A Process for Architecting Real-Time Systems

José L. Fernández-Sánchez[1] and Bill J. Mason[2]

[1]Industrial Engineering School
José Gutierrez Abascal 2
28006 Madrid Spain
34 91 3363146
jlfdez@ingor.etsii.upm.es

[2]Qinetiq
Winfrith Newburgh
Dorchester,DT2 8XJ, United Kingdom
44(0) 1305212960
WJMason1@QinetiQ.com

Abstract

PPOOA, Processes Pipelines in Object Oriented Architectures is an architectural style for concurrent object oriented architectures. It can be used when individual paths of execution are required to be concurrent and several processes may be positioned along the path to control the action.

To facilitate PPOOA applicability, we propose an architecting process, also named PAP (PPOOA Architecting Process). PAP is focused on the development of the software architecture for a system conforming the principles inherent to PPOOA architectural style and using the vocabulary of components and coordination mechanisms proposed by PPOOA. The scope of PAP is the architectural or preliminary software development phase of the software life cycle.

PAP should be included as a part of the general software development process. As a part of this general process, it takes some inputs from the analysis phase and produces some outputs to the detailed design phase.

The main purpose of an architecting process is to produce a rigorous description of the solution that allows quality attributes evaluation by assessment techniques, particularly Rate Monotonic Analysis (RMA).

PAP is essentially an iterative process that is split into major steps and minor steps. Architecting process major steps are those that follow the identification of the major components of the system, their interfaces and the main flows of activities that the system will implement. These major steps are namely:

1. Identify Components
2. Define Component Interfaces
3. Describe System CFAs
4. Select Coordination Mechanisms

PAP applicability is evaluated as part of the project CARTS (Computer Aided Architectural Analysis of Real-Time Systems) , an IST European funded research project (IST-1999-20608) to be finished at the end of year 2002.

Keywords: Software Architecture, Real-time systems, Software development process and UML.

1. Introduction

PPOOA, Process Pipelines in Object oriented Architectures, is a software architecture style for concurrent real-time systems. PPOOA proposes a vocabulary of components and coordination mechanisms to be used in software architecture design. UML metamodel was extended to include PPOOA building elements [5]. PPOOA is implemented in CARTS, a prototype of architecting CASE tool. PPOOA can be implemented in any CASE tool supporting UML extensibility mechanisms.

The PPOOA architecting process, named also PAP, and presented here, should be included as a part of the general software development process. As a part of this general process, it takes some inputs from the analysis phase and produces some outputs to the detailed design phase. PPOOA architecting process may be applied in either a waterfall software development process or iterative development process such as the Unified Process.

The main purpose of an architecting process is to produce a rigorous description of the solution, allowing quality attributes evaluation by quantitative and qualitative methods. Responsiveness is a critical quality attribute of real-time systems. It is related to the response time to stimuli (events) either external or internal to the system. Rate Monotonic Analysis (RMA) is an analytical method that eases the assessment of system responsiveness. It does so through a collection of quantitative techniques and algorithms that let engineers understand, analyse and predict the timing behaviour of their designs, mainly in terms of response times [10].

The main inputs to the PPOOA architecting process are the use cases and a static representation of the System types described by either one or several UML class diagrams. Although UML sequences are not essential as an input to PPOOA architecting process, they will also be very helpful.

Software architecture is described in PPOOA using two views: PPOOA Architecture Diagram and PPOOA Dynamic View. The PPOOA architecture diagram is used instead of the UML component diagram to describe the architecture. The PPOOA architecture diagram focuses on "design or conceptual components" and the usage dependencies among them. PPOOA coordination mechanisms are also represented.

The PPOOA dynamic view supports the "Causal Flow of Activities (CFA)" representation. CFA has similarities with use case map representation [1]; both represent a system internal view of the activities performed by a use case. The building elements of a CFA are: Triggering event, activiy(ies) and continuation elements. These elements are described in the PPOOA metamodel [5]. For PPOOA dynamic view, we recommend the UML activity diagram notation extended to implement CFA building elements and their real-time properties.

PAP is essentially an iterative process that is split into major steps and minor steps. Architecting process major steps are those that follow the identification of the major components of the System, their interfaces and the main flows of activities that the System will implement. These major steps are namely:

1. Identify Components
2. Define Component Interfaces
3. Describe System CFAs
4. Select Coordination Mechanisms

The process shown in Figure 1 is repeated iteratively as we split main components into subcomponents in top-down decomposition and responsibilities allocation process.

Each of the major steps shown in the figure 1 is described as a subprocess implementing minor steps. These major steps are described in the following sections.

The PAP architecting process is applied in energy, aerospace and telecom domains. Here, we present the result of its application to a Supervisory Control and Data Acquisition (SCADA) System for a laboratory monitoring.

Another example is being developed by Qinetiq to validate PPOOA, PAP and CARTS tool. The example is based on the design proposal of a MMI for a naval warning system. The MMI is required to interface to a bespoke real-time signal processing system, tactical advice system and decoy and stores management systems. The distributed nature of the system, criticality of the mission and the 'closed loop' of device control and environment display provides some interesting challenges in system architecting. At the time of write the paper Qinetiqs is elaborating the architecture diagrams using CARTS tool.

2. Industry expectations concerning an architecting process

One of the main concerns creating PAP software architecting process is its applicability in real-time systems where time responsiveness is a main issue.

The CARTS project industrial partners proposed some requirements to be met by the architecting process and supporting CASE tool.

These requirements can generally be categorised as:

- Relating to recognised design methodology, standards and tools (E.g. UML diagrams import from diverse CASE tools or access to requirements data in ODBC databases.)
- Relating to proposed system components (e.g. Hard and soft - mandating a specific processor or simply specifying a 'circular buffer')
- Providing feedback on system performance (e.g. Are the timing constraints met?)
- Realising distributed design (e.g. Integrating inter-system link timing and 'black box' response specifications.)

Many of these requirements refer to 'enabling re-use' - whether of existing, proven design, code or hardware modules. The aim of this re-use generally is to reduce cost, however in some instances the challenge may be prototype an old product with a new processor or a change of architecture. Investment in design may often be considerable, and the requirement to maintain consistency with existing product lines or compatibility with product families may have reduced emphasis when time responsiveness is a prime driver.

2.1 Components granularity

System components in any architecting process can be defined in differing granularity - some of which may be subjective and open to interpretation. Architecting processes should seek to identify duplication and duplicity in system requirements without impinging on the flexibility they offer the practitioners. Any such system should allow the practitioner the facility to develop a personalised library of validated components - based on a standard library of unambiguous building blocks.

2.2 Continuity

In describing any 'system of systems' continuity from one system to another is essential, in the 'real-world' this is the interface specification between physical systems. Similarly in the architecting process, however there is also further continuity required between system and sub-system - this logical dimension of system architecting may not physically exist - either in the hardware or software component domain. This artificial boundary may be used by the practitioner to segment the system design - either for ease of description or functional decomposition for analysis.

2.3 Testability

Having successfully fully described the system architecture the practitioner will need to validate the viability of that architecture. This may be successfully achieved by simulation or analytical techniques using reference timings obtained from historical examples, though this may need to be achieved in a rigorous way, or over a determined number of discrete simulations to achieve results of suitable confidence level in complex system architecting. The results from this validation may be cumulative and used to inform further architecture modelling or populate 'black box' system modelling within the architecting process.

2.4 Transfer of Results

Having achieved a result in an appropriate representation within any architecting system it is also a prime requirement that these should be able to be made available for use by other systems. This may mean that data requiring to be entered in a product database is populated automatically or alternatively that graphs of predicted response times could be merged within the design hierarchy of the System.

3. Architecting process detailed description

In this section the PPOOA architecting process is detailed. The four major steps of the process mentioned above and in Figure 1 are decomposed into minor steps identified with romans.

3.1 Identify Components

The main goal of this step is to create the initial set of System conceptual components. These components will be the core of the System architecture.

Components in PPOOA are considered as conceptual components but not as implementation components. A conceptual component is a concept used in diverse software architecting methods [6]. A conceptual component is a computation entity that performs an assigned responsibility, provides interfaces to other components and may require some interfaces from other components.

The minor steps of Identify Components are:

I. Identify independent types.

It is important to analyse the System types diagrams obtained in the analysis phase to discover the types that have an independent existence within the System, that is they do not have mandatory associations to other System types, except to categorising types. A categorising type is one whose instances classify the instances of other types [2].

The independent types identified here are the first-formed System conceptual components.

II. Assign responsibilities to the components identified

In the first iteration, the software architect identifies not assigned responsibilities in the use cases and assigns them to the types previously identified. A technique similar to CRC cards may be used.

In the next iterations of the architecting process it is important not to miss the assignment of CFAs activities to the identified components.

III. Select the most suitable PPOOA vocabulary element for the components identified

The basic criteria of selection of one or other PPOOA vocabulary component are the responsibilities to be supported by it, the necessity of maintaining an internal state and the necessity to support concurrent activities.

The responsibilities that a PPOOA component can implement can be related to computing, storing of data, control or signalling activities.

The component candidate for performing pure computation activities is the algorithmic component. If the component has to maintain an internal state, the candidate component is the domain component, but if typical data structures have to be stored, the candidate component is the structure. Those components that include data base access or that isolate the rest of the application from its implementation can be domain components or subsystems.

Process or controller components are required for performing control activities related, for example, to the triggering of a flow of activities. The concurrent activity control is also a responsibility of this type of components.

IV. Assign real-time attributes to components

Component real-time attributes are described in PPOOA Metamodel [5]. The assignment of values to these attributes is necessary to allow the assessment of System time responsiveness.
Two are the basic real-time attributes needed: execution time per activity and priority.

This step should be completed in the latest iteration of the architecting process.

V. Determine composition relationships between components

Component granularity and component structural properties determine the composition relationships. Component granularity depends on the architected System. It is important to consider two software-engineering principles. First, components should be as independent as possible. Second, larger components imply higher reuse levels.

Structural properties related to the type of components are defined in PPOOA vocabulary [5].

3.2 Define component interfaces

The main goal of this step is the grouping of the identified component operations in interfaces that are meaningful to the architecture solution.

The minor steps of Define Component Interfaces are:

I. Identify component operations

The operations provided by a component are related to the responsibilities or activities implemented by the component and the architectural restrictions imposed by the PPOOA architectural style

II. Group component operations in component provided interfaces

A component interface specification is more important for an architectural perspective, than the way the interface is realised or implemented. It should be possible to replace one component with another of an equivalent interface without changing the architecture.

Component instances in the physical world participate in many interactions, playing different roles in each one. The architecting process focuses on the interfaces and interactions of components. In a component-centric design, it is quite feasible to have a component offering many different services for the different problem domain roles it plays and the different technical infrastructure services required [3].

III. Determine component required interfaces

Although one of the principles in component design is to promote component independence, it is no feasible to avoid completely component dependencies to other components. The modelling of these dependencies is a critical issue of the architecting process.

In this minor step, the software architect shall identify the "required" component interfaces for each component previously identified.

3. Describe system CFAs

The main goal of this step is to describe the dynamic view of the System conceptual architecture. This behaviour modelling is essential to determine the use of coordination mechanisms and also, it is a precondition to the application of time responsiveness assessment techniques, such as RMA.

- The CFA represents a causal flow of activities, so it is not focused on modelling messaging interactions.
- The CFA may be considered a refinement of a use case scenario but focusing in an internal view of the system (glass box view).
- Due to the previous aspect, there is not a 1 to 1 mapping between use case scenarios and CFAs. The reason is that an internal event can trigger a new CFA not considered in the system black box view represented by use case scenarios.
- Several instances of the same CFA can coexist in time.
- A CFA instance can stop at certain waiting points where there is coordination with another CFA instance.

The minor steps of Describe System CFAs are:

I. Identify events and their arrival patterns

The trigger of a CFA instance is called an "event". Events are generally external to the System, but they can also be internal or timer [8].

The following event arrival patterns can be considered [8]:

- Periodic: Interval [± Δt] [phase].
- Irregular: Interval, Interval, [Interval].
- Bounded: Minimum_Interval [Maximum_Interval].
- Bursty: Number of Events, Interval.
- Unbounded: Distribution Function.

II. Identify CFAs

A CFA 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. This chain progresses through time and executes the activities provided by the system components until it gets to an ending point.

The response associated to a triggering event corresponds to the activities allocated to the triggered CFA.

Many times, parts of the response are considered overheads. Aspects such as interrupts handling, device management, context switching, remote servers and system calls, are considered as part of the response.

The smallest division of a response is an activity, or "action" in terms of RMA [8]. Due to scheduling requirements, the computation that takes place in an activity cannot cause changes in the system resources allocation. The scheduling points are each of the time instants where decisions relative to system resources allocation are made.

Activities can be arranged sequentially, in parallel or as alternatives.

III. Assign real-time attributes to resources and activities participating in a CFA.

An activity is described with a series of attributes that describe its temporal behaviour. This temporal behaviour is affected by:

- Resources used by the activity.
- Priority of the activity in the resource.
- Usage time of the resource
- Policy used for the resource

The types of resources considered in the PPOOA architectural style are: CPU, Device, Database and Coordination Mechanism.

In shared resources, the priority inversion can cause delays called blocking, and have to be considered in the analysis of CFAs.

It is obvious that an activity can use more than one resource at the same time, considering that unbounded blocking does not occur.

Each of the activities of the Architecture is described with the following attributes:

- Resources used
- Exclusivity when using resources
- Usage time of the resources
- Priority

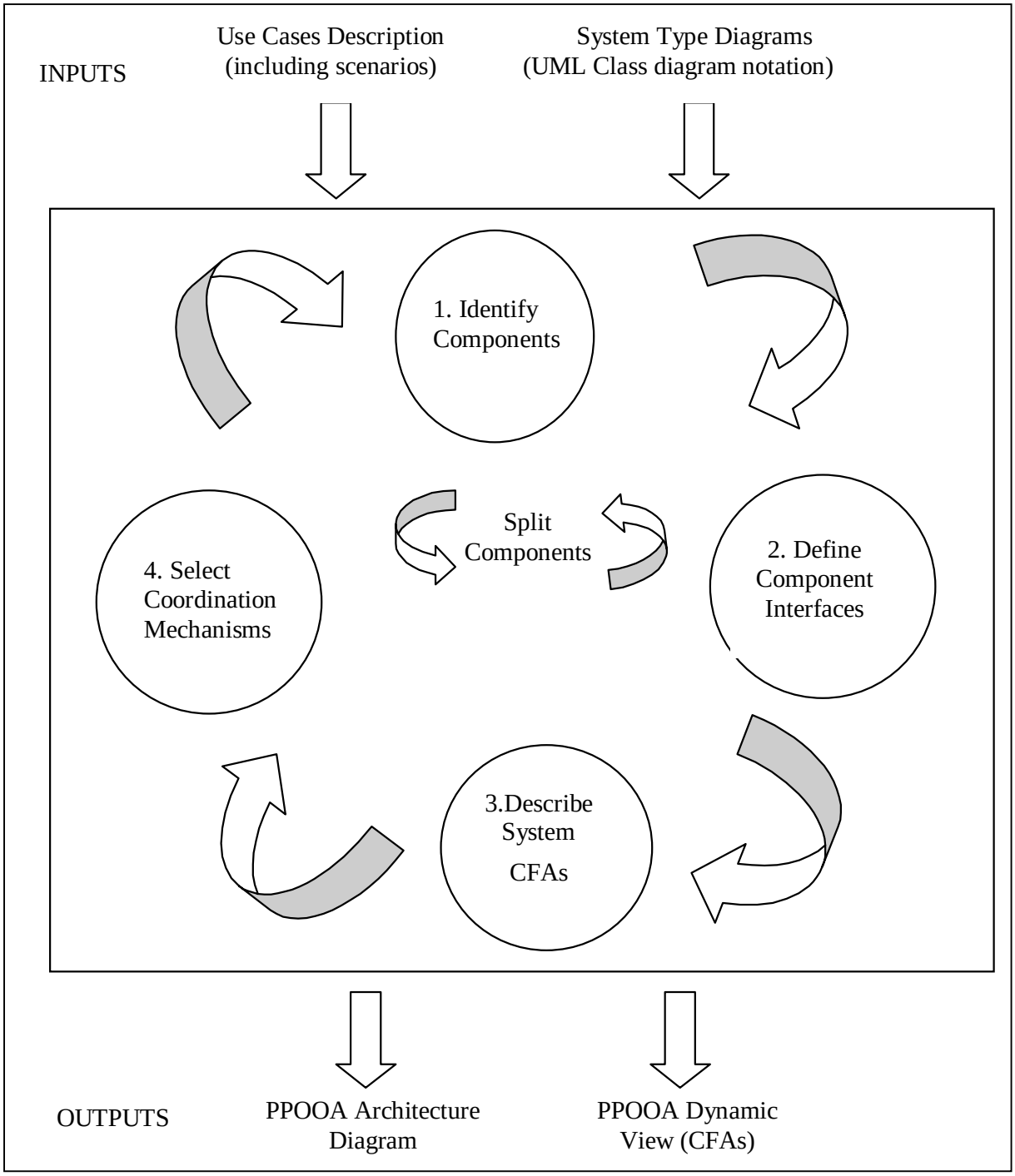

Figure 1. PAP Architecting Process

The CFA is described from the point of view of the response that it implements.

CFA textual description includes the identification of the event that causes the CFA. Temporal attributes are also described. The attribute description follows the RMA model [8]: Type of event, arrival pattern, and associated temporal requirement

IV. Establish coordinating CFAs.

The CFA instances can stop at certain waiting points where there is coordination with other instances. At these points, coordination mechanisms or objects with concurrent access are located

V. Create activities allocation table for each CFA.

The activities allocation table (Table 1) is a contribution of PPOOA style to represent the mapping of the CFA activities to the components instances and coordination mechanisms of the System architecture. It is represented in tabular form.

Activity / **Component Instance**	a_1	a_2	...	a_{n-1}	a_n
c_1			×		
c_2	×			×	
c_3		×			
...					
c_n					×

Table 1. Activities Allocation

The activities allocation table allows the estimations for RMA because the component instance that takes part in the response associated to the CFA is well known. It helps in finding which components are shared either among CFAs or among CFA instances.

VI. Build PPOO Dynamic View of the System.

PPOOA dynamic view can be represented by several UML activity diagrams.

Activity diagram models each System CFA, considering the constraints imposed to the activity bounds. An example of activity diagram for the SCADA raw data processing CFA is shown in Figure 2.

The complete Dynamic View of the SCADA example is described by more than 10 different CFAs.

3.4 Select Coordination Mechanisms

PPOOA architectural style enforces the use of coordination mechanisms as the solution to solve asynchronous interactions between the components of the architecture. The main goal of this step is to identify the more suitable coordination mechanisms for each asynchronous interaction.

The minor steps of the coordination mechanism selection are:

I. Discover the concurrency problem to solve

There are three typical problems that need to be solved in concurrent systems:

- Mutual exclusion problem
- Producer-Consumer problem
- Multiple Readers-Writers problem

The mutual exclusion problem occurs when a component needs an exclusive access to a certain resource, shared data or physical resource. Exclusion could be guaranteed if a component supporting concurrent access is selected. Another option would be to use a semaphore protecting the access to component operation.

The producer-consumer problem arises when a component needs to communicate with another component to pass messages. In PPOOA this problem is main concern and it is usually solved using the buffer coordination mechanism.

The multiple readers-writers problem is similar to the mutual exclusion problem but readers do not need to exclude one another. It is a characteristic problem of data base access.

II. Select the most suitable PPOOA coordination mechanism

The selection of the most suitable coordination mechanisms is based on two issues:

- The concurrency problem to be solved
- The needed coordination mechanism properties, that is, the synchronization and communication features that characterise the coordination mechanisms. These features are described in PPOOA Metamodel [5].

III. Assign real-time attributes to the coordination mechanism.

Coordination mechanisms real-time attributes are described in PPOOA Metamodel . The assignment of values to these attributes is necessary to allow the assessment of the System time responsiveness.

This step should be completed in the latest iteration of the architecting process.

IV. Build the PPOOA Architecture Diagram of the System.

The PPOOA architecture diagram is used instead of the UML component diagram to describe the architecture, but it maintains some similarities with the UML component diagram. The PPOOA architecture diagram focuses on design component representation and the dependence relationships between them. Composition relations between components are also represented. In the example of Figure 3, the raw-data processing subsystem of a Supervisory Control and Data Acquisition System is shown. Besides components, structural relations and dependencies, some asynchronous interactions supported by coordination mechanisms are represented. The PPOOA coordination mechanism used in the SCADA are the buffer and the transporter.

Several architecture diagrams may be used, one to represent main subsystems and their dependence relationships. Others for each subsystem and their components.

3.5 Split Component

PPOOA architecting process is presented as a top-down process where high level components are decomposed into more primitive ones. But in practice the approach followed is a mixed approach where top-down and bottom–up approaches are concurrently used.

It is recommended that the architect identifies subsystem components and then allocates to the subsystems, minor components already identified.

In some cases, due to several reasons (concurrency, persistence or others), a larger component is factored into smaller ones.

4. Conclusion

An architecting process for real-time component-based architectures is proposed. This process is part of an architecture style called PPOOA[4].

To implement PPOOA, UML metamodel was extended to include PPOOA building elements and being respectfully with UML semantics.

To facilitate the use of PPOOA style an architecting process called PAP is defined. The principles inherent to this process are abstraction refinement and usefulness. The process has to main goals: architecture components identification and behaviour description. The outputs of PAP process are an architectural view and a dynamic view; one to many CFAs represents the last. The notation used for dynamic or behavioural description allows seamless application of time responsiveness assessment techniques such as Rate Monotonic Analysis (RMA). This description is not easy found in other architectural representations [9]

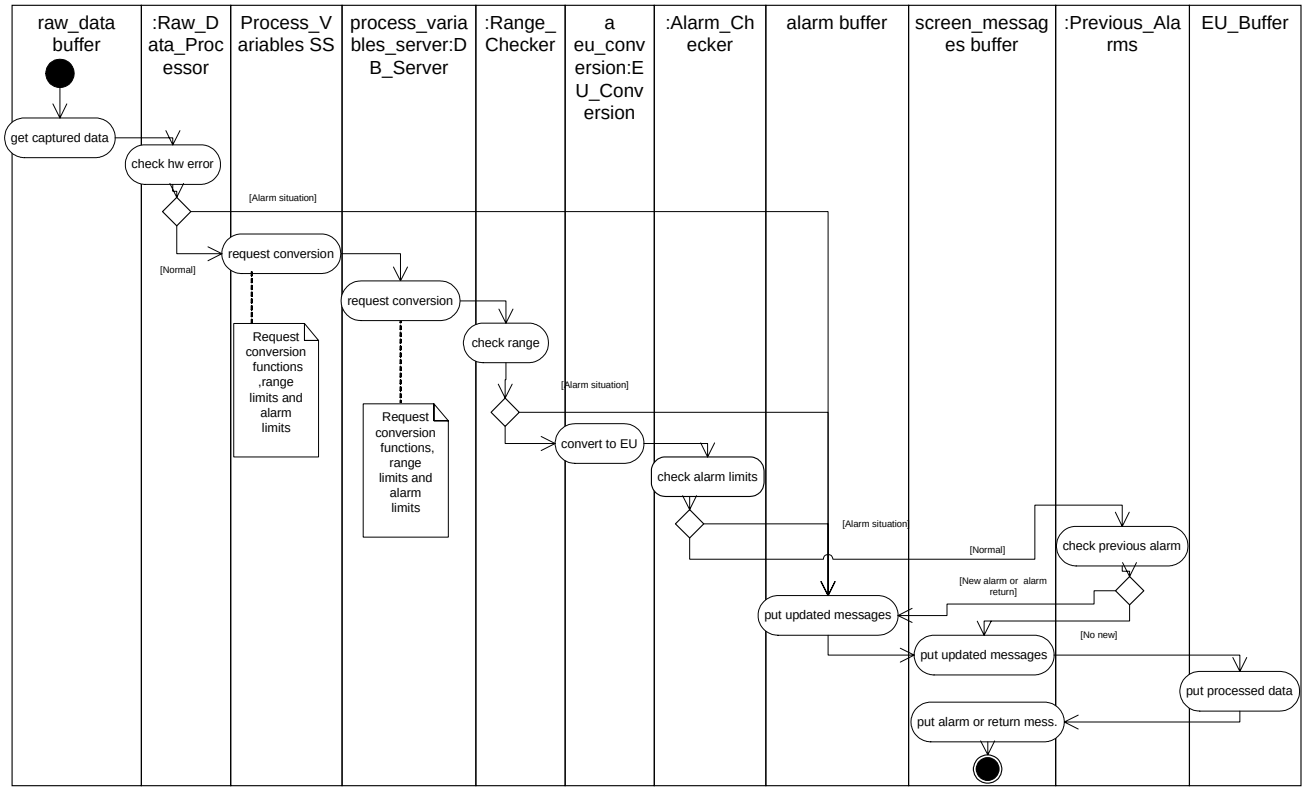

Figure 2. Raw Data Processing CFA

PAP is used as part of CARTS project, a European Union IST -1999-20608 research project. The goal of the project is to develop a CASE tool to implement PPOOA. Several application domains were chosen to validate PPOOA, the CASE tool and the PAP architecting process. The application domains chosen are telecommunications and aerospace. The PPOOA architectural style was previously used in supervisory control and data acquisition systems for laboratory monitoring.

Any CASE tool supporting UML extension mechanisms may be adapted to support PPOOA architectural style.

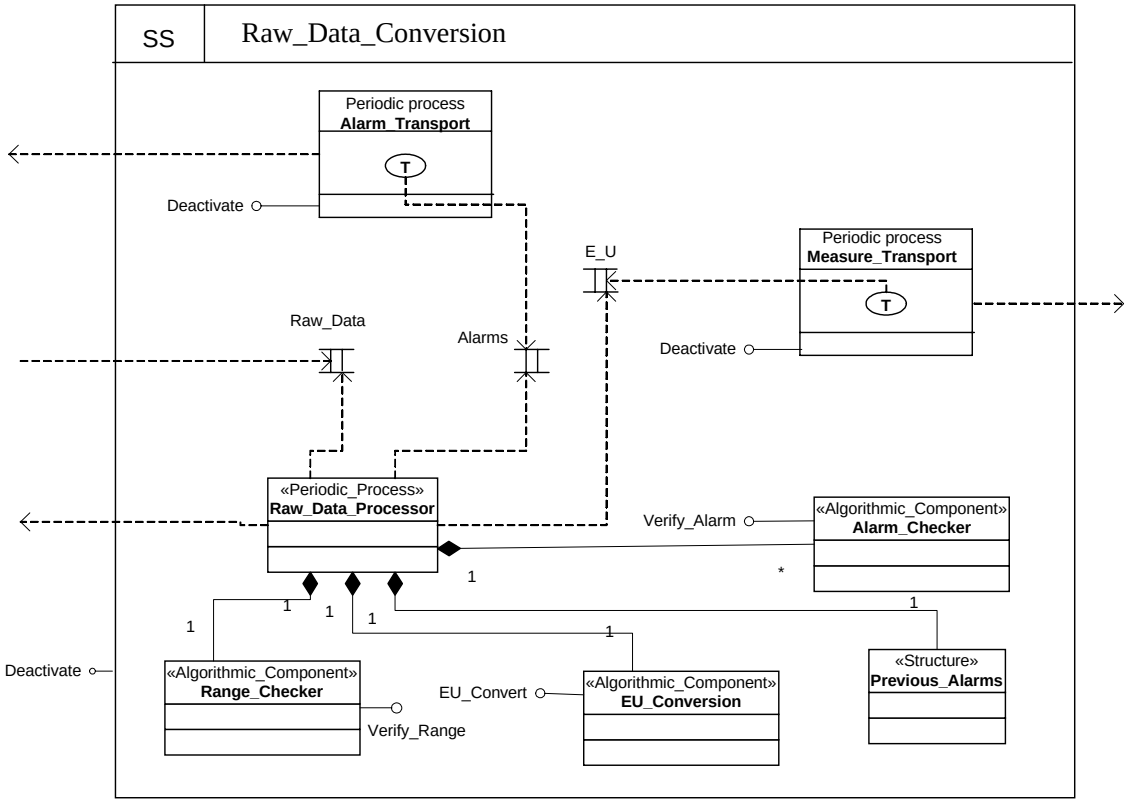

Figure 3. Raw data processing subsystem

5. References

[1] Buhr R.J.A. and Casselman R.S. Use Case Maps for Object-Oriented Systems.Prentice Hall, Upper Saddle River NJ, 1996.

[2] Cheesman, J. and Daniels, J. UML Components. A Simple Process for Specifying Component-Based Software.Pearson Education, Upper Saddle River NJ, 2001.

[3] D´Souza D.F. and Wills A.C. Objects, Components and Frameworks with UML. The Catalysis Approach. Addison Wesley, Reading MA, 1998.

[4] Fernandez J.L. An Architectural Style for Object Oriented Real-Time Systems. Proceedings of the Fifth International Conference on Software Reuse Victoria (Canada), 1998, IEEE.

[5] Fernandez-Sanchez, J.L and Monzon A. Extending UML for Real-Time Component -Based Architectures.14th International Conference on Software and Systems Engineering and their Applications. Paris. December 2001.

[6] Hofmeister, C., Nord, R. and Soni, D. Applied Software Architecture.Addison-Wesley Longman, Reading MA, 2000.

[7] Jacobson, I.,Booch, G. and Rumbaugh, J. The Unified Software Development Process. Addison-Wesley Longman, Reading MA, 1999.

[8] Klein, M.H., Ralya,T., Pollak, B. Obenza, R. and Gonzalez Harbour, M. A Practitioner´s Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1993.

[9] Kruchten, P. The 4 +1 View Model of Architecture. IEEE Software. November 1995.

[10] Oshana, R. Rate Monotonic Analysis Keeps Real-Time Systems on Schedule. EDN. September 1, 1997.

Acknowledgements

Thanks to CARTS project partners and to CARTS project officer and CARTS reviewers for their enthusiasm and support throughout the project.