Designing the SRT control software: Notes to the UML schemes

Andrea Orlati[1]
Simona Righini[2]

1 - I.N.A.F. Istituto di Radioastronomia.
2 – Dip. Astronomia - Università degli Studi di Bologna.

Dicembre 2008

IRA 423/08

Introduction

This document is meant to be a quick reference for readers and a memorandum for the developers about the symbols and conventions used to draw the diagrams modelling the server part of the ESCS(Medicina VLBI Antenna)/NURAGHE(SRT) system. The client or user interface parts are not included. A good comprehension of these pages depends on a good familiarity with terminology and concepts of UML and ACS framework, in particular with its component/container model.

The Component Diagram.

The component diagram shows the components that are involved in the system, it also reports the artefacts that are included in the system like files, libraries and database tables. This diagram is the first that should be drawn since it can provide a quick overview of the system itself and a snapshot of the relationships between the components.

Of course a lot of details are missing but this is not a limitation, since they can be provided in other diagrams. For example, the attributes or the methods that each component exposes can be described in class diagrams while deployment information of each instance can be provided in deployment diagrams.

Subsystem: Components and table artefacts are grouped into subsystems that correspond to a single functionality unit. A subsystem is a container of components and artefacts that belong to the same sub-part of the whole system. The subsystem is generally drawn in the upper left corner of the scheme (figure 1); the name of the scheme should reflect the name of the subsystem itself.

Component: The component corresponds to the ACS/CORBA component/remote object. Each component can implement a specific functionality that is required to realize the system. No information about the instances is provided at this stage. Implicitly all the components drawn(figure 2) in a scheme are part of the subsystem that has the same name of the scheme. Components have relationships between each other and/or they can implement or realize interfaces.

Interface: An interface defines a common way for components to access other components or for clients to call components through common patterns. Interfaces allow to group components that show the same services but differ in the way the services are implemented or executed. Since the aim of this diagram is to provide a generic overview of the system no details - apart from the name of the interface - are required.

Artefacts: These diagram objects can be files, libraries or database tables. For example the CDB tables can be drawn as a grid (figure 5) in component diagrams if one or more components depend on them for some reasons. Since artefacts are peculiar of components diagram and do not appear anywhere else, more details can be given using a note.

Relations: The components diagram allows for two kind of relations:

- generalization: it describes exactly the concept of “realize” or “implements”. It can be drawn between an interface (destination) and a component (source). The symbol for generalisation is a solid line with a solid arrowhead (figure 6).
- dependence: a component depends on another when it needs some services from it. It is drawn via a dotted line with an open arrowhead (figure 6).

The Class Diagram.

The control software under development is based on the ACS platform; this platform brings the following constraints: distributed object and component-server paradigma. Moreover, a certain flexibility comes from the fact that the schemes are not required to automatically generate the code. All these considerations led us to draw class diagrams of the Unified Modelling language (UML) using a sub-set of the formalism and defining, at the same time, non-standard meanings for standard symbols.

If we adopt *component*, *server* or *remote object* as meanings of the word *class*, the class diagram allows us to model the system by describing all the objects in it, specifying their characteristics and properties - and moreover the relationships among them.

This diagram describes very well all the CORBA interfaces that will be included in the system, in other words it is the graphical illustration of the IDL (Interface Definition Languages) files required to develop the system using ACS.

The adoption of ACS framework brings also some advantages for the system design:

- every attribute can be archived (i.e. it is sampled with a given frequency and stored in the archiving system);
- every attribute can be monitored and linked to the alarm system (i.e. it is possible to set limits and if the value of the attribute goes beyond these thresholds an alarm is triggered and propagated to the user);

1. every event in the system is transformed into a log message and then sent to a centralized logger so that everything happening during the observation is known and stored for later analysis.

Since all of these are ACS built-in services they are not drawn in the UML schemes because they are available in any case.

The list of objects that may appear in a class diagram is now provided:

Package: It is a container of an amount of elements that constitute a sub-set of the system or that stay logically or functionally together. All the elements belonging to a package are drawn in a scheme with the same name of the package. The package symbol (figure 7) usually appears on the upper left corner of the scheme and it is a sort of tag of the scheme itself.

The whole project is package-based; this means the project must be divided into groups or working unit. Every working unit has its own package that describes it.

In the implementation process of the project a package maps into an IDL module.

Elements: They are represented as red-bordered boxes (figure 8) filled with yellow ink if they are public (belonging to the current package) or with blue ink in case they are published (imported by other packages). Data types, interfaces, enumerations and classes are elements.

Data type: It adds a new data type to the system. The type name should be self-explanatory, otherwise a note (see below) is attached. It is drawn (figure 9) as a normal element in which it is specified the stereotype <<*datatype*>>.

Interface: Interfaces can group classes according to their semantic and functionality. They allow classes to access other classes that inherit from an interface through common methods and/or attributes. Interfaces are drawn (figure 9) like classes but the stereotype

<<*interface*>> is prefixed. Since in standard UML attributes for interfaces are not allowed, a normal class is drawn as “contained” by the interface (see also containment relationship). In that class all attributes and methods of the interface are drawn.

Enumeration: Exactly as the data type it adds a new type to the system. In this case the new type is a fixed-values type.

Class: it represents an object/component/server of the system and it is ploted according to figure 11. Every class has its own attributes and operations. In case a class is virtual (the object cannot be physically present in the system but it only indicates the characteristics that an object must have to "play the role") the definition of the class is enlarged by the containment of an interface element (see also containment relationship and interfaces).

If an object is linked against a specific hardware or device, which means that a specific implementation (one for each telescope) must probably be given, it is represented by a green-bordered box.

There may be the case in which more than one component will be required by the system. This is represented by an extra box drawn in the upper-right corner of the class itself. This box will contain an attribute that could be considered the “search key” of the class: it conveys information on how the objects belonging to that class can be distinguished one from the other. If the particular instantiation is not important, i.e. if any component of that type does the required job, then the search key is replaced by the character *. This last item can be mapped into an ACS dynamic component (it has no instantiation in the configuration database and it can expose only operations, not attributes). The class box can also expose some items according to the following list:

- attribute: it represents a significant value that can be read from the server, for example the temperature of the meteorological sensors. It is also possible to provide for the single attribute a multiplicity which allows to represent attributes as lists, sequences or vectors. In that case a * represents that the number of elements is not fixed or not known in advance. If an attribute is static, it is drawn underlined which means it is immediately available to the whole system independently from the particular package. This feature is mapped into the ACS notification channel. Every attribute is given together with its own data type and visibility scope:

  - public: it's indicated by a + as a prefix. It means that the attribute is available for external consultation, for example by a client or a GUI;
  - private: it's indicated by a - as a prefix. It means that the attribute is not visible externally and it is just for internal usage;
  - protected: it's indicated by a # as a prefix. It means that the attribute is not directly visible externally but can be accessed via an operation (accessor).
- operation: an operation is an action or a service that the object is able to perform. There are two kinds of operations: synchronous operations, that usually take a short time to complete, and the asynchronous ones that can run for several minutes. A synchronous operation returns immediately to the caller while the asynchronous ones (drawn in bold font) return exploiting the callback mechanism. An operation can have a varying number of parameters. Every parameter has its own data type. If the parameter is an output parameter the word *'out'* will be added before. Like attributes, operations have their return type and visibility scope:
  - public: it's indicated by a + as a prefix. It means that the operation is available for external usage;
  - private: it's indicated by a - as a prefix. It means that the operation is not visible externally and it is just for internal usage;

Note: it is a box containing simple text giving details about the element the note is attached to.

Relations: The relations are structural connections between the elements of the diagram. The representation of the relationship between the elements is the foundation of class diagrams. We take into consideration 5 kinds of relations:

- Composition: if a data type of an attribute is user-defined (not basic) a clever way to represent it is by means of a composition. The concept is based on the typical whole/part relationship. Of course cycling relations are not allowed and the 'whole' is directly responsible of creating and destroying the 'part'. Compositions are drawn with a solid diamond at the tail end. This relation is also integrated by the addition of the name of the property and its scope and multiplicity.
- Containment: This relationship reflects the fact that the destination is contained by the source, so the destination enlarges the definition of the source. Typically

this relation is used to enlarge the concept of interface so that also attributes can be provided. This relation is represented by a solid line with a circle in the tail end (source).

- Generalisation: This relationship describes exactly the concept of inheritance in an object-oriented paradigma. The foundation of this relation is the idea of "substitution", that is the source can substitute the destination in its features. The symbol for generalisation is a solid line with a solid arrowhead.
- Association: If an object has another object or class as a member (public or not), the relation is represented by a solid line with an open arrowhead. This relationship is somehow weaker than the composition in the sense that the destination could itself survive independently from the source. In theory it allows for cycling relationship. Moreover an association permits to give a multiplicity in order to specify how many objects take part to it:

  1 exactly one element takes part

  0..1 the element is optional

  1..* one or many elements can take part

- Dependence: If an element depends on another one, this constitutes a relation which is drawn via dotted line with an open arrowhead. The element having the dependence is called “client”, while the other one is the “supplier” (the one indicated by the arrow). This relationship also indicates that a change in the “supplier” will affect also the “client”. The relation must also be detailed with one of the following stereotypes:
  1) <<*call*>> *[Dereference::method]* the source can call a method of the destination;
  2) <<*use*>> *[Dereference::class]* the source requires the destination for its implementation, a sort of master/slave relationship;
  3) <<*initialize*>> the source will initialize the destination.
  4) <<*get*>> *[Dereference::attribute*] the source makes use of an attribute of the destination

The Deployment Diagram.

The deployment diagram shows the instances of each component as they will participate to the system. This diagram will give a very close description of the deployment part of the CDB and consists in nodes and component instances.

Node: In standard UML a node is everything that can host some software, in two forms: devices and execution environments. In our case execution environments are containers and devices are the pieces of hardware that are directly connected/controlled to/by the system. The nodes are connected by communication paths that show how the various nodes are linked together (solid line). Execution environments are represented by a cube surrounded by a box (figure 13), everything that is included in the box or listed in the node will be executed/deployed in the node itself. Devices are drawn in green ink to highlight their hardware nature (figure 14).

Components: In this diagram they represent ACS component instances. They are usually drawn inside nodes to represent the fact they will be executed by the specified component. The name of the item is the name of the instance plus the name of the component type, separated by the symbol of dereference. If a * is used in place of a normal name it means that the component can appear in the system an undefined number of times, in ACS this will result in a dynamic component. In case the number of instances is more than one and well defined, a ring with a dotted line with an open arrowhead is added listing all the instances. All components enlisted in deployment diagrams will have an entry in the CDB, so this kind of diagram is a very close representation of the component deployment part of the CDB itself. For example a component called *ANTENNA/Mount:MedicinaMount* will result in the following entry in the CDB:

$ACS_CDB/CDB/MACI/Components/ANTENNA/Mount/Mount.xml.

Appendix A: The component diagram

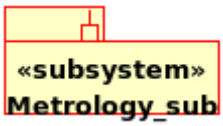

Figure 1 - The subsystem

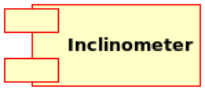

Figure 2 - The component

Figure 3 - The Interface

Figure 4 - The file artefact

Figure 5 - The table artefact

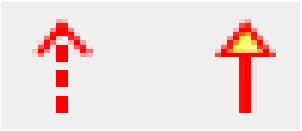

Figure 6 - The dependence and generalization relationships

Appendix B: the class diagram

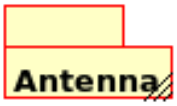

Figure 7 - The package

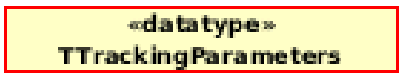

Figure 8 - The data type element

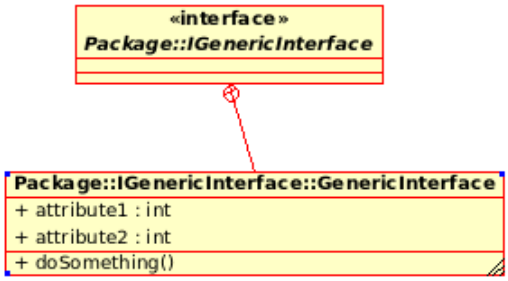

Figure 9 - The interface

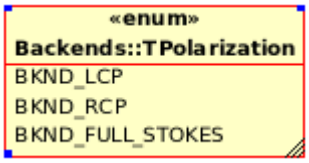

Figure 10 - The enumeration data type

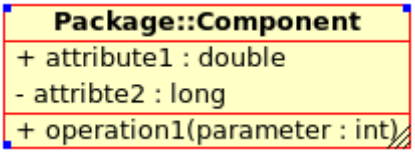

Figure 11 - The class

Figure 12 - The class diagram relationship, association, dependence, generalization, containment, composition

Appendix B: the deployment diagram

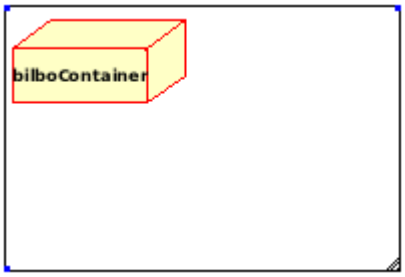

Figure 13 - The node representing the execution environment of the container

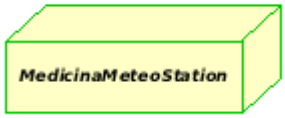

Figure 14 - The node representing hardware devices

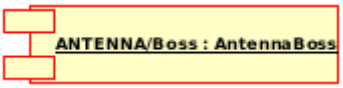

Figure 15 - The component

References

"The Unified Modelling Language User Guide" - Grady Booch, James RumBaugh, Ivar Jacobson - Addison-Wesley

"UML Distilled" - Martin Fowler - Addison Wesley

http://www.eso.org/~almamgr/AlmaAcs/