FESA Essentials

The new Front-End Software Architecture (FESA) is a comprehensive framework for designing, coding and maintaining LynxOS/Linux equipment-software that provides a stable functionnal abstraction of accelerator device.

FESA Essentials provides a synthetic overview covering key concepts, sample C++ code and check-lists for equipment-specialists to get a first grasp of what equipment-software development means when relying on the method, generic architecture and tools that constitute the FESA framework.

As a new FESA user armed with the knowledge captured in the *essentials*, you are encouraged to take benefit of the tools and utilities to jump-start equipment-software development within hours. Rolling-up your sleeves, you will probably find-out that the tools on-line help and tutorials are the natural complement and immediate stage after reading this book.

1 Equipment Software

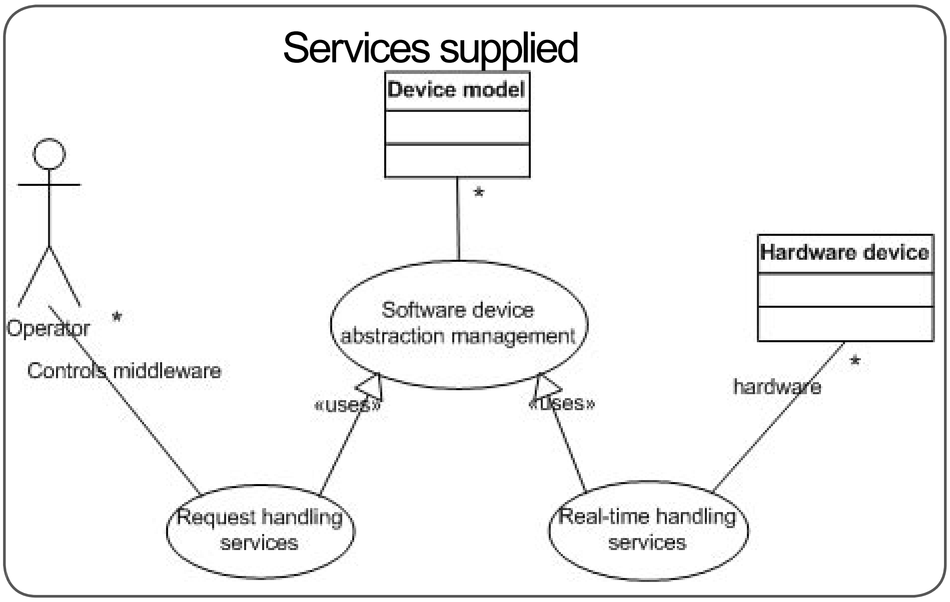
Equipment software provides a stable and homogeneous functional abstraction on top of accelerator equipment (sensors, actuators...) whose hardware implementation is heterogeneous and evolves over time.

Particle accelerators are fitted with terminal devices that can be sensors, actuators or a combination of both. From a remote control room, operators access these devices accross the control system infrastructure which consists of layers of hardware, software and communication protocols.

Equipment Software. A crucial part of the control infrastructure, it is located at the junction of two worlds: on one hand, it communicates with the control-room's computers and handles operator requests (property interface). On the other hand, it must deal directly with hardware.

Services. As depicted by the use-case diagram below, request-handling and hardware control are the two complementary services equipment-software renders. The two differ very much in nature since the former is an on-demand service, whereas the latter is subject to tight real-time constraints. Obviously, request handling must run at a lower level of priority and shall not be able to preempt and wreak havoc with the real-time task. In order to decouple the two, equipment-software includes a software abstraction of the device. Thanks to this abstraction, an operator does not directly see the hardware device, but rather accesses it through its proxy.

Software Device. The software equivalent of an underlying hardware device is a data-holder that contains attributes which can be settings, acquisitions, or dynamic state-variables, and whose values at any given time provide an accurate snapshot of the underlying hardware device.

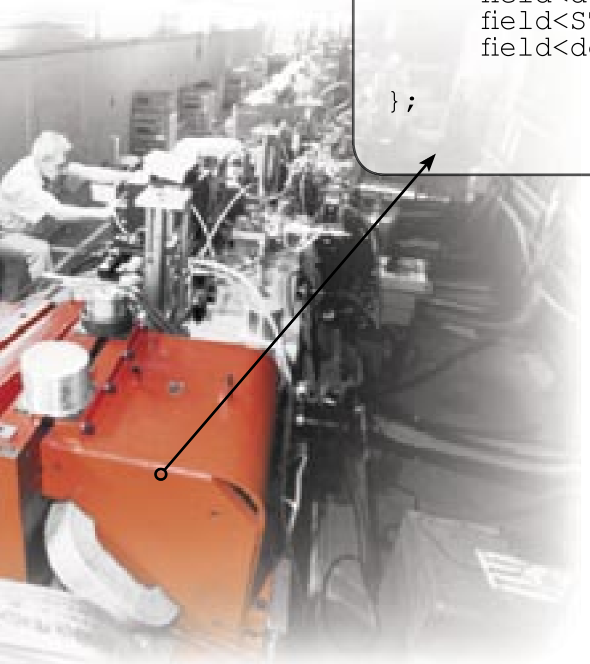


Real-time Task. An equipment-software's core activity is to ensure that both the software abstraction and its underlying hardware device continuously reflect each other's state at run-time. Ensuring such a real-time correspondence involves information flowing in both directions: Controls flow from the device model and down to the hardware; Acquisitions flow from the hardware and up to the device model. Such transfers are usually synchronized by the accelerator's central timing system which orchestrates machine activity.

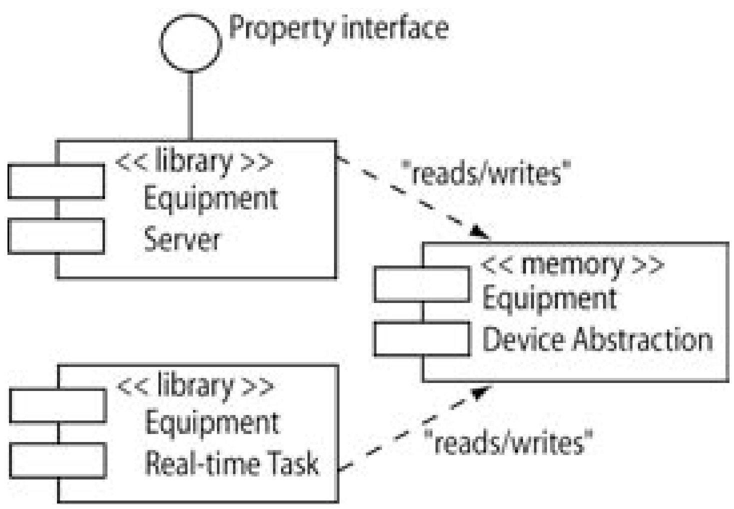
Componentization. From the above, it turns out that an equipment software function consists of three parts: a server component implementing request-handling function, a real-time task implementing real-time handling, and a memory segment embed-

Bending magnet software device

```
class BendingMagnet : public Device() {

 public:

     field<double, USER, PERSISTENT>   magneticFieldRef;
     field<double, USER, VOLATILE>     magneticField;
     field<STATUS, NONE, VOLATILE>     status;
     field<double, NONE, FINAL>  maxFieldRef;

};
```



Equipment software components



ding the software model of the device. This structure is depicted by the diagram on the right.

Implementation. Development of a new equipment software always involves coding the three above-mentioned components: defining the device model and coding respectively the request-handling and real-time handling. For performance reasons, C++ is the programming language of choice for developing real-time equipment software targetted at LynxOS, a real-time flavor of the Linux operating system.

Code reuse. In spite of the overwhelming diversity of accelerator equipments, development of equipment software exhibits some routine work, thereby suggesting that some form of code-reuse is achievable. To this end, object-orientated technologies such as those that come with C++ provide several options: inheritance, delegation and generic programming through the means of templates. Software frameworks provide the ultimate form of code-reuse.

Framework. This approach defines at defining a software package that provides a partial yet generic solution that can be tailored, i.e. customized, on a case-by-case basis in order to suit the specific needs of the equipment specialist. A framework makes for a software package that contains a set of base classes that encapsulate the essentials or key concepts of equipment-software. Customization reduces to deriving concrete classes from these base classes.

Method. Relying on the FESA framework requires the equipment-specialist to recast the problem at hand in standard form: what is the structure of the equipment data-storage and actions; how such actions are orchestrated. The analysis and design phases consist in specifying the equipment model.

Tools. Equipment-modeling is supported by a design tool, whereas automated code-generation is used to produce C++ code from the high-level model of the equipment.

2 Framework Basics

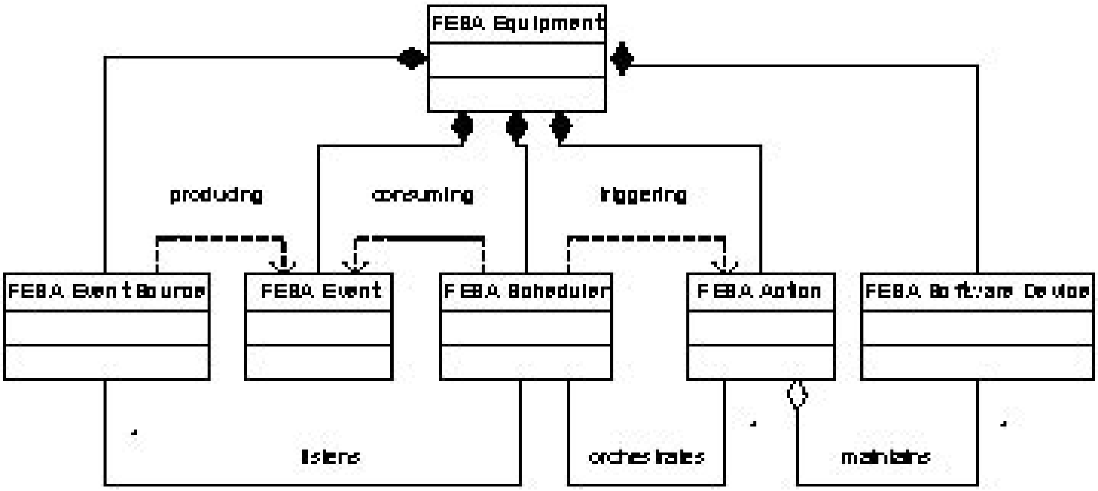
The framework approach aims at defining a software package providing a partial yet generic solution to equipment-software, and which can be refined when applied to a specific equipment.

Equipment-software development for accelerator controls has matured over more then fifteen years and exhibits some recurrent design patterns which the FESA framework intends to capture.

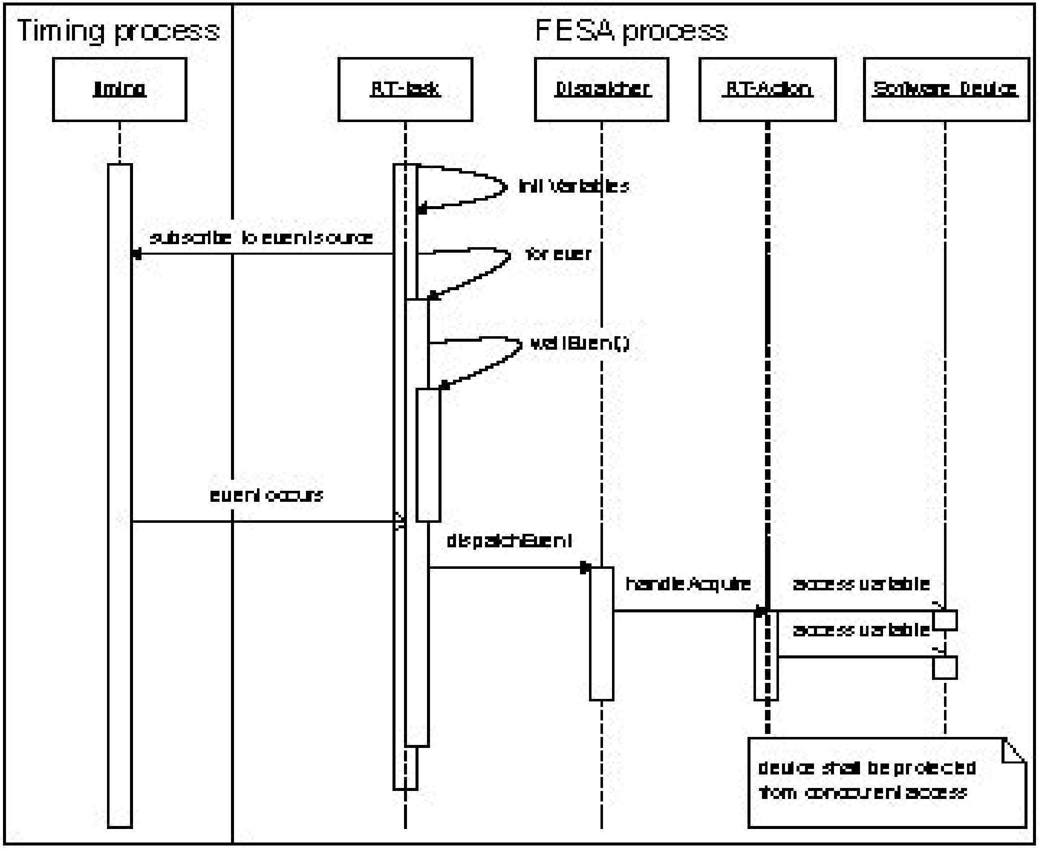
Purpose. The FESA framework encapsulates recurrent aspects of equipment-software development as a reusable software package that can be tailored – or customized – on a case-by-case basis. The generic software package contains a set of base classes that encapsulate the essentials or key concepts of front-end software. In this case, customization reduces to deriving concrete classes from the base classes and implementing them to suit specific needs. The following sections describe the object structures and theri interactions inherited from the framework, and then conclude by listing the degrees of freedom given to the equipment-specialist for tailoring the base package.

Static structure. At the heart of any equipment software activity, there's a source of events which acts as a pacemaker. At the other end, equipment-software is here to do something. And this 'something' can be structured as a set of elementary actions. Such actions correspond to the work-breakdown structure of the equipment-software's job. In between the two, one needs a scheduler which continuously listens to the event-source. Whenever an event occurs, the scheduler triggers an appropriate action according to some pre-defined logic. Actions can be further categorized as being either real-time actions, that is to say actions that

Framework overview



Logging information access



deal directly with the hardware, or they can be server-actions – that serve and fulfill operator requests. Software devices provide a convenient decoupling between these two kinds of action. A device object is simply a data-holder that contains attributes which can be settings, acquisitions, or dynamic state-variables and whose values at any given time provide an accurate snapshot of the underlying hardware device. These seven classes form the backbone of the framework's architecture.

Real-time behaviour. The scheduler is continuously listening to the event-source. Whenever an event is fired, the event-source manufactures an event-object which is forwarded to the scheduler.The scheduler examines its type and contents and triggers an appropriate action by relying on some pre-defined logic.The code of the action which is supplied by the user updates the device in read or write mode.Once the action is completed, the scheduler consumes the event and then waits for another event to occur. This whole process can be viewed as a simple event production-consumption scheme whereby the scheduler waits for events and consumes them by triggering associated actions. What this diagram shows is that this behavior is inherited from the framework. The only code provided by the equipment-specialist corresponds to the hashed activity.

Request-handling. On the client-side, equipment software provides access to the underlying hardware device, offering this service as a set of pre-defined requests that the equipment software responds to (property access). Request-types can be classified as follows: Simple read / write access to device variables. Read / write access attached to specific cycle or filtering conditions. Treatment request involving some on-demand processing within the server process, with preliminary or subsequent access to one or several instance variables by the server process.

Customization. In order to tailor the framework package and apply to a specific equipment class, the equipment specialist needs to configure it with a design tool, and then to supply pieces of C++ code that implement the actions.

Customizable parts

Part	#
Design model	1
Real-time actions	0..*
Server actions	0..*

3 Design overview

Before jumping to the C++ coding stage, development of new equipment software with FESA, first involves specifying what the equipment software is doing and what its structure is. An equipment specialist carries out this specification stage using the framework's design tool. By doing-so he describes the equipment based on some high-level modeling language.

Design of an equipment software component starts with recasting the problem at hand in a standard-form, which consists in asking and answering recurrent questions: (1) What are the published services provided by the component to the outside? (2) What is the software abstraction of the accelerator device? (3) What are building-blocks, (4) What is the real-time behaviour? The FESA design tool assists the equipment specialist in specifying the equipment from this abstract point of view.

Model. FESA defines a language through which an equipment-specialist specifies an equipment design. This language is encoded as an XML Schema with which the FESA tools comply with. The design tool enforces all design-constraints defined by the FESA grammar and lets equipment-specialists carry-out out their design work according to the degrees of freedom given to them by the metamodel. The metamodel is subdivided into several complementary areas, for which the equipment specialist has to make some design choices via the tool.

Information. A FESA class is identified by the combination of its name and a version number.

Interface. This defines the set of services published to the outside (clients from the control-room or middle-tier software layer). Designing an equipment's interface involves listing so-called «properties» that can be remotely accessed through the controls-middleware.

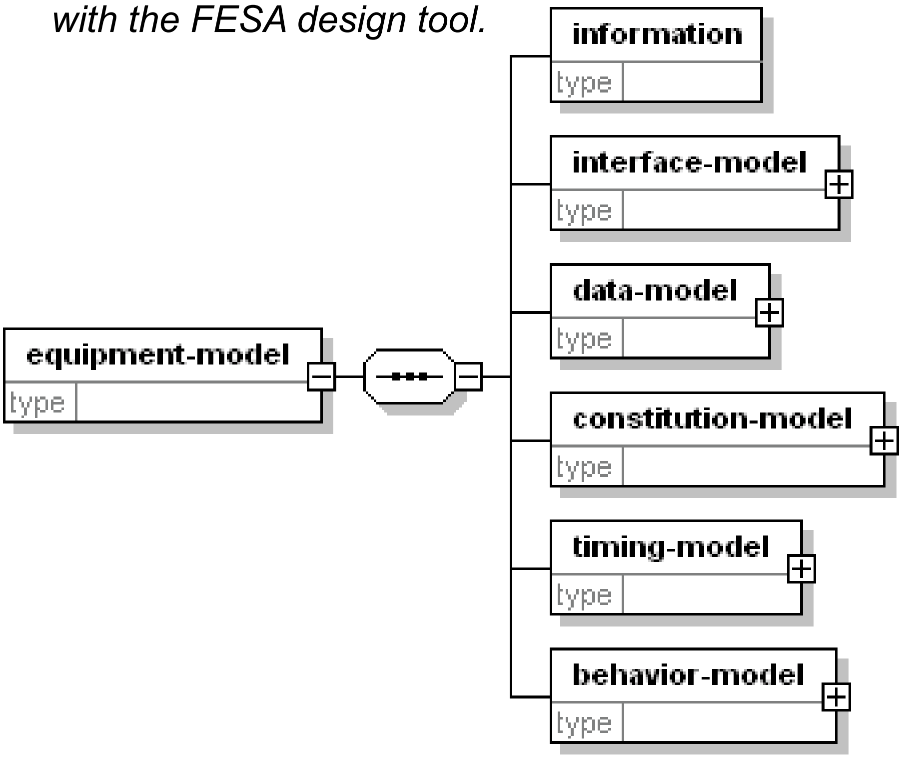
Data. At the heart of any equipment-software, the device-model is a data-holder whose attributes continuously provide a snapshot of the state of the underlying hardware device.

Constitution. Actions are the basic work-units of equipment software. They come in two flavours: the real-time actions are triggered by central-timing events and interrupts. The server actions implement request-handling. Right from the design stage, the equipment specialist has to list all the action-classes that can be execu-

Equipment specification to-do list

✓ *Define a name and version*

✓ *Define all parts of an equipment's specification with the FESA design tool.*



ted at any one time by the equipment-software component.

Timing. An equipment-software component is usually synchronized with overall accelerator orchestration by receiving synchronization events. For each class, the equipment-specialist has to define a list of logical events by giving them names within the scope of the equipment-class. Linking these logical events to accelerator or hardware interrupts is left until a later stage when the equipment is deployed on specific front-end computers.

Behavior. After having listed both the elementary actions and the triggering events, the equipment-specialist can complete the picture by relating the two, i.e. by deciding when and which action is triggered upon occurrence of an event. This last aspect of an equipment-software's design is referred to as the behavioral specification.

Recommendations. In object-orientated software development, getting the design-right from the start is even more important than for procedural languages such as C. In many cases, all the C++ classes which structure the code of an equipment-software will come from the design stage through the use of automated code-generation. Afterwards, re-architecting the software is hardly feasible. Hence it is of paramount importance that equipment-specialists devote time and effort up-front to carry-out a careful analysis and design. The fact that the tool reduces design to filling-in some forms and clicking on the mouse is not a pretext to hasten the design but rather an opportunity to spend more time on it.

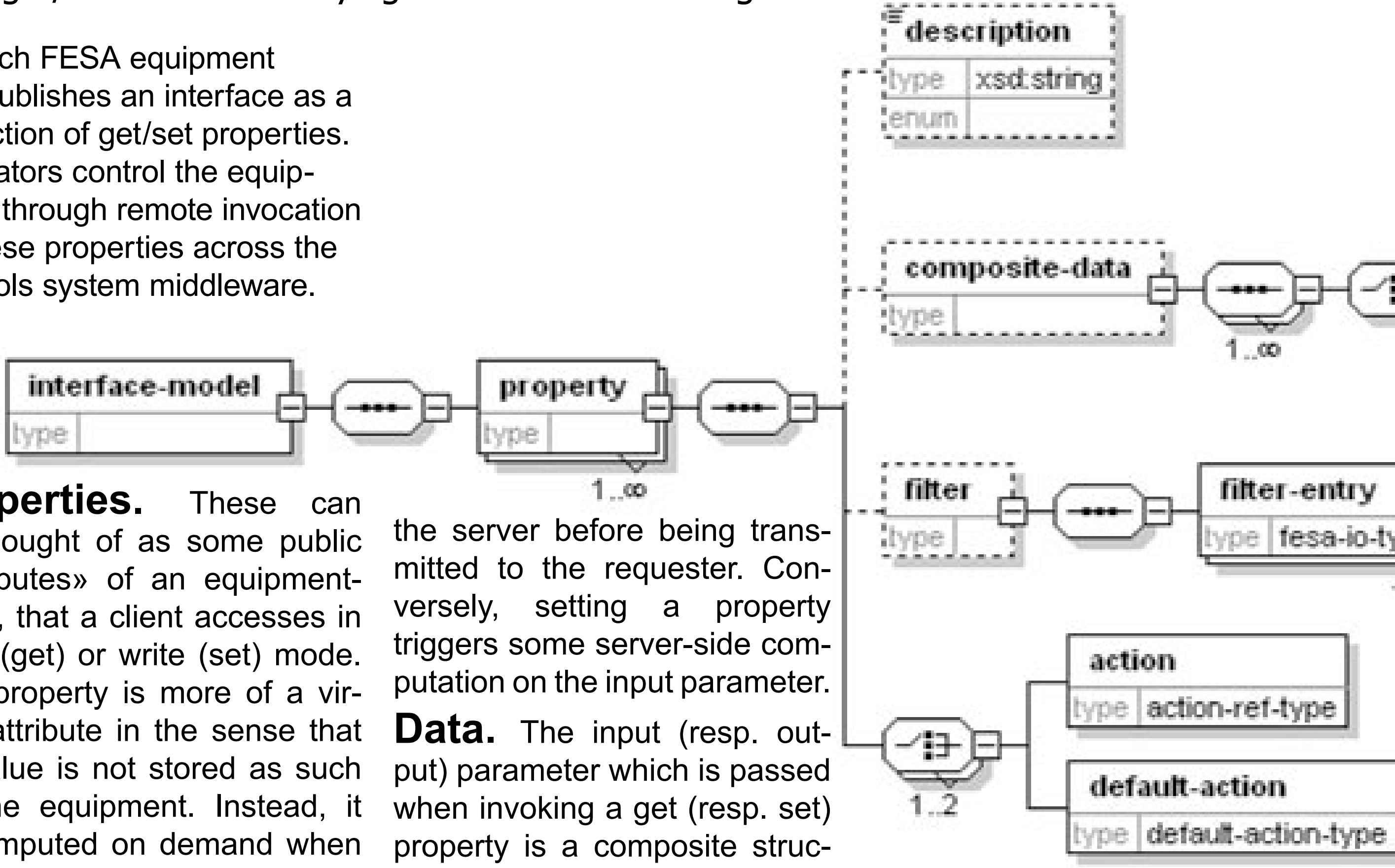
A well-formed equipment specification (interactively created with FESA design tool)

```
<?xml version=»1.0» encoding=»UTF-8»?>

<equipment-model xmlns:xsi=
  »http://www.w3.org/2001/XMLSchema-instance»
  xsi:noNamespaceSchemaLocation=
  »../../../MODEL/FESA_metamodel.xsd»>


  <information name=»Trivial» version=»0»/>


  <interface-model>
     <property name=»Acquisition»>
          <composite-data>
               <field-name-ref-data-entry>
                    sample
               </field-name-ref-data-entry>
          </composite-data>
          <default-action get-set-type=»get»/>
     </property>
  </interface-model>


  <data-model>
     <device-model>
          <fesa-field name=»sample»
          multiplexing-criterion=»MUX_NONE_ID»
          persistency=»VOLATILE»>
               <scalar type=»float»/>
          </fesa-field>
     </device-model>
     <global-store/>
  </data-model>


  <constitution-model>
     <server-action name=»Acquisition»>
          <input-field-ref field-name-ref=
               »sample»/>
     </server-action>
     <rt-action name=»Acquire»>
     </rt-action>
  </constitution-model>


  <timing-model>
     <logical-event name=»AcquisitionTiming»/>
  </timing-model>


  <behavior-model>
     <schedulable-units>
          <rt-action-ref rt-action-name-ref=
               »Acquire»/>
          <trigger>
               <explicit-event-ref
                    logical-event-name-ref=
                    »AcquisitionTiming»/>
          </trigger>
     </schedulable-units>
     <scheduling-scheme>
          <event-action-map/>
     </scheduling-scheme>
  </behavior-model>


</equipment-model>
```

3 Property Interface

This chapter summarizes the successive steps for specifying and implementing the services that an equipment software publishes to the outside world. The public interface of a FESA equipment class is composed of a set of properties. The equipment specialist may implement them by either supplying dedicated server get/set actions or relying on automated code generation.

Each FESA equipment publishes an interface as a collection of get/set properties. Operators control the equipment through remote invocation of these properties across the controls system middleware.



Properties. These can be thought of as some public «attributes» of an equipment-class, that a client accesses in read (get) or write (set) mode. The property is more of a virtual attribute in the sense that its value is not stored as such by the equipment. Instead, it is computed on demand when requested from the client. Getting a property causes the property to be computed by the server before being transmitted to the requester. Conversely, setting a property triggers some server-side computation on the input parameter.

Data. The input (resp. output) parameter which is passed when invoking a get (resp. set) property is a composite structure that aggregates one or several typed data-entries. The name and type of these individual entries are user-defined (authorized types are the same as those allowed for fields) unless the name is already reserved by a device or global-store field. When this is the case, the type of the data-entry is constrained to be identical to the type of the field that bears the same name.

Filter. This is an optional means to fine-tune the processing being carried-out when getting or setting the property. When there is no filter attached to the property, the processing is fixed. When the get or set request is transmitted, the filter is used to fine tune the treatment of the input (resp. output)

Property interface definition to-do list

✓ *name the property «MyProperty»*

✓ *define the composite-data*

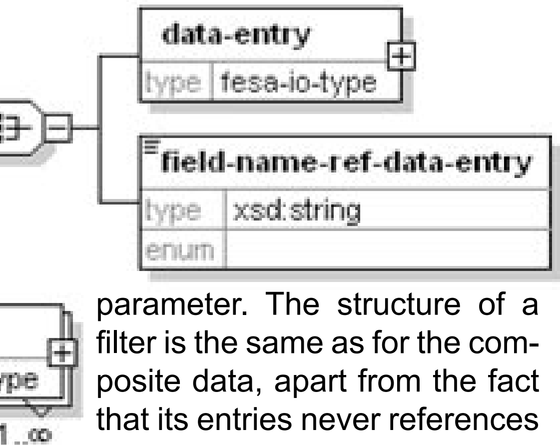
(*OPTION:*
✓ *define the filter*)

[✓ *define the get action «GetMyProperty»*
AND/OR
✓ *define the set action «SetMyProperty»*]

✓ *choose default implementation or code in C++*

Sample code of a custom get with filter

```
GetFilteredCurrent::execute(RequestEvent *ev) {

   BeamCurrentSensor * device = ev->getDevice();
   MuxContext context = ev->getContext();

   double fc = this->filter.cutOffFrequency;
   int n = this->filter.order;

   double * rawData = device->raw.get(context);
   int size = device->points.get(context);
   smoothedData = lowpass(fc, n, rawData, size);

   this->compositeData.smoothedCurrent =
      smoothedData;
   this->compositeData.length = dataSize;
   this->compositeData.time = time.get(context);
};
```



parameter. The structure of a filter is the same as for the composite data, apart from the fact that its entries never references existing fields and are instead always defined within the filter's naming scope. As an example, filters could be used for data conversion, low-pass or filtering and averaging of measurements, selection or a particular signal component or time-window, or parameters of a signal-transform (e.g. radix of an FFT).

Server Actions. Whether the property is accessed in get or set mode, its remote invocation causes some server-side processing to occur. In FESA, every object that does something on the server-side is encapsulated as a server-action. Hence, specification of a property always involves attaching it to at least a get or a set action, or may be both. All actions that require non-trivial processing (e.g. data shaping or logics) must be coded in C++ by the equipment specialist.

Get/Set coding. Programming server actions falls into the same mold as programming real-time actions: the developer needs to implement the `execute(Event*)` method of the action in which the event argument carries-out the context within which the action occurred. The composite-data and the optional filter objects are accessed within the action. The bulk of most server actions consist of transferring data in between the composite data and several fields, while applying some data-shaping that may depend on a given filter.

Default Get/Set. In certain cases the sole processing associated to getting (resp. setting) a property reduces to multiplexing (resp. de-multiplexing) the composite-data to and from the individual fields. In this case the composite is made of individual entries that refer to device or global-store fields. When this is the case, the C++ code that implements the property's get and set methods is automatically synthesized from the equipment-design specification.

How it works. When the equipment server receives a request across the controls middleware, it first packages it as an event and transmits to a server-action, similarly to the way real-time events are handled. It must also be pointed-out that request-handling activity always runs at a lower level of priority than the thread or process within which the real-time actions execute.

Recommendations. Properties form the contract that an equipment-class passes to its potential clients and should remain stable in the long run. Ideally, this interface shall be agreed-upon before-hand with the operators or programs that access the equipment. The interface should also be simple so as to present an abstract view of the equipment as seen from higher-level controls. For this reason, it is also a good practice to keep the interface short and to gather related data into coarse-granularity composite properties rather than to scatter information into several single-entry properties which do not convey enough information by themselves. The trouble with fine-grained properties is that they can cause «fragmented» traffic and may require and recombination on the client-side.

Sample code of a default get

```
GetRawCurrent::execute(RequestEvent *ev) {

   BeamCurrentSensor * dev = ev->getDevice():
   MultiplexingContext ctx = ev->getContext();

   this->compositeData.signal=dev->raw.get(ctx);
   this->compositeData.length=dev->points.get(ctx);
   this->compositeData.time=dev->time.get(ctx);

};
```

4 Scheduling

The real-time behaviour of an equipment-software is orchestrated by a central object referred-to as the «Scheduler». You can specify how it behaves by configuring an event-action map which simply associates logical events and real-time actions. By relying on this built-in map, you can design your equipment's behaviour without writing a single line of code.

Earlier on, you learned that actions are the basic work units, or building-blocks or an equipment-software's function. The scheduler puts them together and orchestrates them to form the equipment software behaviour.

Scheduling. The real-time behaviour of an equipment-software is inherited from the framework, yet it is fully customizable by the equipment-specialist who decides which real-time actions execute upon occurence of particular events.

Event-action map. This is the standard means for configuring (i.e. customizing) the scheduler. The equipment-specialists assembles the map by succesively entering a list of entries, where each entry associates a particular real-time action to an event-name. This map fully defines the behaviour of the equipment-class, without requiring any C++ programming by the equipment-specialist.

Events. Events that appear as key in each entry of the event-action map are named within scope of the equipment-class. Equivalence between such class-scope names and machine-level timing names is achieved through the means of a dedicated table which is maintained on a per-FEC basis.

Explicit triggering. Each entry of the map is a couple (event-name, action-name), where the event refers explicitly or implicitily to an event, the two approaches are mixable in the map.

Implicit triggering. Leaving the event-name of an event-map's entry blank and replacing it with a read-only device-field identifier instead means that the actual triggering-event is not known at design-stage and is postponed to the equipment-software component is initialized and loads device-instance parameters into the FEC memory. When instantiating devices, the value of the «interrupt-field» must be restricted to the logical event-names authorized within the class' scope.

Device-grouping. An instance of a real-time action typi-

Scheduler definition to-do list

✓ *select the standard event-action map*

✓ *define the list of class-scope event names*

✓ *repeat as many times as required:*

[

- ✓ *choose «explicit» triggering by selecting an event-name*
- ✓ *select action-class and optionnal device-selector*

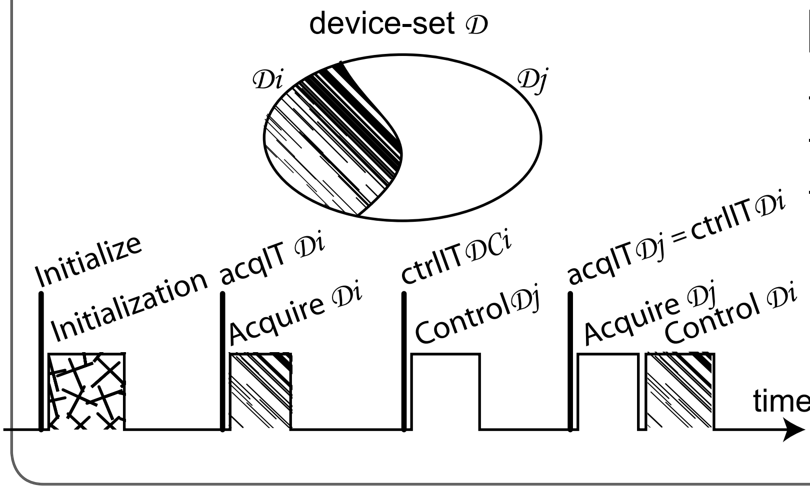
OR

- ✓ *choose «implicit triggering» with an «interrupt» device-field*
- ✓ *select action-class and a device-selector (implies «interrupt» field)*

]

Sample event-action map configurations

Logical event	Real-time action	Device-selector
Initialisation	Initialize	all
field::acqIT	Acquire	((acqIt==?)&&(hwAddress@(SIS3003/?/*)))
field::ctrlIT	Control	((ctrlIt==?)&&(hwAddress@(SIS3003/?/*)))



Logical event	FEC	Timing event
Initialization	*	sps.gen.start_cycle
it1	fecF1	sps.rocs.ring.start
it2	fecF1	sps.rocs.ring2.start

Timing configuration

Device	FEC	ctrlIT	acqIT
dSample1	fecF1	it2	it1
dSample2	fecF1	it2	it1
dSsample3	fecF1	it1	it2

Device instances

cally manages a set of devices which are homoegenous w.r.t. some criteria. For instance the `execute()` method of a real-time action can process at the same time a set devices operating at the same moment. On the other hand, physical devices are usually connected as homogeneous groups, for which access from a hardware module (VME board, field-bus adapter, PLC gateway etc...) is carried-out as a block. Block access is associated to very significant performance gain: indeed, the cost for transmitting information over a communication channel or over the bus is usually similar whether the transaction involves one or several device instances.

Data-transmission. In addition to scheduling real-time actions supplied by the developper, it is also possible to interleave upstream data transmissions due to subscribed properties. The developer specifies when the communication action occurs in the same way as for real-time actions. The procedure is described in the section devoted to subscriptions.

How it works. At initialization, real-time actions are instantiated and attached to groups of devices that meet the device-selector requirements. Each real-time action together with its attached device-collection is entered into the event-action scheduling map. When the same event triggers several actions, the order in which they execute is the same as their order in the map.

Recommendations. The scheduler is one of the most important parts of the design. In order to get it right up-front, it is advisable to generate the code from the specification and execute it as soon as possible, even before coding the real-time actions. When specifying device-selectors, one must ensure that there will be no 'orphan' device, i.e. each device must meet at least one of the logical conditions and be associated to some real-time action instance.

Samples of grouping criteria

Shared criterion	Motivation	Device-selector
Interrupt	Synchronized access of a set of devices that need to operate at the same time whithin the accelerator's timing cycle.	(AcqIT==?)
Hardware address	Improve data-transmission efficiency by accessing all devices on a same communication-node in one go.	(hwAddress@'GPIB/?/*')
Both of the above	When two-above objectives must be fulfilled simultaneously	((hwAddress@'PLC/?/*')&&(AcqIT==?))

Samples of grouping criteria

5 Device Model

At the heart of any equipment-software, the device-model represents the software abstraction of an underlying device. This is a data-holder whose fields are continuously updated and transferred to and from the hardware in order to ensure that the real device and its software proxy reflect each-other's state at run-time.

Accelerator-devices are functional pieces of equipment which extract some measurements, exert some actions on the particle-beam or do a combination of both.

Device. Specifying a proper device-model is one of the most important steps of equipment-software design. Practically modelling the device consists in defining a set of fields.

Fields. Fields make for the fine-grained fabrics of the device-model. Every piece of information about the underlying hardware-device is stored in fields. Fields are full-fledged objects that provide access methods, notably get/set accessors for C++ specialist code to store and retrieve their value.

Type	Purpose
standard	user-defined
hardware	addressing
interrupt	implicit trigger

Field types

Addressing fields. The framework defines dedicated hardware fields, which consist of a three-part combination of `type`/`logical-unit`/`channel` string fields for encoding the hardware addressing of a device. The `type` designates the hardware board family. The `logical-unit` typically identifies the board index within the VME crate. The `channel` typically refers to a specific port of the board which is used to connect the specific device-instance. For devices which are connected through more than one hardware-module, it is possible to rely on up to three sets of such address-fields.

Interrupt fields. The framework also defines dedicated interrupt-fields, which can be referred-to as implicit-triggers of real-time actions by the equipment's scheduler (see chapter on scheduling for details). The device model may refer to one or several interrupt fields.

Device modeling to-do list

✓ *Define device hardware addressing as a set of hardware-field string triplets of the form:* (`type/logical-unit/channel`)

✓ *OPTION: define interrupt fields*

✓ *define standard fields*

Standard fields. Apart from two pre-defined types dedicated to hardware-addressing and implicit-triggering, the framework does not impose any detailed class hierarchy for the standard fields. Hence, the equipment-specialist has complete freedom for defining what fields actually stand-for. To this end, it is important to keep in mind the functional purpose of each before deciding to make it a persistent or a multiplexed one. To this end, the table entitled «Fields categories according to functional purpose» proposes a taxonomy of fields inspired from the standard terminology used in the domain of dynamic systems and compatible with accelerator operations usage.

Multiplexing. The accelerator-complex provides beams to several users, making it a shared resource that relies on time-multiplexing scheme orchestrated by the central timing-system: the basic period of the accelerator is split as a set of successive time-slots during which the settings of a specific user stay valid. Switching from on multiplexing context to the next is triggered by the timing system, which in turn causes a switch of equipment settings from one user to the next. Accounting for this multiplexing behavior at the device-level requires that fields accommodate not a single

Field-access code-fragment

```
// pCtx is an opaque multiplexing-context object
// passed along the action's triggering event.
// pDev is a pointer on a device instance

float currentUserVoltage=pDev->voltage.get(pCtx);
```

but a set of values, namely one different value for each different user. As illustrated by the above code-fragment, such multiplexing-management is transparent to the equipment-specialist whose sole responsibility is to define which fields are multiplexed, and with respect to which criterion. Possible criteria are listed below:

Multiplexing	Purpose
`NONE`	not multiplexed
`USER`	cycle user
`PARTICLE`	particle-type
`DESTINATION`	beam-target

Multiplexing criteria

Persistency. Fields are accessed to and from the FEC memory at run-time. For backup and data-management purposes, they can be assigned different persistency levels as defined below:

Persistency	Purpose
`FINAL`	database constant.
`PERSISTENT`	periodic backup save into persistent-storage.
`VOLATILE`	RAM data.

Persistency

Standard data-types. Data-types are restricted to the ones supported across the whole controls system by the communication middleware, which comprises the following:

C++ Scalar type	Size
`bool`	1
`signed char (byte)`	8
`short`	16
`long`	32
`long long`	64
`float`	32
`double`	64

Scalars

Unsigned types are not supported in conformance with a middleware restriction which can be traced-back to the fact that there are no unsigned types in Java.

Array type
`bool`
`signed char`
`char`
`short`
`long`
`long long`
`float`
`double`

Uni and bi-dimensional arrays

Types allowed for uni-dimensional and bi-dimensional arrays are identical to those permitted for scalars, with a major difference: the `char` array type is meant as a C-style null-terminated string holder whose dimension stands for the maximum size allowed.

Extended types. In addition to the standard types, one may rely on either custom types (e.g. enumerations and bit-patterns) or extended types (types brought into the design by inheritance). You must be careful with such types as they are not transmissible as such by the middleware.

How it works. Fields are managed by the framework as C++ template classes. This means that field-access does not incur the cost of a virtual function typical of inheritance schemes. This also means that there is no hard constraint regarding the types supported by the framework, which are only constrained by those required by the rest of the system for serialization, data-transmission and storage. During initialization, the values of the `FINAL` configuration parameters are retrieved from the data-base and stored into the FEC's memory. `PERSISTENT` fields are restored to the value they previously held before the reboot.

Field categories according to functional purpose

Category	Example	Persistency	Access	
			by real-time action	by server-action
Configuration parameter	an hardware setting	FINAL	get	get
Operational parameter	an amplifier gain setting	PERSISTENT	get	set (/get)
Acquisition	measurements	VOLATILE	set	get
Setting	a bending magnetic field	PERSISTENT	get	set (/get)
State-variable	past inputs' shift-register	VOLATILE	get/set	get/set

Taxonomy of fields

6 Subscriptions

Sensors typically acquire measurements as time-sampled signals and/or on certain time-windows. When a client or middle-tier program subscribes to a sensor's acquisition property, an upstream communication channel is established through which sensory information flows. In this section, you will learn how to interleave and synchronize this upstream data-flow with real-time task activity.

Acquisition equipment can require significant upstream bandwidth and CPU resources. Hence, it is important for the equipment specialist to have control on when acquisition data are sent across the network. Two levels of control are possible with the FESA framework: a semi-automated upstream data-flow control scheme or a full-custom manual option.

Subscriptions. Any property which is served by a get action may be subscribed to by a remote client. The client expects to be notified for property changes by the equipment software. Upon notification, the controls-middleware invokes the get action and transmits the data upward to the remote client.

Automatic scheme. in this mode, there is no need for the equipment-specialist to notify each each individual property being updated within a real-time action. The framework keeps tracks of the changes on its own. In the meantime, the equipment-specialist still has control on when the data-transmission takes place. To this end, the equipment-specialist must enter a two-stage specification: first define a `cmwNotification` action within the equipment-classe's behaviour-model, then define when this action is triggered in the behavior-model (see figure 2).

Subscription management to-do list

✓ *define a cmwNotification action, which selects the automatic update mode.*

✓ *specify when the notification occurs in the behavior model*

OR

✓ *do not define any cmwNotification action, which selects the manual update mode.*

✓ *invoke the property update call from within real-time actions that cause a property change*

✓ *maintain the above dependencies for each subsequent modification of either the property interface or the C++ code of server and real-time*

Manual scheme. In this mode of operation, the equipment-specialist is fully responsible for deciding when and which property must be updated, and by which real-time action. This provides a finer level-of-granularity for controlling how upstream communications interleave with real-time activity. On the other hand, it may be tedious to maintain the dependencies between properties and actions manually, especially in the case where either the interface or the action's implementation are meant to evolve independently from each other...

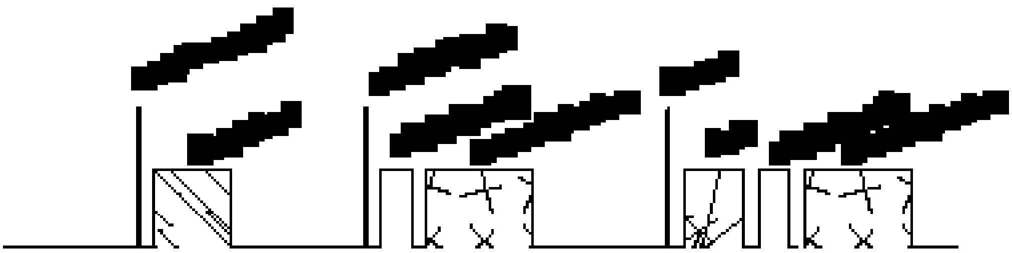
How it works. For the manual mode of operation, nothing happens under the hood of the FESA framework and how it works is really up to the equipment-specialist. For the automatic scheme, a so called `Recorder` core class of the framework keeps track of

Example of Automatic property update interleaved with subscriptions

```
class BeamSensor : public Device {
   field<HWADRS,NONE,FINAL>         hwAdrs;
   field<int,NONE,FINAL>            ctrlIt;
   field<int,NONE,FINAL>            acqIt;
   field<float,TGM_USER,VOLATILE>   cur;
};
```

Logical event	Action	Device-selector
field::crtlIT	Control	((hwAddress==?)&&(ctrlIt==?))
field::acqIT	Acquire	((hwAddress==?)&&(acqIt==?))
Reset	Init	*
endOfCycle	cmwNotify	* *(implicitly those devices for which property changed...)*
Reset	cmwNotify	* *(implicitly those devices for which property changed...)*

Behaviour-mc



all real-time activity. The `Recorder` is notified at run-time of the completion of real-time actions and of the context within which they were triggered. From this dynamic information, the list of updated properties is maintained by the `Recorder`. This involves property-action relationships built-up at initialization from static information coming from the model about dependencies between actions and fields. When the `cmwNotification` action is triggered in accordance to the equiment class' behavior model, it causes subscribed properties to be accessed in get-mode. At the same time, it resets the update-history maintained by the `Recorder`.

Recommendations. Grouping data is an efficient way to improve transmission over a packet-switched network. It is better to group pieces of information that are meant to be subscribed-to by remote clients as composite properties rather then to scatter them in numerous, low-granularity properties.

Example of manual property update

Property name	Data	GetAction	Output fields
Current	cur	GetDefault	*irrelevant*
CurrentAverage	averageCur	GetCurrent	cur timeWindow

Interface-modelextract (properties and associated get-actions)

```
class Acquire: public RTAction<RTEvent, BeamSensor>;


Acquire::execute(RTEvent *ev) {

   BeamSensor* device = ev->getDevice();
   MuxContext ctx = ev->getContext();

   for (unsigned int i=0; i<devCol.size(); i++){
      BeamCurrentSensor* dev=devCol[i];
      AcquisitionBoard* board=
           AcquistionBoard(dev->hwAdress.get());
      float current=board.getSample();
      dev->cur.set(ctx,current);
   }

   // updating the «current» field implies
   // that the following properties are also
   // updated for the device-collection managed
   // by the Acquire instance.

   Current.update(ctx, devCol);
   CurrentAverage.update(ctx, devCol);

};
```

9 Real-time Actions

Real-time actions transfer data to (resp. from) the hardware device and from (resp. to) its software counterpart through hardware interface modules. They do so within a loop involving a subset of devices which are grouped according to some pre-defined criteria. There is usually one instance of a real-time action class per homogeneous subset of devices.

The set of real-time actions defines the work-breakdown-structure of an equipment software's real-time task. Implementing them is the main C++ coding task for the equipment-specialist.

Real-time Actions. They are the basic work-units of the equipment-software's real-time task., i.e. any real-time handling activity is eventually broken-down as a set of elementary actions. For each action class, say `MyAction`, the equipment-specialist has to implement in C++, the body of a `MyAction::execute(RTEvent *)` method. The input parameter of this method refers to an `RTEvent` object that carries with it the wake-up context within which the action is invoked by the framework's real-time scheduler.

Device-collection. For each front-end on which an equipment class is deployed, a set of device instances are configured. An instance of a real-time action class typically manages a group of devices that are homogeneous with respect to certain grouping criteria (e.g. devices connected to the same communication-node or hardware module or devices that need to operate at the same time). There can be as many instances of a given action class as there are homogeneous groups resulting from the scheduler's device-selectors.

Device-loop. Transfers take place in a device loop which involves all devices attached to the real-time action instance. This set of devices is referred to as the action's `deviceCollection`. Sample C++ code for the device-loop is depicted in the «device-loop» code fragment shown on the next facing page.

Wake-up context. The multiplexed nature of accelerator usage means that the same action may be invoked from within different multiplexing contexts of the machine. In turn, this implies that the field's storage for settings and acquisitions may differ between two successive invocations of the real-time action's `execute(RTEvent *)` method. As discussed in chapter 12, FESA allocates several slots for multiplexed fields, each slot being dedicated to one specific usage-context of the field. In order to let the real-time action work-out the appropriate slot, the event embeds a `MultiplexingContext` object containing information about the machine context. As a matter of fact, the real-time action does not need to decode the context, which means that the equipment-specialist shall not worry about the actual context passed when implementing the `execute(RTEvent *)` method. Instead the framework does it transparently for him when he invokes the field's `get` and `set` methods by and transmits the

Logging to-do list

✓ *Identify the minimal information which is sufficient to convey understanding of the normal course of execution of your program. Register such pieces of information with `INFO` statements appropriately located in your code.*

✓ *Identify what could go wrong at run-time.For each situation, list the pieces of information necessary to pin-point the source of the problem and insert one or several `DEBUG`, `WARNING`, `ERROR` and `FATAL` statement to register them precisely.*

Real-time action sample

```
// This example supposes a specific kind of analog measurement devices
// which are connected through a set of VME acquisition boards
// (one VME board manages one or several devices, each device being
// connected to one specific channel of the board). It is assumed
// that the on-board buffer of each acquisition board samples
// inputs from all devices at the same time.

Acquire::execute(RTEvent * pEv) {

   try {

      // Perform block-access of the VME board, which is shared by
      // the device-collection managed by this instance of Acquire.

      // Board's logical unit number can be asked to any, e.g. first device
      Board* pBoard = Board::getBoard(deviceCollection[0]->hw1_lun.get());
      Buffer buffer = pBoard->getBuffer(ALL_CHANNELS);

   }
   catch(...){
      log<<«Failed to retreive buffer from acquistion board»<<endError;
      return;
   }

   for (unsigned int i=0; i<deviceCollection.size(); i++){

      VacuumSensorDevice* pDev=deviceCollection[i];

      float measure = buffer.extractChannelDataAsFloat(pDev->hw1_ch.get());
      pDev->pressure.set(measure);

   }

};
```

context as a parameter (see chapter about multiplexing).

Hardware access. Within the device-loop, the real-time action transfers data between a device's fields and «appropriate» channels of the hardware module. Finding-out the appropriate channel is achieved by decoding the hardware-addressing field of the current device-instance.

Error handling. Several abnormal situations may occur within the course of running the body of the `execute(RTEvent *)`. Refer to the chapter on logging to find out how to report such situations.

How it works. Invocation of the real-time action's `execute(RTEvent *)` method at the specified time is ensured by the framework scheduler which also transmits the triggering event. The current mono-thread execution model of the framework lies on the assumption that the duration of each real-time action is negligible when compared to the repetition-rate of the triggering interrupts. The equipment specialist is responsible for making sure that the assumption practically holds. To this end, he has to check that the timing requirements of his equipment do not lead to an over-run situation.

Recommendations. Real-time actions are meant to be implemented as independent, possibly reusable C++ classes. The standard way to "compose" them together is to attach them to the same triggering event at the scheduler-level, in which case they are invoked in defined sequence. Real-time actions can also exchange information by sharing fields. On the other hand, C++ theoretically also makes it feasible to rely on (a) delegation to custom handy classes and (b) implementation-inheritance amongst real-time actions. Such approaches are strongly discouraged. They can all too often be abused and misused. They result in tight couplings among custom action-classes which makes them hard to reuse elsewhere. Furthermore elaborating the C++ class hierarchy beneath the framework buries it into your C++ code without any visibility at the design-document level.

10 Logging

Logging's purpose is for an equipment-software component to notify «interested parties» that something noteworthy, unusual, or abnormal occurred within the course of execution. To this end, FESA defines an API and mechanism for developers to log run-time information with a range or severity levels that conform to the log4j standard.

Logging-support takes the form of a `log` class derived from C++ style string-streams. This makes logging similar to writing messages to the standard output, apart that it also requires setting a severity level.

Logging objects. Equipment-specialists can log run-time information messages by relying on a dedicated `log` object. Logging objects are available from the four different locations in which the specialist can place some custom code: from within the `execute()` method of a server-action, from within the `execute()` of a real-time action, from within the `specificInit()` method of an equipment's interface part, or from within the `specificInit()` method of an equipment's real-time part. In each situation, the logger is accessed in the same fashion as a local `log` object. However, there are actually four distinct logger object instances, one for each of the four usage context mentioned above. Note that all server-actions share the same log object. Similarly, there is one single log object shared by all real-time actions (see facing table).

Logging API. The `log` object inherits from the C++ string stream class (`ostrstream`), which implies that logging messages can be sent to the logging-system in a fashion much similar to the way one writes into a standard C++ stream such as `cout`. This means that logging an error for instance, is achieved with a stream output call of the form: `log<<«this is an error»<<endError`, where `endError` marks the end of the error message. The stream can be fed-in with character strings as well as with numeric types, whose conversion and formatting into character strings is automatic.

Logging levels. FESA directly borrows logging levels from the log4j standard (`http://jakarta.apache.org/log4j/docs/api/index.html`). As registered in the table on the facing page, there are five such levels, namely `DEBUG`, `INFO`, `WARNING`, `ERROR`, `FATAL`. Setting the appropriate level consists in terminating each logging message with a dedicated ending.

How it works. Messages logged through the C++ `<<` operator flow-in into a local stream-buffer. This local buffer is kept into local memory until the stream is explicitly flushed by terminating the log with an `endDebug`, `endInfo`, `endWarning`, `endError` or `endFatal`. Until encountering such an ending call, the local buffer progressively grows-up and consumes memory in order to accommodate the increased string. It is therefore assumed that the equipment-specialist makes sure that the logging of each message is followed by an appropriate ending. In case, the equipment-code fails to do so due to a programming oblivion, some built-in

Logging to-do list

✓ *Identify the minimal information which is sufficient to convey understanding of the normal course of execution of your program. Register such pieces of information with `INFO` statements appropriately located in your code.*

✓ *Identify what could go wrong at run-time. For each situation, list the pieces of information necessary to pin-point the source of the problem and insert one or several `DEBUG`, `WARNING`, `ERROR` and `FATAL` statement to register them precisely.*

Logging levels and files

Severity	Purpose	C++ marker
DEBUG	«fine-grained informational events that are most useful to debug and application».	`endDebug`
INFO	«informational messages that highlight the progress of the application at coarse-grained level».	`endInfo`
WARNING	«designates potentially harmful situations».	`endWarning`
ERROR	«designates error events that might still allow the application to continue running».	`endError`
FATAL	«designates very severe error events that will presumably lead the application to abort».	`endFatal`

Severity levels, as copied from the log4j standard (http://jakarta.apache.org/log4j/docs/api/index.html)

Log file	Purpose	File type
RtLog	records log information from all real-time actions.	FIFO
ServerLog	records log information from all server-actions.	FIFO
InitRtLog	records log information from the initialization-stage of the equipment-software's real-time component.	static after initialization.
ServerLog	records log information from the initialization-stage of the equipment-software's server component.	static after initialization.

Logging files available for each equipment-class on a given front-end computer

mechanism may force the flush whenever the internal stream buffer size exceeds some predefined limit. In this case, the message is logged at `WARNING` priority and complemented with an indication of the stream's forced flush. Each equipment-class deployed on a given front-end computer possesses its own of set of four `log` objects as described above. On the front-end computer, logging information from different equipment-software components is channelled through a message queue and dumped into files by a couple dedicated processes which perform some time-stamping. Accuracy of such automatic time-stamping is approximative as it not applied at the source. Files are allocated a configurable maximum length, which means that information may be lost after some time.

Recommendations.

Do not overlook logging as this is the primary and almost sole means whereby you can convey detailed information about what is going-on once your equipment is deployed and running. At development and debugging stage, it may be tempting to bloat your code by scattering messages you find useful in order to get your code right but which may prove useless afterwards. Hence, adequate logging is probably better thought-of at a late stage of the development cycle, once the code is stabilized. You may then make sure that logging messages convey coarse-grained information about the normal course of execution on one hand; fined-grained details that help you diagnose the

Sample code with several logging statements

```
AcquireTemperature::execute(RTEvent *ev) {

   log << «acquire temperature» << endInfo;

   for (unsigned int i=0;i<deviceCollection.size();
      i++){
      ThermometerDevice* pDev=deviceCollection[i];
      AcqBoard* board= getAcqBoard(pDev->hw.get());
      try {
            float temperature=board.getSample();
            if (temperature<0.0) {
                  log   << temperature
                        << « is too cold!»
                        << endWarning;
            }
      } catch (...){
            log   << «accessing thermometer »
                  << pDev->name.get()
                  << endDebug;
            log   << «fail to access board»
                  << dev->hwAddress.get()
                  << endError;
      }
}
```

11 Inheritance

When some classes of equipment software exhibit functionnal, structural or behvioral similarities, this suggests code reuse or some form of sharing. FESA modeling language supports inheritance to reuse model-parts of existing classes.

The FESA framework is based on inheritance: an equipment-specialist applies and tailors the framework by deriving concrete classes from a set of core abstract classes. This chapter presents a means for equipment specialists to achieve custom code-reuse through the same means of OO inheritance.

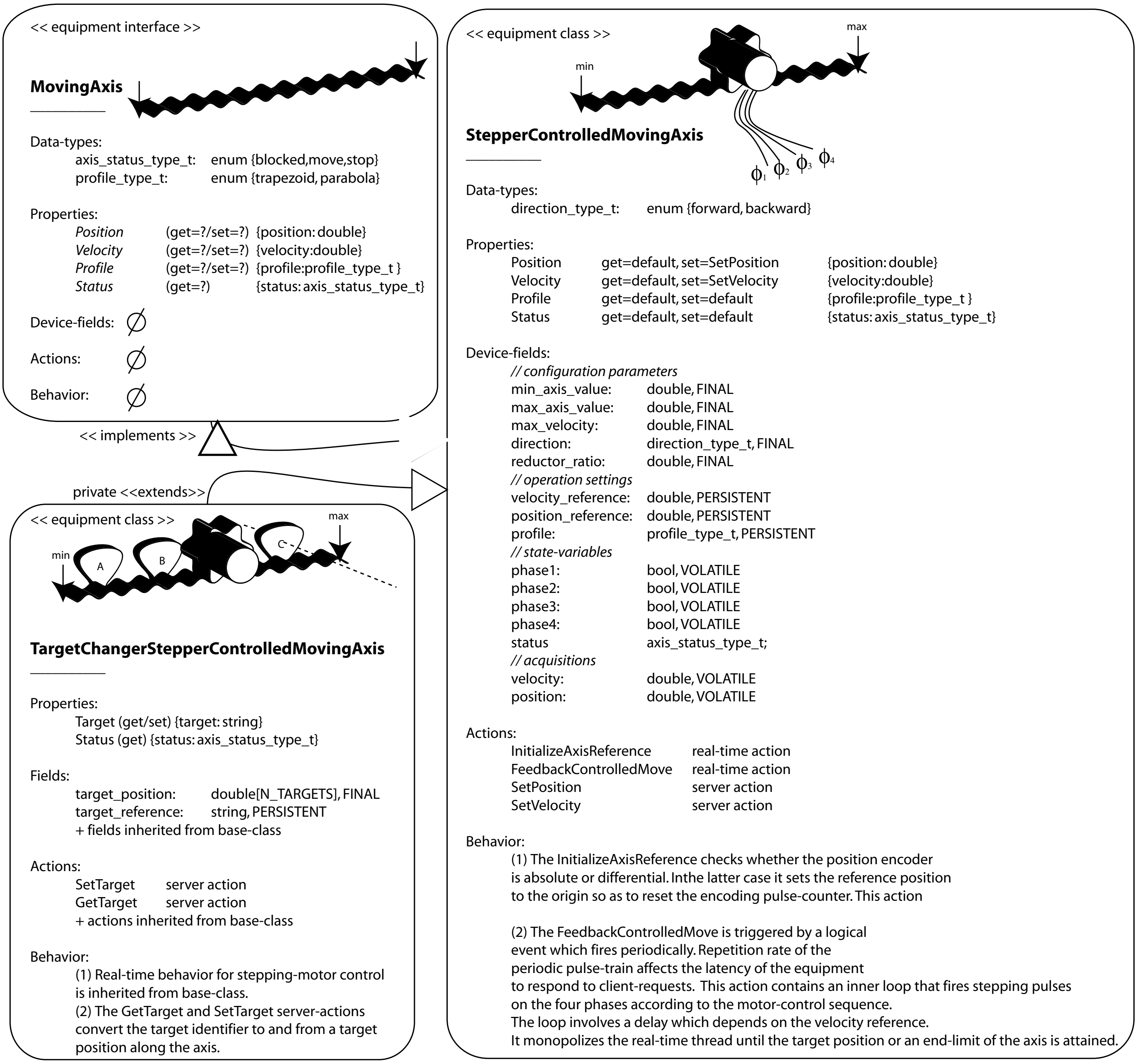
FESA inheritance. Contrary to the common form of inheritance encountered in C++ for which refining a general class into a more specialized one simply consists in making the latter inherit from the latter, one must be more cautious when dealing with inheritance within the FESA framework. Indeed stating merely that "a class of equipment software B inherits from another class B" does not mean anything in this context. You have to be more specific here, i.e. do you want to inherit the interface of the equipment-software? it's device model? its set of server-actions? its set of real-time actions? its behavior? Or do you want to inherit everything all at once? The short answer to all these questions is that in FESA you never inherit completely one class from another one (albeit you can start a design by copying a reference one). Rather, inheritance of a new equipment-class is achieved bit-by-bit, i.e. you can elaborate a new class by picking parts of a design here and other parts there, from several other classes. The kinds of building bricks you can "inherit" from are listed in the following sections.

Extend or implement? May be the first question to ask yourself when talking about inheritance is whether you need to inherit from an interface or from an implementation. In Java the two options are distunguished by different keywords: "implement" refering to the former, "extend" referring to the latter. In C++, the two notions are dealt-with with pure virtual (abstract) and virtual methods. With FESA, the choice first depends on the class you inherit from: full-fledged equipment-classes possess both an interface and an implementation whereas there are pure interface classes. Second, it eventually goes down to which model-parts he picks-up from them. The diagram shown on the next facing page illustrates the two approaches of interface and implementation inheritance.

Property inheritance. This is probably the most important form of inheritance. You use it when you want to provide a new class with an existing (and perharps long-used) interface to the operators and middle-tier. By doing so, you let applications access your equipment in the same fashion they already access the base class. Inheriting properties means that you assign the same set of properties together with their composite data and filter structures to the new equipment class. This is however a pure interface inheritance: the get and set actions that implement the properties (default or custom) do not come together by default and you'll have to either re-implement them or to inherit other complementary parts (see below).

Field inheritance. The next logical step after inheriting properties is to inherit fields (possibly with custom data-types). You may want to inherit all the fields of the device-model. Alternateky, you may only require a subset of those fields.

Action inheritance. Once a class inherits a set of fields defined by other classes, one may consider the third level of inheritance: inheriting the implementation code of actions. A prerequisite is that the input and output fields attached to the actions are defined by the equipment-software class. Both server and real-time actions can be inherited. Since the C++ code is based on templates, the very same action code will indeed work with



<< equipment interface >>

MovingAxis

Data-types:

axis_status_type_t:	enum {blocked,move,stop}
profile_type_t:	enum {trapezoid, parabola}

Properties:

Position	(get=?/set=?)	{position: double}
Velocity	(get=?/set=?)	{velocity:double}
Profile	(get=?/set=?)	{profile:profile_type_t }
Status	(get=?)	{status: axis_status_type_t}

Device-fields: ∅

Actions: ∅

Behavior: ∅

<< equipment class >>

StepperControlledMovingAxis

Data-types:

direction_type_t:	enum {forward, backward}

Properties:

Position	get=default, set=SetPosition	{position: double}
Velocity	get=default, set=SetVelocity	{velocity:double}
Profile	get=default, set=default	{profile:profile_type_t }
Status	get=default, set=default	{status: axis_status_type_t}

Device-fields:

// configuration parameters	
min_axis_value:	double, FINAL
max_axis_value:	double, FINAL
max_velocity:	double, FINAL
direction:	direction_type_t, FINAL
reductor_ratio:	double, FINAL
// operation settings	
velocity_reference:	double, PERSISTENT
position_reference:	double, PERSISTENT
profile:	profile_type_t, PERSISTENT
// state-variables	
phase1:	bool, VOLATILE
phase2:	bool, VOLATILE
phase3:	bool, VOLATILE
phase4:	bool, VOLATILE
status	axis_status_type_t;
// acquisitions	
velocity:	double, VOLATILE
position:	double, VOLATILE

Actions:

InitializeAxisReference	real-time action
FeedbackControlledMove	real-time action
SetPosition	server action
SetVelocity	server action

Behavior:

(1) The InitializeAxisReference checks whether the position encoder is absolute or differential. Inthe latter case it sets the reference position to the origin so as to reset the encoding pulse-counter. This action

(2) The FeedbackControlledMove is triggered by a logical event which fires periodically. Repetition rate of the periodic pulse-train affects the latency of the equipment to respond to client-requests. This action contains an inner loop that fires stepping pulses on the four phases according to the motor-control sequence. The loop involves a delay which depends on the velocity reference. It monopolizes the real-time thread until the target position or an end-limit of the axis is attained.

<< equipment class >>

TargetChangerStepperControlledMovingAxis

Properties:

Target (get/set) {target: string}
Status (get) {status: axis_status_type_t}

Fields:

target_position:	double[N_TARGETS], FINAL
target_reference:	string, PERSISTENT

+ fields inherited from base-class

Actions:

SetTarget	server action
GetTarget	server action

+ actions inherited from base-class

Behavior:

(1) Real-time behavior for stepping-motor control is inherited from base-class.
(2) The GetTarget and SetTarget server-actions convert the target identifier to and from a target position along the axis.

any concrete class of device.

Behavior inheritance. This option is theoretically feasible. In the case the new equipment class does only defines server-actions, as in the illustration above, it makes sense to inherit real-time behavior. Otherwise it may be very intricate to decide how inherited real-time actions interleave with the ones introduced by the derived class.

Copy vs inheritance. In some cases, you simply want to start a new design from an existing template, in which case you are more interested in copying bits of other equipments and assemble those parts within the new class. What you can do with inheritance (i.e. picking these and those parts from other classes), you can do it with copy. The difference is that inheritance implies a reference and a long-term relationship between the base class and its derived sub-class, whereas copying is a one-shot decision with no further binding. If you are not sure, you should go for the copy approach then for inheritance since it creates some burden in the long run.

How it works. Inheriting parts of a design from another existing class draws a reference from the former to the latter, which means that any subsequent change to the base class will be reflected in the derived classes.

Recommendations. Interface inheritance is advisable once a clear set of interfaces has been established by operation. In the meantime, be very careful when attempting implementation inheritance since things can then get complicated and hard to maintain: any subsequent change to the base class may have disastrous impact on all the derived classes. Whenever you think about relying on inheritance, also ask yourself whether a mere copy would do.

11 Synchronization

Synchronization of a software equipment is a deployment issue which complements the scheduler's configuration. The latter step binds actions to logical events, named within the equipment class' scope. The former step associates logical events to the underlying machine events that orchestrate the whole accelerator's activity.

Synchronization configuration attaches an equipment software to the central timing system of the accelerator complex. This synchronizes an equipment-software's real-time activity with the overall orchestration of the machine.

Accelerator timing. The central timing system is responsible for the temporal coordination of the accelerators' complex. This system manufactures machine events which are distributed across a dedicated timing network to the various front-end computers.

Event sources. The FESA framework features a set of pre-defined event-sources: a periodic event source whose repetition rate is customized by the equipment-specialist, and several timing event-sources (one per timing-domain) which fire at a pace synchronous with the accelerator's central timing system. The latter type of source is the most important in that it is the standard means for synchronizing an equipment software activity with the overall orchestration of the accelerator-complex activity. Configuration of the timing event sources involves the following. Specifying the scheduler's configuration consisted in entering a list of event-action couples, where the event could be either explicit or implicit.

Synchronization to-do list

PREREQUISITE:

✓ define explicit triggering rules in scheduler
AND/OR
✓ define implicit triggering-rules in scheduler

FOR EACH DEPLOYMENT UNIT FEC&CLASS :

✓ define the timing domains known on FEC

✓ define the synchronization binding as a set of pairs (logical event, timing event)

OPTIONAL: in case of implicit triggers only

FOR EACH DEVICE-INSTANCE ON A FEC :

✓ select triggering field's value from set of logical events available on the FEC.

Explicit triggers. Explicit triggering means that the real-time action is executed whenever the timing event-source fires an event whose logical name is identical to the one specified by a given event-action firing rule.

Implicit triggers. This links an action to a custom-defined device field. The field value of each device instance is restricted to be one of the logical event names. This constraint is enforced by the FESA device-instantiation tool.

Event-binding. This is a deployment-stage configuration that temporally connects your equipment's behavior to the underlying machine activity. For each deployment-unit, i.e. for each pair (front-end computer, equipment class), a map associates the class's logical events to corresponding machine events. This map consists of a set of

Synchronization information sample

Logical event
BeamStart
BeamEnd
Forewarning
Initialization

Timing-model

Trigger	Real-time action	Device-selector
Initialisation	Initialize	all
field::acqIt	Acquire	((acqIt==?)
field::ctrlIt	Control	((ctrlIt==?)

Behavior-model

Logical event	Underlying timing event
BeamStart	pix.sinj
BeamEnd	pix.apow
Forewarning	pix.fpow
Initialization	-

Deployment of the equipment-class on a FEC

Name	field::acqIt	fieldCtrlIt
dev1	BeamStart	BeamEnd
dev2 *(slow)*	Forewarning	BeamEnd
dev3	BeamStart	BeamEnd
dev4	BeamStart	BeamEnd
dev5	BeamStart	BeamEnd

Some device instances

pair entries (logical event, timing event). Once this mapping is done, you can think as if the timing event-source manufactures and fires event-objects that bears logical event names, converted on-the-fly from the incoming event names.

Timing domains. Different parts of the accelerator-complex form different timing domains. The central timing system broadcasts different pieces of information across the timing network to these domains. Each equipment software component deployed on a given front-end computer must register to one or two of them (the latter case is typical of equipment operating on a transfer-line).

Timing controls. In addition to being a source of machine events, the local hardware components of the timing system are programmable and they feature an interface for equipment software to fine-tune delays after which events fire. To this end, the equipment class relies on delegation: at design-stage, the equipment specialist can declare that the equipment class is dependant on the timing control class (see the chapter on composition), at either the device-level or the equipment-software-level. In the former case, the device model must define a field that holds the name of the associated timing equipment. In the latter case, the field in question must be part of the global (class-level) data-store. In both cases, the control of the remote timing device is performed within a real-time action's `execute(RTEvent *)` method. The API for controlling the remote timing object is documented separately by the timing equipment class.

How it works. At initialization, the equipment-software retrieves configuration files (extracted from the data-base) for the front-end computer on which it runs. A first file contains the timing configuration containing a list of timing domains as well as the binding between logical events and low-level timing events. A second file contains the set of device installed on the front-end computer. After initialization, the scheduling map converts the list of logical-event keys by the corresponding low-level timing event keys. Hence, the specified scheduling scheme is translated for the actual timing context.

Recommendations. Before testing a new equipment software on the machine, performing some tests with a simulated event-source can be useful. This makes for a controlled environment within which the equipment specialist can stimulate at will its equipment with a variety of timing situations.

17 Equipment Access

From a control-room computer, an equipment class is accessed across the controls-middleware via device handles through a narrow interface. On a front-end-computer, it is possible to link an equipment's interface library and access an equipment class in a similar fashion.

Equipment software is realized by a set of binary components: the equipment interface (or server) and the real-time task. The two can be deployed in separate processes communicating through a third, shared-memory component. A front-end C++ application may link against an equipment interface.

Equipment access. Each specific equipment software `MyEquipment` features a concrete class `MyEquipmentInterface`, inheriting from a base-class of the framework, `AbstractEquipmentInterface`, and which is responsible for providing access to the equipment's properties. The C++ implementation of this class is automatically generated from the equipment's design. All equipment-access requests emanating from a remote or local client end-up as calls to this `MyEquipmentInterface` class.

Local access (C++). In order to access an equipment from within a local front-end application written in C++ you have to link your program against the server-library component of this equipment. Your code first needs to obtain a reference on the `MyEquipmentInterface` object. Once you get this interface, you may require any device instance or property of the equipment-class from it. Sample code is given on the next page.

Remote access (C++ & Java). When a client program accesses an equipment-software through the controls-middleware, it does it over a device handle which is provided by the `RDAService`. Sample code is given on the next page. The controls-middleware ensures marshalling of the request across the network. When it reaches the FEC, the request is processed by relying on the `MyEquipmentInterface` that represents the class, in a fashion similar to the one described above.

How it works. The controls-middleware's server may be linked against one or several FESA classes. Client programs issue requests on a specific device-name. When the middleware server receives the request, it invokes a method of the `AbstractEquipmentInterface` which returns a reference to the concrete `MyEquipmentInterface` that manages the device. Then the controls middleware retrieves both the request's device and property, and invokes the get or set method on it. Hence, there is no much difference whether your equipment is accessed locally or remotely as illustrated by the two types of code-fragments shown on the next page. However, going through the middleware supports subscriptions, which is not the case otherwise.

To-do list for accessing an equipment locally

✓ *Link your C++ application against the respective FESA server libraries of one or several equipment classes you need to access from within the application.*

In the C++ application:

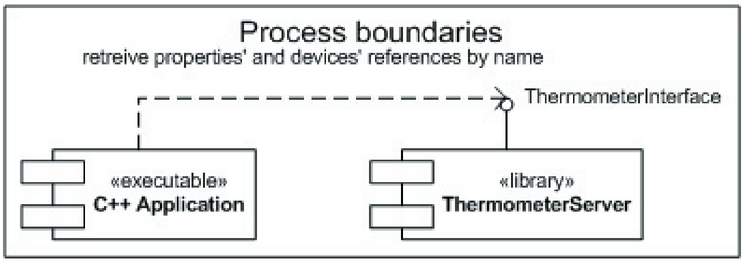
✓ *instantiate the specific equipment-interface objects for each equipment class.*

✓ *Retreive properties and devices.*

✓ *Invoke `get`/`set` methods on these interfaces.*

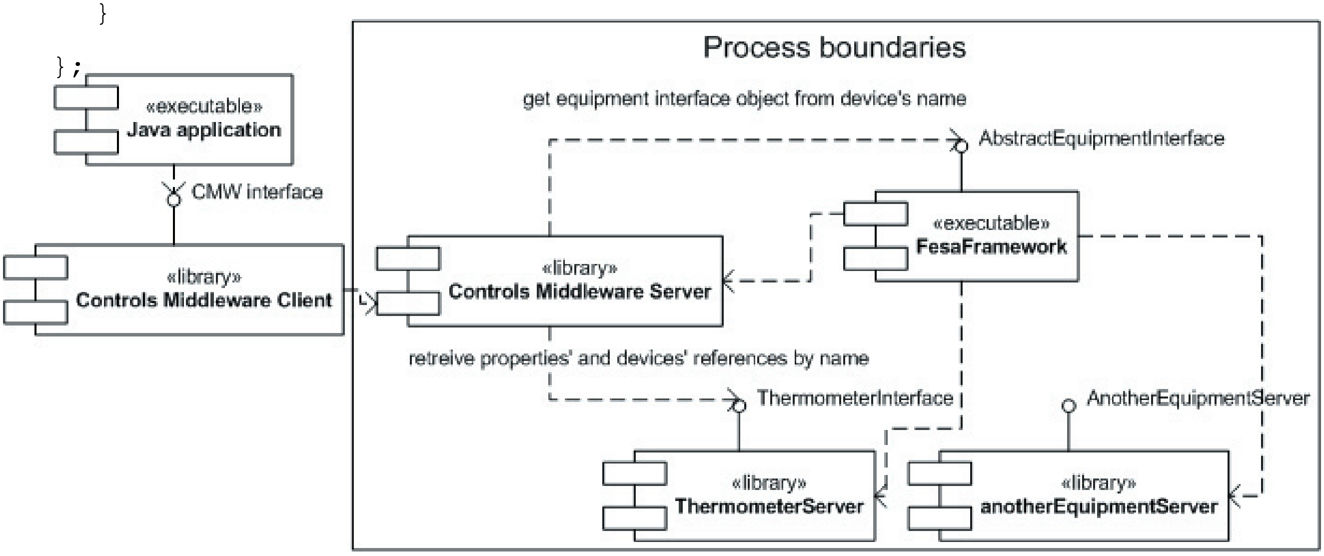
Local access from a C++ application running on the same front-end computer

```
#include <ThermometerEquipmentInterface.h>

int main(int argc, char ** argv) {

  ThermometerEquipmentInterface
     thermometerInterface(«Thermometer»);

  thermometerInterface.init(type, argc, argv);

  string deviceName(«thermometer1»);

  Property * pProp;

  Device* dev = thermometerInterface.getDevice(deviceName);

  RequestMultiplexingContext ctx(«CPS.USER.SFTPRO»);

  pProp = thermometerInterface.getProperty(«Temperature»);
  rdaData data1;
  pProp->get(dev, ctx, &f, &data1);
  float temperature=data1.extract(«temperature»);
};
```



Remote access from a Java application running in a control room computer

```
  import cern.cmw.rda.client.*;
  import cern.cmw.*;

  public class get

  {
    public static

    void main(String[] args){

       String deviceName=«thermometer1»;
       String property=«Temperature»;
       String cycle=«CPS.USER.SFTPRO»;

       RDAService rda=RDAService.init();
       DeviceHandle device=rda.getDeviceHandle(deviceName);

       Data temperature=device.get(property,cycle);
    }

  };
```



18 Alarms

The purpose of alarms is to notify interested parties of faults, which are detected by an equipment-software component, so that corrective action can be taken according to the priority of the fault. To this end, FESA relies on dedicated fields and properties for equipment-specialists to raise alarms to the LASER system.

Alarms-support takes the form of dedicated alarm properties and associated fault-fields. This means that alarms are dealt-with in the same way one deals with standard properties and fields.

Alarms. During the course of execution of an equipment-software program, several faults may occur. For instance, some hardware component may fail, some parameters may leave their allowed range, or the software may raise some exception. In such cases, you don't know or don't want to handle the situation within your equipment-software code and need to transfer the responsibility of deciding how to cope with or remedy what you observe to the people in charge of operating and monitoring the accelerator. This means that raising alarms is different in purpose than logging. When you raise an alarm you want to communicate its description to operators of the accelerator, equipment specialists or any other party that is responsible for correcting and responding to the fault state. Hence, you must make sure that the list of faults is well understood and accepted by them before-hand.

Alarms to-do list

✓ *With the design-tool, add the Alarm properties.*

✓ *In your design's data-model, register a fault-field for each fault (hardware failure, harmful operating point...) your equipment notifies.*

✓ *Configure actions that trigger the Alarm properties by making sure they reference them, as usual.*

✓ *Ensure that all interested parties approve your alarm model, and provide in collaboration with them complementary pieces of information.*

✓ *In C++ code raise alarms by setting fault-fields to true. Lower them by setting-fault fields to false.*

Alarm system. FESA provides a front-end layer to the underlying LASER alarm-system, which defines its own protocol and API. Although this FESA layer insulates you from dealing directly with the LASER interface, you need to be aware about how alarms are transmitted and processed.

Alarm properties. You can introduce alarm properties into your equipment model in the same way as regular properties. Every alarm property automatically retrieves the state of its associated fault-field together with the fault time-stamp. Alarm properties are notified by server actions and real-time actions and the association between alarm properties and actions needs to be specified in the device design.

Fault-fields. You specify possible faults as dedicated fault-fields in your design. As the other FESA fields, fault-fields may be multiplexed in case you want to restrict a fault to a particular operating context of the accelerator. The LASER API identifies all faults by the fault triplet: the default mapping is as follows: (1) your equipment's class name stands for fault family (FF), (2) the device-name stands for fault member (FM); and a descriptive text field must be supplied by you when defining the device class.

Fault-invalidation of regular properties.

The fact that some fault-field is set “on” may indicate that the device is not properly functioning - the consequence of this is that using normal properties (which means reading or writing) may not be reliable. You can declare at the design stage that a given property is conditional to some selected set of faults. FESA automatically checks whether any of the related fault-fields is in the “set” state and will throw an exception, instead of allowing you to set or get a value that may be unreliable or even have no meaning at all.

Time stamping. If the UTC time is available in the equipment, the fault-fields are stamped with the time the fault state is generated and this time is communicated as part of the whole fault information to the Alarm Monitor. The Alarm Monitor uses this timestamp as the ‘LASER user timestamp’ when the fault state is sent to LASER.

How it works. A component called the Alarm Monitor subscribes to your equipment’s Alarm property and is notified automatically when the fault situation changes. All information concerning the fault is then assembled to call the LASER source API, which transmits the fault to LASER. Using this information, the Alarm Monitor maintains a list of all faults currently active, an “active list”, for the devices it is responsible for. Additionally, the Alarm Monitor makes periodic calls to the LASER subsystem to ensure that it operates with the updated alarm information.

Recommendations. Fault fields contain a fault-state which is entirely controlled from your C++ code. Hence if you raise an alarm, it will stay-on until you explicitly reset the fault-field. Therefore you must make sure that for each ‘raise’ statement, you have a ‘lower’ counterpart, as illustrated by the example depicted.

Sample code which raises alarms

```
// One fault-field named «overheat», associated
//  to a Temperature property for diagnostics.
AcquireTemperature::execute(RTEvent *pEv) {
  for (unsigned int i=0;i<deviceCollection.size();
     i++){
     ThermometerDevice* pDev=deviceCollection[i];
     AcqBoard* board= getAcqBoard(pDev->hw.get());
     float temperature=board.getSample();
     if (temperature>pDev->maxTemperature.get()) {
          pDev->overheat.raise(); // raise alarm
     } else {
          pDev->overheat.lower(); // no alarm
     }
     pDev->temperature.set(temperature);
}
```

Example of context-dependant alarms

```
// Two fault-fields «badVoltageRef» «regulationFail»

setVoltage::execute(RequestEvent *pEv) {
  MultiplexingContext * pContext =
     pEv->getMultiplexingContext();
  voltageRef = value.voltage;
  bool goodSettings =
           (voltageRef < pDev->maxVoltage.get())
     &&    (voltageRef > pDev->minVoltage.get());
  if (goodSettings) {
     pWorkingDevice->
          refVoltage.set(voltageRef,pContext);
  } else {
     pWorkingDevice->
          badVoltageRef.raise(pContext);

     // the equipment copes with the situation
     // by maintaining current settings and
     // ignoring new ones, yet alarm is
     // registered and will stay-on until new
     // valid settings are applied by upper layer
     // to whom exception is returned meanwhile.
     throw FesaIOException(«out of range»);
  }
}

AcquireVoltage::execute(RTEvent *pEv) {
  MultiplexingContext * pContext =
     pEv->getMultiplexingContext();
  for (unsigned int i=0;i<deviceCollection.size();
     i++){
     CapacitorDevice* pDev=deviceCollection[i];
     VBoard* pBoard= getVBoard(pDev->hw.get());
     float voltage = pBoard->getVoltage();
     bool badVoltage = (fabs(voltage - pDev->
          refVoltage.get(pContext))>TOLERANCE);
     if (badVoltage) {
          pDev->regulationFail.raise(pContext);
     }
  }
}
```