starting point of a new sequence.
A single sequence represents an instance of its implicit class. The reason is that once an instance of a sequence is started, that instance continues to the end through some implied underlying cause-effect machinery, without reacting to any triggering stimuli at the sequence start point. A new triggering event at the start will trigger a new instance of the sequence.

\[ \text{Activity/Component} \quad \text{A1} \quad \text{A2} \quad \ldots \quad \text{An-1} \quad \text{An} \n \begin{array}{|c|c|c|c|c|c|} \hline C1 & X & & & & \hline c_2 & X & & & X & \hline \ldots & & X & & & \hline Cn & & & & & X \hline \end{array} \]

Table 1. Activities Allocation Table

The activities allocation table (Table 1) is a contribution of the style to represent the mapping of the sequence activities to the components and coordination mechanisms of the system architecture. Other authors prefer graphical representation of the mapping [3]. The mapping is necessary to determine which components are shared among sequences or among sequence instances. The table allows automatic verification by tools and assessment of real-time behavior using RMA.

4. Constraints on Architectural Elements

Properties are used to constrain the choice of architectural elements. That is properties are used to define constraints on the elements to the degree desired by the software architect [14].

The proposed style gives twenty five guidelines constraining the design space of the architectural style user (the software architect). These guidelines are split into the following groups:

- Guidelines concerning the Architectural Style
- Guidelines concerning the use of Components
- Guidelines concerning the Coordination Mechanisms
- Guidelines concerning Sequences
- Guidelines concerning Scheduling Policies

It is not possible to describe all the guidelines in this paper. The interested reader can find them in the thesis dissertation [7]. As an example, the guideline concerning the constraints in the operations supported by each component type is described:

Guideline #8

The constraints in the operations supported by each component should be considered.
- Definition Component. No operations supported.
- Algorithmic Component. Only supports computation operations.
- Domain Object. Supports reading (selector), writing (modifier) and computation operations.
- Structure. Supports reading (selector) and writing (modifier) operations.
- Process. Periodic Process may support deactivation operation. Aperiodic process supports an activation operation and may support other operation types.
- Controller Object. No restrictions in the operations supported.
- Subsystem. Operations supported by subsystem are reading (selector), writing (modifier) and computation.

Except for the case of the activation operation supported by the aperiodic process, and for some operations supported by the controller object, whenever a component operation is invoked, control is immediately transferred to that operation. Some of these limitations may be prevented using a coordination mechanism.

5. Architecture Characterization

Architecture characterization encompasses the form and format of the information collected about an architecture [5]. To provide some structure to this information, it is recommended to separate the architecture information into various views [13]. There seemed to be little consensus on what the views should be.

For the proposed style we proposed two views emphasizing different but complementary concerns:

- **Domain Solution View.** This view represents the constructive elements of the architecture, emphasizing composition and communication among them. Synchronous and asynchronous (using coordination mechanisms) communication is represented. Interfaces are described following the textual notation presented above. Components also include an schema of the activities implemented by them.
- **Context View.** Dynamic system properties regarding behavior are represented by sequences (description, attributes and activities allocation table). This representation allows the quantitative evaluation of real-time behavior by RMA. It will also be useful for integration and testing purposes.

The style provides a simple graphical notation (Figure 6) consistent with the vocabulary. Coordination mechanisms are explicitly represented. Composition is represented by component containment. Rules governing composition are considered in the vocabulary. Next section shows an example of Domain Solution View for the Domain Architecture and Raw_Data_Processing SS.

6. Application of the Style to a Specific Domain

To validate the proposed style, it is applied to a well known and mature domain, real-time data acquisition and monitoring of laboratory or industry processes. The capabilities supported by the systems in the domain are:

- Data acquisition. This requirement is related to the reception of data from sensors. Since these data are analog, that is electrical signals that can vary continuously over some range, I/O boards are used to sample and convert to digital value these analog electrical signals.
- Conversion to Engineering Units. Conversion is
typically required before the data are either presented to the operator, displayed as an alarm or used by other system. Conversion algorithms are stored in the system database and conversion parameters are consulted in real-time.

- Range and Alarm Checking. Range checking is performed with each data sample before converting it to engineering units. Signals out of range are not converted. Alarm checking is supported by comparing the engineering units value with low and high alarm limits. Transition from alarm condition to normal situation is also reported and historically logged.

- Historical Storage. Converted samples are stored in a database to allow future analysis by the scientist or plant engineer.

- System Operator can interact with the system to activate or deactivate sampling, change the gain or modify the range or alarm limits of an specific sensor.

Two architectures were developed implementing these capabilities. Their scalability properties were evaluated and a new architecture was modeled based on the best evaluated design decisions. The resulting architecture is split into seven subsystems: Sensors, Drivers, Raw-Data-Processing, Measures, Alarms, Process-Variables and Man-Machine-Interface (MMI). See Figure 7.

Here, it is shown the Raw-Data-Processing Subsystem using the proposed style notation (Figure 8). A more complete description of the architecture may be found in the thesis dissertation [7]. As an example we show here the sequence related to Raw-Data-Processing. The sequence triggering event: “Raw_Data_Processor activation”, is timed.

The sequence scheme is: A1 --> A2 --> A3 --> A4 --> A5 --> A6 --> A7 --> A8 --> null --> A9 --> A 10 --> A 11 | null

The associated activities are:

A1: get raw data from the raw data buffer
A2: hardware error checking
A3: consult Process-Variables SS to obtain conversion parameters and alarm limits of the sensor
A4: range checking
A5: conversion to engineering units
A6: alarm checking
A7: consult previous alarm condition

There are twelve sequences related to the dynamics of the architecture. Their complete description (purpose, attributes, activities and allocation tables) can be found in the thesis dissertation [7].

![Diagram of Data Acquisition and Monitoring Architecture](image-url)
### Tabla 2. Raw_Data_Processing Activities Allocation Table

<table>
<thead>
<tr>
<th>Activity Component</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>A5</th>
<th>A6</th>
<th>A7</th>
<th>A8</th>
<th>A9</th>
<th>A10</th>
<th>A11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw_Data_Buffer</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Raw_Data_Processor</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SS Process Variables</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DB_Server</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Range_Checking</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Units_Conversion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alarm_Monitoring</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alarms_Buffer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Screen_Messages_Buffer (SS Man Machine Interface)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Engineering_Units_Buffer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

AS: change of alarm situation is put in the alarms buffer
A9: update message is sent to the screen messages buffer of the MMI SS
A10: processed data is put in the engineering-units buffer of the MMI SS
The sequence activities allocation table is also shown (Table 2).

### 7. Conclusions and Future Directions

The proposed architectural style could be defined as “pipelines of processes using objects”. A path with more built-in control is a pipeline of processes. Each process controls a segment of the path sequentially.

The style is validated solving the problem related to real-time data acquisition and alarm monitoring although it would be applicable to other domains with similar design constrains, concurrent paths of sequential processing.

The application of the style (vocabulary, notation and guidelines) is very helpful for inexperienced software architects making decisions in the preliminary design phase. The style also supports the quantitative assessment of the design by mathematical approaches (RMA). This evaluation previous to the testing effort can save budget and resources assigned to testing.

Future directions concerning the style are:

- Find ways to integrate the proposed style smoothly into broader methods and processes of software development. Traceability with requirements and to detailed design.
- Consider the enhancement of the style to implement distributed systems (latency of the calls, coordination mechanisms, partial failure).

![Figure 8. Raw_Data_Processing_Subsystem](image-url)
• New techniques (additional to RMA) for determining and predicting properties of the style architectures.

Acknowledgments

The author likes to express his gratefulness to Rubén Prieto, Juan A. de la Puente and Francisco Gómez. He is also indebted to a number of graduate engineering students particularly Barbara Alvarez, Angel Pérez and Francisco Garcia. The reviewers comments and advice have helped to improve this paper.

References