The Design of the DAQ-Unit in the ATLAS DAQ/EF -1 Project

Authors : G. Ambrosini, E. Arik, H.P. Beck, S. Cetin, T. Conka, A. Fernabdes, D. Francis, Y. Hasagawa, M. Joos, G. Lehmann, J. Lopez, A. Mailov, L. Mapelli, G. Mornacchi, Y. Nagasaka, M. Niculescu, K. Nurden, J. Petersen, D. Prigent, J. Rochez, L. Tremblet, G. Unel, S. Veneziano, Y. Yasu.

Keywords :

"Supreme excellence consists in breaking the enemy's resistance without fighting."

Sun Tzu.

NoteNumber :
Version : 0.4.d
Date : 00-00-00
Reference :

1 Introduction

1.1 Purpose of the document

This note is an attempt to present the design of the DAQ-Unit subsystem of the DataFlow in DAQ/EF -1 system. The design is presented using the Unified Modelling Language (UML) V1.3.

The design presented here should be viewed as a snapshot of an evolving process.

1.2 Overview of the document

Section 2 presents the DAQ/EF -1 context and the DataFlow context and its major subsystems. In section 3, at the level of subsystems introduced in section 2, the interaction of the DataFlow subsystem with other subsystems is presented. The design of the DAQ-Unit subsystems is presented in section 4. Detailed interaction diagrams, at the level of the man classes within the DataFlow, are presented in section 5. Deployment diagrams are presented in section 6.

2 DAQ/EF -1 context

2.1 System context

The DAQ/EF -1 projects aim was to prototype the design and implementation of a fully functional vertical slice of the ATLAS DAQ and Event Filter as outlined in the ATLAS technical proposal. The project paid particular attention to the functional requirements.

The DAQ/EF -1 and its main subsystems are shown in the context diagram of Figure 1.

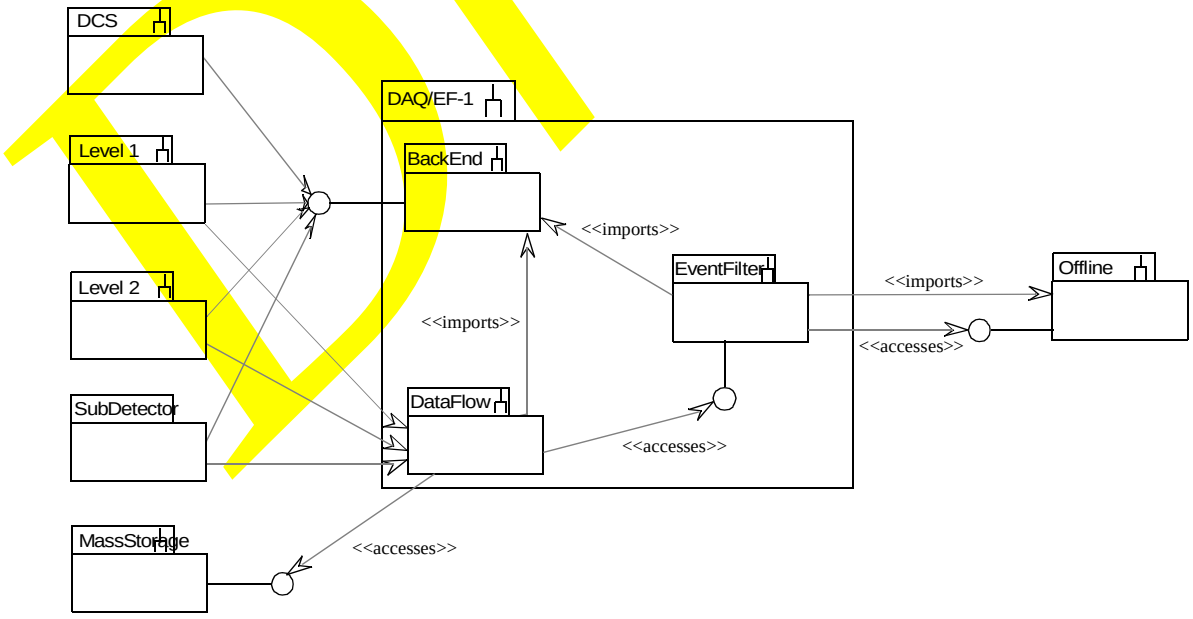

Figure 1: DAQ/EF -1 main subsystems and context diagram.

It can be seen that, based on functional requirements, the DAQ/EF prototype -1 has been organised into three subsystems:

- *Back-End*: provides all the functionality of a DAQ system not related to the movement of event data, e.g. configuration, control and monitoring.
- *Event Filter*: provides for the final selection of events. It also provides the monitoring and calibration of the ATLAS detector using complete events. It receives/returns events from/to the DataFlow.
- *DataFlow*: is the hardware and software elements responsible for: receiving, buffering and distributing event data; providing event data for monitoring; storing event data from the detector. It also provides and receives event data from the EventFilter subsystem using an interface implemented by the EventFilter.

Figure 1 also shows the dependencies between the DAQ/EF -1 and other Trigger/DAQ subsystems and Atlas systems. The Level 1, Level 2, DCS and SubDectector subsystems have a dependency on the Back End. The Event Filter, due to the requirement to use offline algorithms, has an import relationship with the Off-line system. In addition, the Event Filter accesses, via an interface, the Off-line package *e.g.* “offline databases”. The DataFlow subsystem accesses an interface, implemented by the Mass Storage subsystem, for event storage. It is also shown in this figure that the Level 1, Level 2 and Sub-Detector subsystems have a dependency on the DataFlow subsystem. This is not a priori a software dependency but indicates a physical dependency between these subsystems *i.e.* a physical link.

2.2 DataFlow subsystem context

The DataFlow, see Figure 2, is composed of three subsystems: the *ROC* (Read-Out Crate), which provides the receiving, buffering and forwarding of data fragments from the detector; the *EventBuilder*, which provides the merging of event fragments into full events; the *SFC* (SubFarmCrate), which provides the sending to and retrieving of events from the Event Filter and for the sending of events to mass storage.

The ROC contains two packages: the *LDAQ* and the *DAQUnit*. The latter is the subsystem which provides the receiving, buffering and forwarding of data fragments from the detector. The LDAQ provides the control and monitoring within a ROC and implements an interface which is used by the BackEnd subsystem to control and monitor a ROC. The LDAQ is a global package within the DataFlow subsystem, it is imported by the EventBuilder and SFC subsystems

The flow of event data through the DataFlow subsystem is supported by the DAQUnit, EventBuilder and the SFC.

.

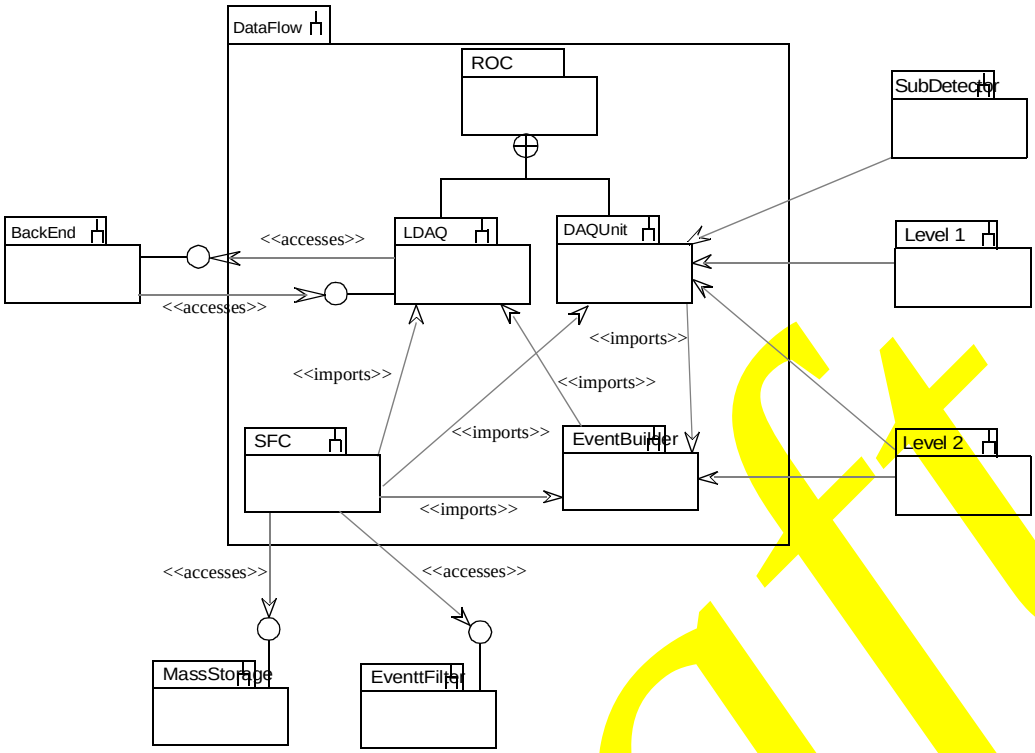

Figure 2: DataFlow context and packages.

Figure 2 also shows the relationship between DataFlow subsystems and other subsystems within and external to DAQ/EF -1:

- The SFC uses an interface for the sending and receiving of events from the EventFilter, the latter implements the interface. Similarly, the SFC uses an interface implemented by the MassStorage subsystem for the storing of events.
- The DAQUnit depends on: the SubDetector for the input of event fragments; the Level 1 for the acceptance of calibration events; the Level 2 for the level 2 accept, reject and ROI request.
- The EventBuilder subsystem depends on the Level 2 subsystem for the level 2 accept.
- As mentioned in the previous section, the dependency of the Level 1, Level 2 and SubDetecor subsystems on the DAQUnit and EventBuilder indicate physical dependencies not software dependencies.

3 DAQ/EF -1 Interaction diagrams

3.1 General

This section presents the interaction diagrams for four specific patterns: Level 2 reject, Level accept, ROI request and calibration event[1]. The interaction patterns are shown at the level of

1. The term calibration event is used to refer to any data taking activity which does not require the use of the Level 2 subsystem.

subsystems described in Figure 2. More detailed interaction patterns, i.e. at the level of the main classes in each DataFlow subsystem, are presented in section 5.

3.2 Level 2 Reject interaction pattern

The Level 2 Reject interaction pattern is shown in the form of a collaboration diagram in Figure 3. The pattern is a single asynchronous communication between the Level 2 system and the DataFlow system. One or more level 1 IDs are communicated to the ROC via this communication.

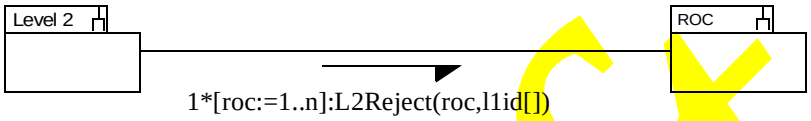

Figure 3:

3.3 Level 2 Accept interaction pattern

The Level 2 Accept interaction pattern is shown in the form of a collaboration diagram in Figure 4. The pattern starts with the asynchronous communication of the L2Accept to all ROCs.

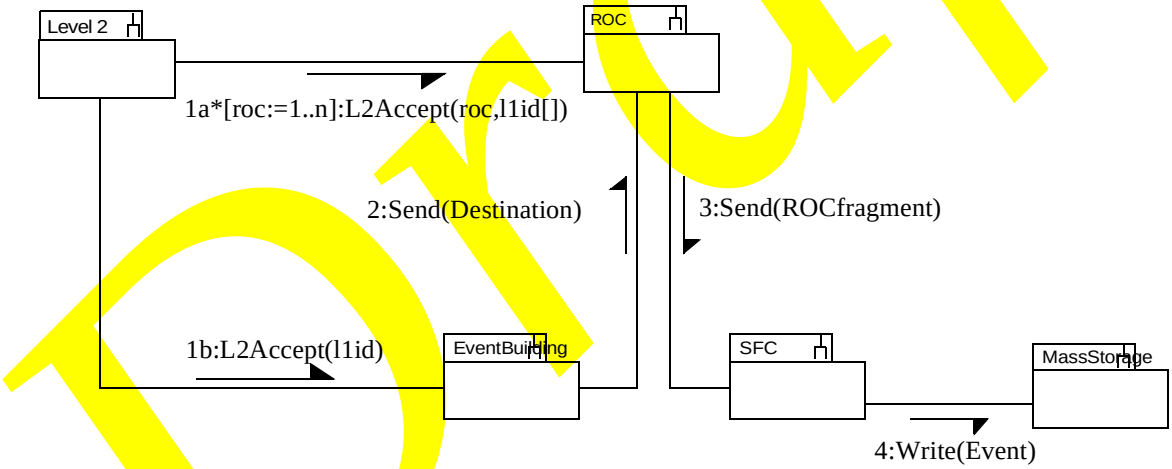

Figure 4:

In this communication, the level 1 ID of events accepted by the Level 2 subsystem is communicated to the each ROC. Concurrent to this communication, sequence number 1b, the same communication is made by the Level 2 subsystem with the EventBuilding subsystem. On reception of these messages the ROC builds the ROC fragment object and the EventBuilder assigns a Destination to the event. This Destination is subsequently, communicated to the ROC, sequence number 2, and the ROC then communicates the ROC fragment to the SFC subsystem. On reception of all ROC fragments from all ROCs, and the building of the ROC frgaments into an event and any event processing, the event is sent to MassStorage.

3.4 ROI Request interaction pattern

The ROI Request interaction pattern is shown in the form of a collaboration diagram in Figure 5.

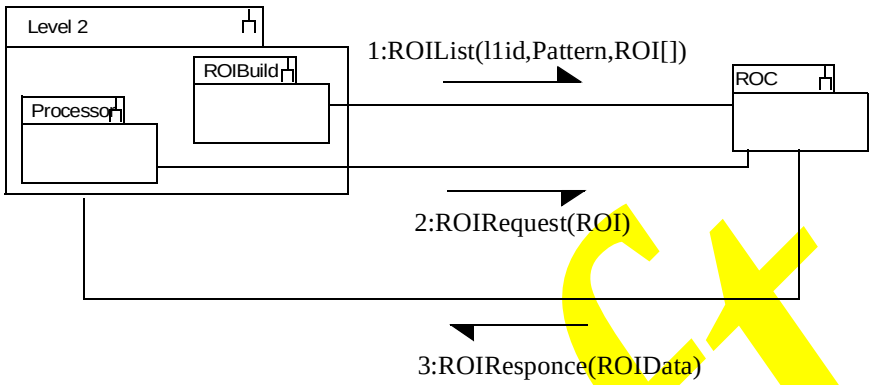

Figure 5: ROI Request interaction pattern.

The pattern is initiated by an asynchronous communication between the Level 2 subsystem and the ROC. The communication consists of the exchange of a list of *ROIs*. The Level 2 subsystem, some time later, requests ROI data from the ROC with an asynchronous communication, sequence number 2. The ROC replies, with an asynchronous communication, with *ROIData* to the Level 2 subsystem.

3.5 Non-physics event interaction pattern

The interaction pattern for non-physics events, *e.g.* calibration events, is shown in the form of a collaboration diagram in Figure 6. The pattern is very similar to that shown in Figure 4, the main differences being that the Level 1 subsystem replaces the Level 2 subsystem.

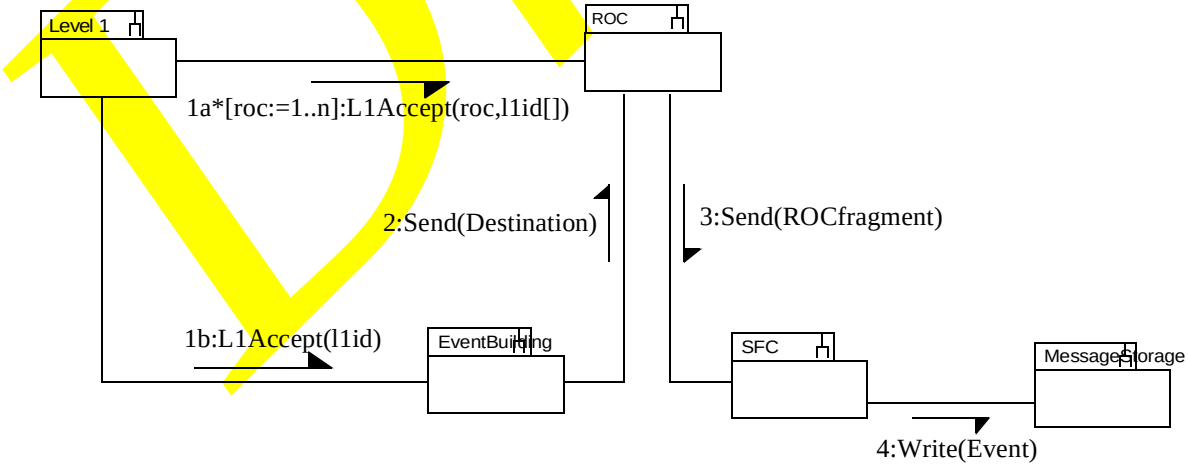

Figure 6: Non-physics event interaction pattern.

4 DAQ-Unit

4.1 Introduction

The DAQUnit is the subsystem of the DataFlow which provides the receiving, buffering and forwarding of data fragments from the SubDetector subsystems. This section introduces the main components of the DAQUnit system.

A DAQ-Unit is associated to one or more external I/O channels, e.g. detector ROLs, and all the functionality required to handle an external I/O channel is called a Task. The latter may be placed on one or more processors and communicate amongst themselves, via internal I/O channels, to ensure the DAQ-Unit functionality.

Tasks are combined to form I/O Modules (IOMs) and the latter provide the framework to schedule, configure and control their Tasks. Examples of instances of IOMs are the Read-Out Buffer (ROB), Event Builder Interface (EBIF) and the Trigger interface (TRG). Each of these instances are associated to a single external I/O channel and one or more internal I/O channels. Other examples of IOMs combine the Tasks designed and implemented foe the afore mentioned IOMs in different ways, an example being a single IOM having the full ROC DAQ-Unit functionality.

The DAQUnit receives input, asynchronously, from three external I/O channels: one or more ROLs, LVL2 and EB. For each input an action is performed:

- For each ROL, per event, a single ROD fragment must be received and buffered. In addition fragment is formatted into a ROB fragment.
- From the LVL2 it must receive: one or more requests for ROI data, a single LVL2 Reject message per event, a single LVL2 Accept message per event. The reception of a L2R message leads to the removel of event fragments from the buffer. The LVL2 Accept messages triggers the preparation of ROB fragments for transmission to the EventBuilder.
- From the EB per event, the destination of the event fragments is received.

4.2 IOM class diagram

The major component of the DAQUnit is the I/O Module (IOM). It provides the means to input, buffer and output data and or data control messages. An IOM is also located at the boundary between the DAQUnit and other functional elements. Therefore, it also implements the interface between the DAQUnit and other subsystems *(e.g.* the Trigger system).

IOMs are data driven: they receive and process data and messages, which control the flow of data, from one external I/O channel and one or more internal[1] I/O channels. Data is buffered and the data control messages indicate the actions to be performed on the buffered data e.g. delete, forward and copy. All functionality associated to an I/O channel is called a *Task*. The framework of the Tasks and the framework supporting the Tasks is common to all instances of an IOM. This common framework and core functionality, *e.g.* message passing between IOMs,

1. Internal to the DAQUnit.

is provided by a package called the GIOM (Generic IOM). The IOM class diagram is shown in Figure 7.

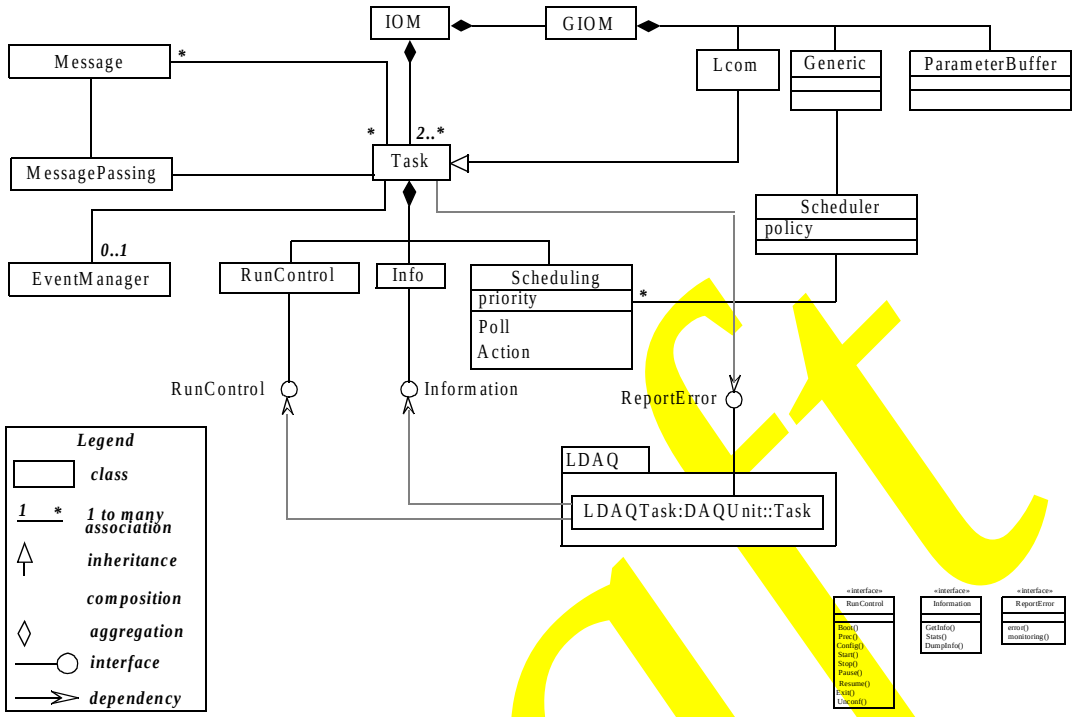

Figure 7: The IOM class diagram.

It can be seen that an IOM is a composition of three classes: *GIOM*, *Instance* and two or more *Tasks*. The latter is an aggregation of three, abstract, classes: *RunControl*, *Info* and *Scheduling*.

The classes RunControl and Info implement the interfaces *RunControl* and *Information.* These interfaces are used by the package LDAQ for the control and monitoring of Tasks and hence instances of IOMs.

Tasks are scheduled by the *Scheduler* using a *policy* and the methods of the class *Scheduler*. They communicate between themselves via the MessagePassing class.

4.3 TRG object diagram

The TRG receives and buffers data control messages from the trigger systems or alternatively, is itself the source of these messages[1]. According to their type, the messages are sent to the ROBs or EBIF. The TRG also provides a mechanism whereby messages may be stacked according to their type before sending.

The TRG is structured around three tasks:

1. *Task1*: This is the task which inputs and buffers the data control messages from the Level 1 or Level 2 trigger systems. The boundary with the trigger system is an interface which is managed by this task. The polling condition for this task indicates that one or more data

1. Specifically when there is no external trigger system available *i.e.* during test procedures.

messages have been or must be transferred to a buffer from the interface. In the latter case, this task may have to control the transfer of the data control message. Which of these conditions is implemented depends upon the design of the interface.

2. *Task2*: This task processes the data control messages placed in a buffer by the input task. The data control messages are identified as those which must be sent to the ROBs or those which must be sent to the EBIF. They may be temporarily stacked according to their type or sent directly to the ROBs or EBIF. When stacking is used, the whole stack is sent as a single message on a stack full condition. The maintenance of the data control message buffer and stack is performed by this task.
3. *Communications with the LDAQ*: This is the task which integrates the TRG with the LDAQ and is discussed elsewhere. At the level of the IOM, functions are required to execute run control and monitoring request commands.

The TRG object diagram is shown in Figure 8.

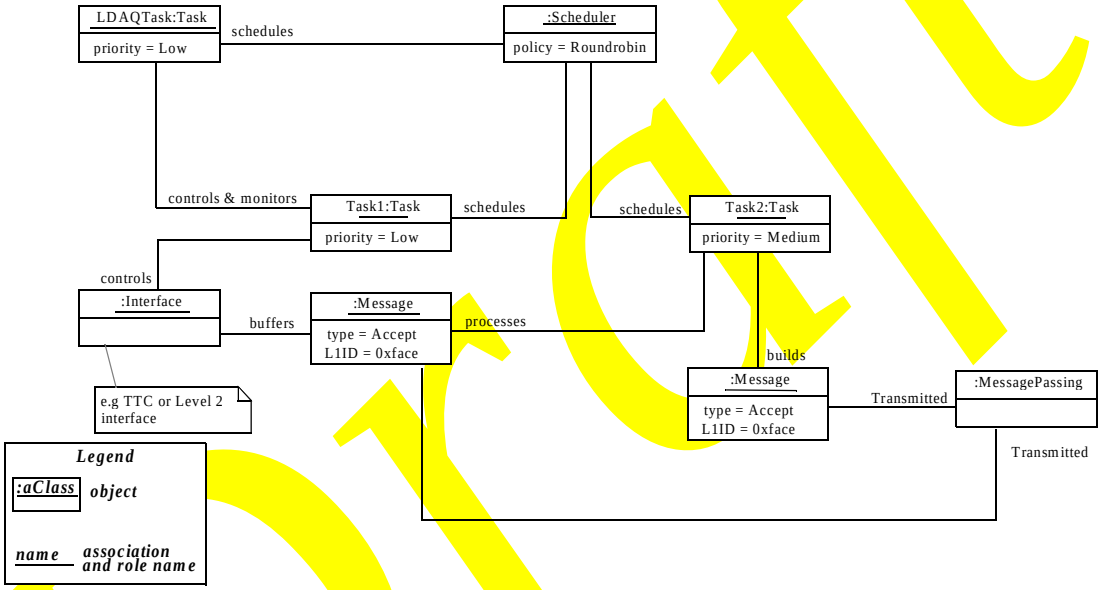

Figure 8: TRG object diagram.

4.4 EBIF object diagram

The EBIF, via the data collection component, builds the crate fragment into an internal buffer. These fragments are subsequently sent to the event building sub-system. The building of a crate fragment is started via the reception of a data control message, e.g. Level 2 accept, from the TRG. Alternatively, the TRG input task may be implemented as a component of the EBIF. In this case data collection occurs as a consequence of a message received directly from the Level 1 or 2 trigger systems.

The EBIF is structured around four tasks:

1. *Task1*: This is the task which receives data control messages from the TRG or from the Level 1 or 2 trigger systems. In the former case the messages are received using the message passing functionality of the generic IOM. A message indicates that an event has been accepted by the Level 2 trigger system (or the Level 1 system in the absence of a Level 2

system) and the identifier (common to the trigger systems and the DAQ) of the accepted event is conveyed within the message. The event identifier is decoded from the message and stored in a data structure shared with the Data collection task. This data structure is maintained by this task and must therefore be accessible via read or write operations.

2. *Task2*: This is the task which performs the collection of ROB fragments from all ROBs in a ROC, for a specific event identifier, to form a crate fragment. The task appends that information required by subsequent elements to directly access a specific ROB fragment. The event identifier of the ROB fragments to be collected is accessed from a read only data structure shared with the Input task. The task also adds EBIF specific data to the crate fragment.

   Data collection may proceed based on a shared memory model between the EBIF and the ROBs or via message passing between the EBIF and the ROBs. The event identifier of the crate fragments are stored in a data structure shared between this task and the source task. The data structure is maintained by this task and must therefore be accessible via read or write operations.

3. *Task3*: This task is responsible for the output from the ROC of crate fragments. Alternatively, in the absence of a EBIF this task becomes a ROB task and is responsible for the output of ROB fragments. The output could be to the event building sub-system or to a local data storage system. This element is discussed elsewhere in the context of output to the event building sub-system.

4. *Communications with the LDAQ*: This is the task which integrates the EBIF with the LDAQ and is discussed elsewhere. At the level of the IOM functions are required to execute run control and monitoring request commands.

The EBIF object diagram is shown in Figure 9.

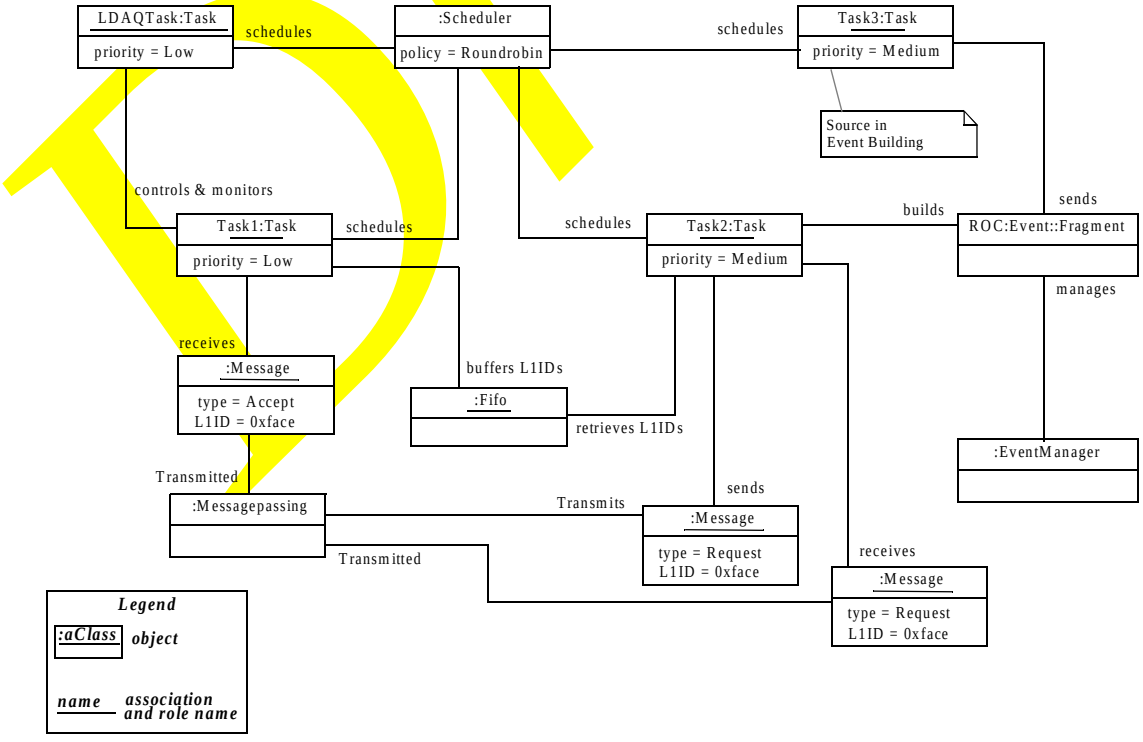

Figure 9: EBIF object diagram.

4.5 ROB object diagram

4.5.1 The Tasks of the ROB

The ROB receives and buffers a ROD fragment from a Read-Out Link (ROL) per event. A ROB may have one or more ROLs. In addition, these fragments must be: copied to an external Trigger system; accessed via the EBIF for Data Collection; and removed from the buffering system. The ROB must also format the received event fragments to form a ROB fragment. The ROB has three main logical Tasks:

1. *Input*: This task receives and buffers the ROD fragments coming from the ROL. In the absence of a ROL *i.e.* for development and testing purposes, it is also the source of ROD fragments. The polling condition for this task indicates that a ROD fragment must be transferred from the ROL to the buffer or that a fragment has been asynchronously transferred from the ROL to the buffer. In the case that the fragment is pending on the ROL, this task may have to set-up and perform the transfer and handle the ROL protocol.
2. *Communications with the TRG*: This task receives and processes data control messages from the TRG. Two types of data control messages are received from the TRG: a ROI type indicating that ROB fragments must be forwarded to the Level 2 Trigger system and a L2R type indicating that ROB fragments must be removed from the buffer. This task polls on the arrival of data control messages from the TRG.
3. *Communications with the EBIF*: This task collaborates with the EBIF to facilitate the Data Collection process. It receives a request data control message from the EBIF and sends a response data control message to the EBIF. The request message contains the Level 1 Identifier (L1ID) of the event fragment to be collected and the L1ID of a previously collected event. The response data control message contains the location (or locations) within the ROB buffers of the data associated to the requested L1ID. The event data associated to the L1ID of a previously requested event is removed from the internal buffers of the ROB. On receiving the response data control message the EBIF transfers the data from the ROBs to its local buffer. The polling condition for this task indicates the arrival of data control messages from the EBIF.

The Tasks of the ROB access the buffer for the storing, retrieving and deleting of events. The buffer management and the access mechanisms to the buffer contents are provided by the Event Manager component. The interface implemented by this component supports one or more buffers.

The ROB object diagram is shown in Figure 10.

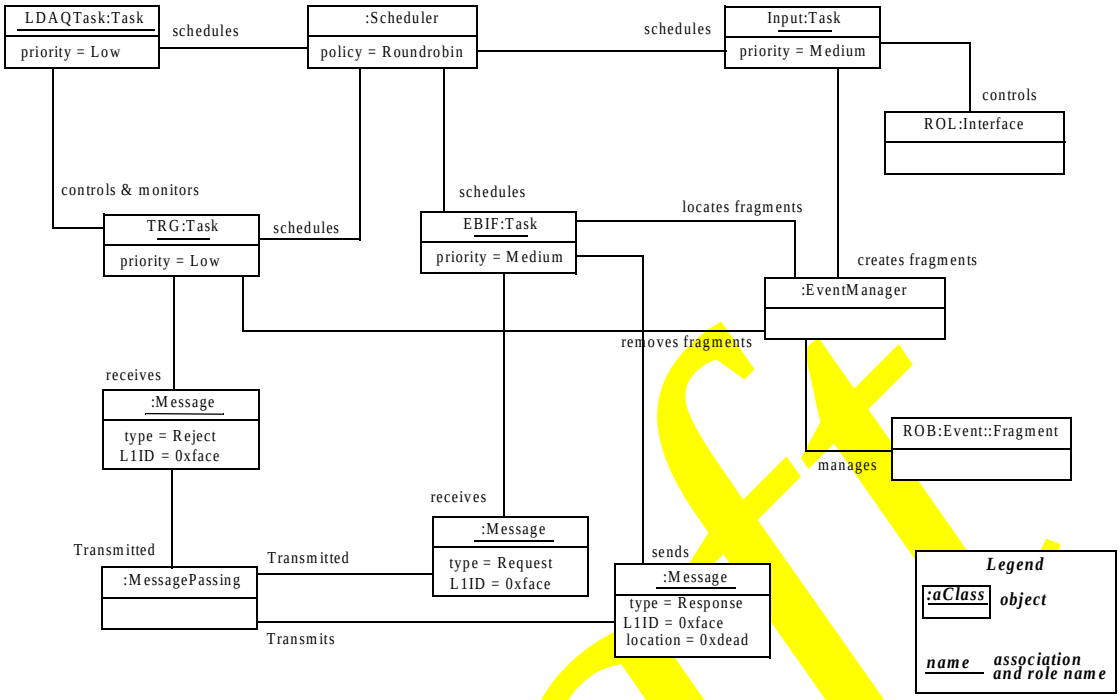

Figure 10: ROB object diagram.

5 Interaction patterns

5.1 General

This section presents the main interaction patterns: the Level 2 reject, Level 2 accept and ROI request. It also presents generalised patterns, those re-occurring in each of the four main patterns.

5.2 Scheduling pattern

The scheduling of Tasks is a recurrent theme through-out the main interactions. Its is presented here to avoid reproduction in the presentation of the main interactions.

Each Task known to the scheduler implements the scheduling interface. The principle methods of this interface are Poll and Action. The scheduler uses the Poll method to determine whether

or not the associated Action method needs to be invoked, i.e. whether the Task needs to be scheduled.

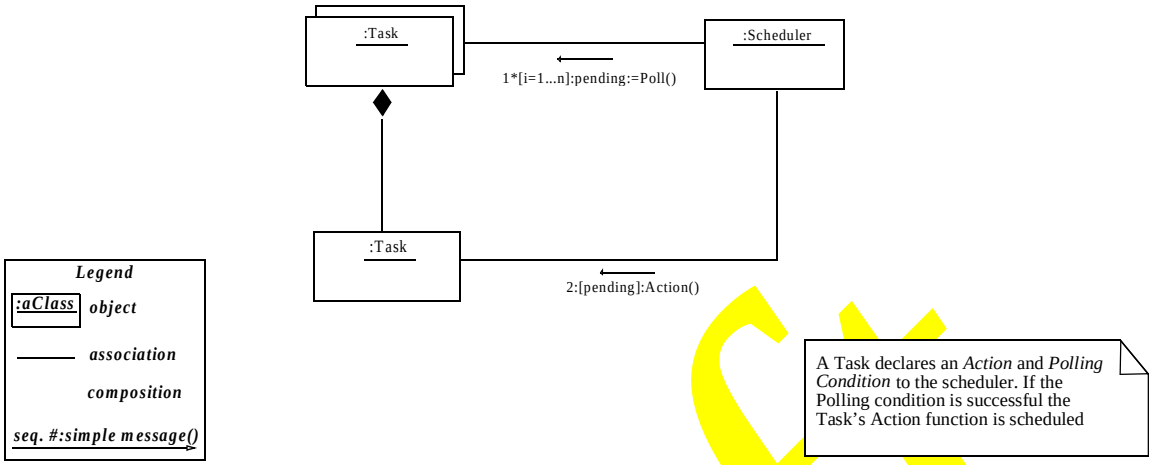

Figure 11: Scheduling pattern.

5.3 Level 2 decision interaction pattern

This section presents the Level 2 decision interaction pattern, Figure 12. The interaction involves the Level 2 system and the object TRG, an instance of the class IOM.

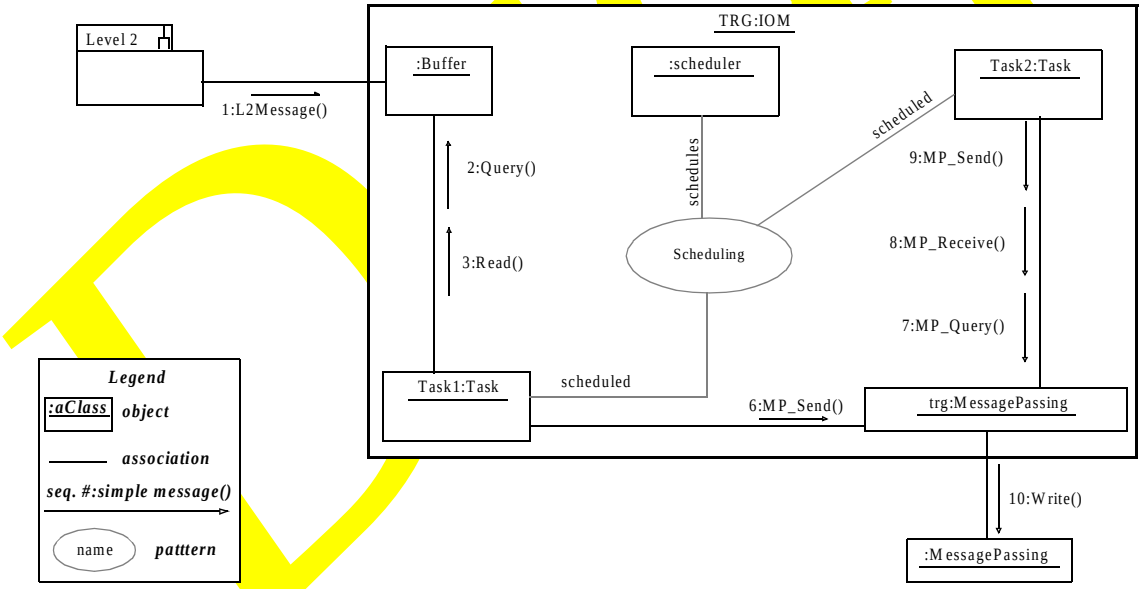

Figure 12: Level 2 decision interaction pattern.

5.4 Level 2 Reject interaction pattern

Figure 13 shows the collaboration diagram for a Level 2 reject interaction pattern. The interaction involves the composite object ROB and the pattern Level 2 decision interaction pattern. The ROB object also uses the scheduling pattern.

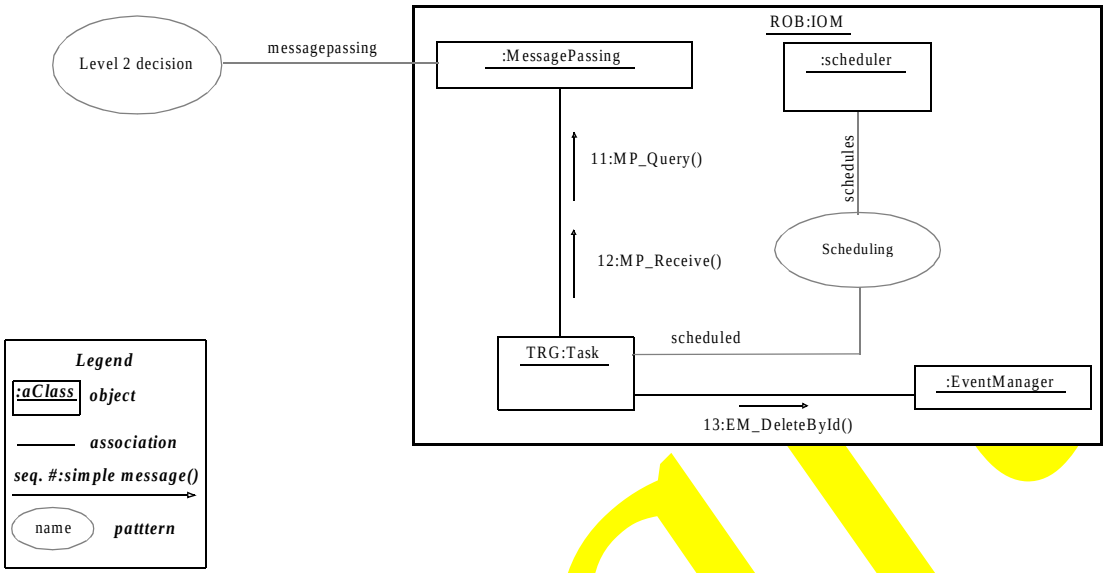

Figure 13: Level 2 Reject interaction pattern.

The interaction pattern starts with an asynchronous communication involving the Level2 system and an instance of the class buffer. The result being an ordered list of L2Decision objects are stored in object buffer. Object Task1 checks that one or more new L2Decisions have been received by the object buffer and subsequently processes them. L2Decisions of type L2Reject are exchanged with the object Task2 via the object MessagePassing. The object Task2 uses the methods MP_Query() and MP_receive() of MessagePassing to query and receive the L2Decisions. Task2 extracts the L1ID from each of the decisions and maintains them in a list. When the list contains *group* L1IDs, Task2 uses the method MP_Send(), sequence number 9, to send the group of L2Rejects to the object ROB.

The instance TRG of the class Task queries and receives, via the methods MP_Query() and MP_Receive() of an instance of the object MessagePassing, the list of L1IDs “sent” by the object TRG. The object TRG subsequently, via the method EM_DeletebyId(), removes the events from the object EventManager.

5.5 Level 2 Accept interaction pattern

Figure 14 shows the collaboration diagram for a Level 2 accept interaction pattern. The collaboration involves two instances of the class IOM, EBIF and ROB, each of which is shown as a

composite object. Note: each has its own thread of execution. The ROB and EBIF objects also uses the scheduling pattern and the Level 2 decision pattern is also present.

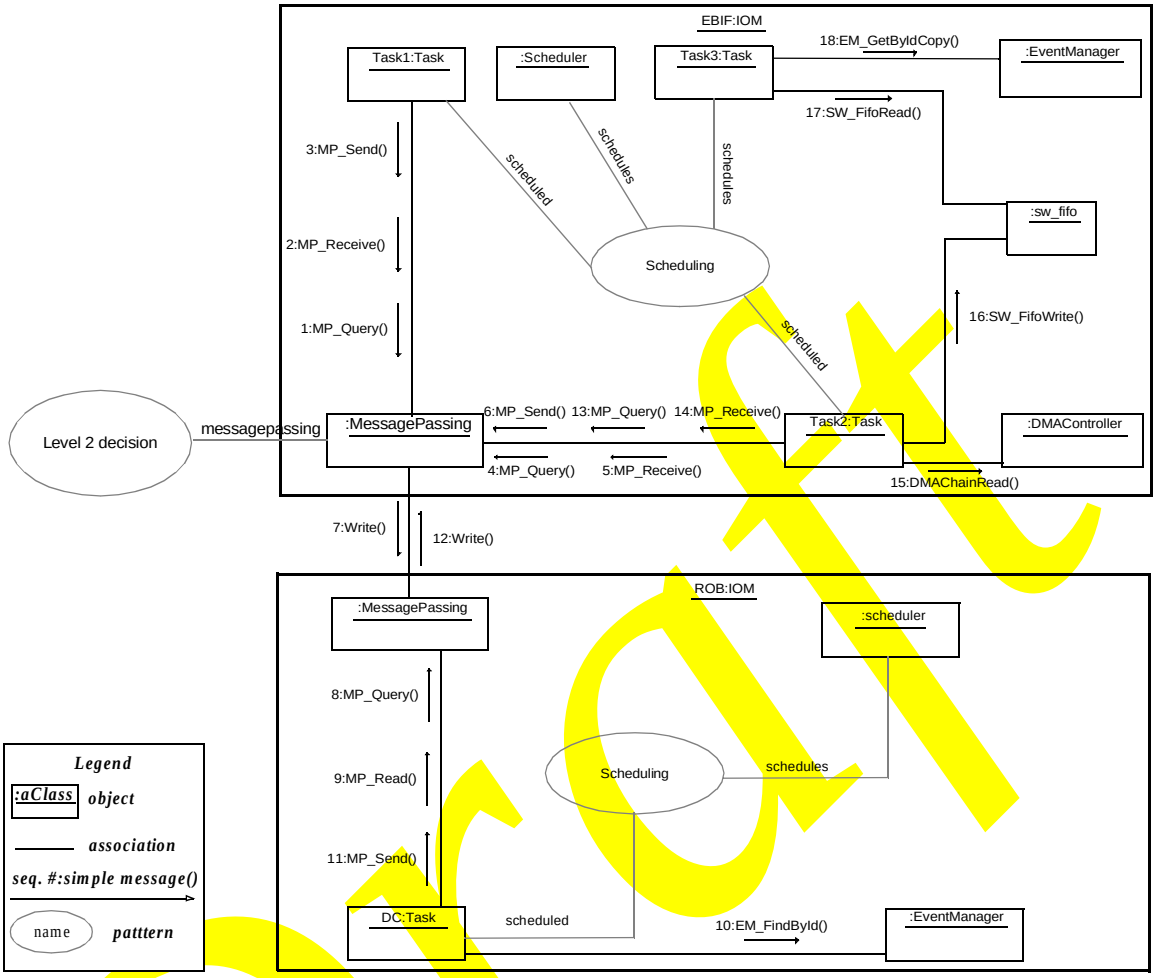

Figure 14: Level 2 Accept interaction pattern.

5.6 Level 2 ROI Request

Figure 15 shows the collaboration diagram of the ROI request interaction pattern. The interaction involves the composite objects ROB and L2IF, both of which use the scheduling pattern.

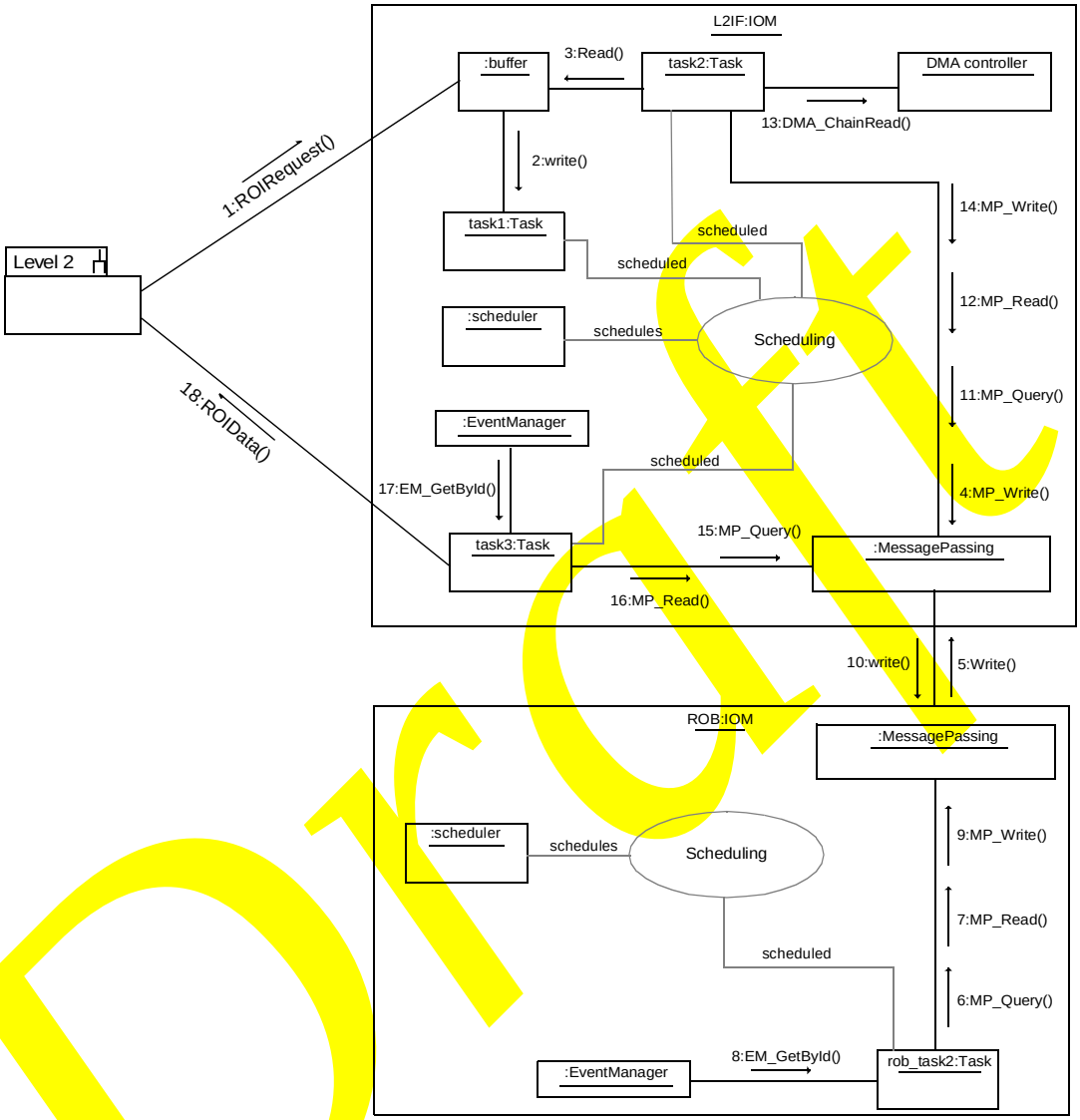

Figure 15: Level 2 ROI Request interaction pattern.

6 Deployment

6.1 General

The initial implementation of the IOMs is based on VMEbus Single Board Computers (SBCs), specifically the CES RIO 8062 (RIO2) and the MOTOROLA MVME2xxx, running the LynxOS operating system (versions 2.5.1 or 3.0.1). Communication between SBCs is via the VMEbus and, optionally, a secondary bus supporting broadcast functionality. To date, only the PCI Vertical InterConnect has been used as a secondary bus.

The SBCs mentioned above have two PMC sites and the number of supported PMCs may be increased with the addition of a PCI expansion board. Currently, only the PEB 640x has been used in conjunction with the RIO2, thus increasing the number of PMCs supported by a SBC to six.

Three deployments of the ROBIN have been made: the Local-ROBIN, the UK-ROBIN and the MFCC-ROBIN.

6.1.1 The Local-ROBIN

In this implementation of the ROBIN the Input task and the Event Manager are implemented on the processor of the SBC and ROD fragments are transferred from a ROL interface over PCI bus into the SBCs system memory under the control of the Input task. In the case that the SBC is a RIO2 or MVME2xxx the ROL interface used is the Simple SLINK to PCI (SSP) PMC. All tasks of the ROB are part of the same application, therefore, access to events via the Event Manager API are function calls.

Alternatively to the RIO2 or the MVME2xxx, this ROBIN could be deployed on an MFCC, which is architectually similar to the RIO2. This deployment is of particular interest in the case of the MFCC-ROBIN.

6.1.2 The UK- and MFCC-ROBIN

These ROBINs differ to that of the Local-ROBIN in that the ROBIN functionality is deployed over the SBC and an intelligent I/O processor. Two intelligent I/O processor, based on the PMC format, have been used for these studies, each consists of a processor (i960 or PowerPC for the UK or MFCC-ROBIN respectively), an FPGA and some “glue” logic. The FPGA handles the receiving and buffering of the data, the Intask, while the main functionality of the processor is to provide Event Management. It should be noted that the full Event Management functionality of the ROBIN is distributed over the processors of the SBC and the PMC.