DOI: 10.13009/EUCASS2022-4691
9TH EUROPEAN CONFERENCE FOR AERONAUTICS AND SPACE SCIENCES (EUCASS)

An innovative architecture for multi-mission Flight Dynamics and satellite programming software

Julien Anxionnat and Véronique Maurussane***
** CNES — Attitude Guidance & Mission Planning Dpt.*
julien.anxionnat@cnes.fr
*** Thales Services Numériques — Critical Information Systems - Software Engineering*
veronique.maurussane@thalesgroup.com

Abstract
This paper browses through a set of software components whose defined and well isolated roles allow us to easily take into account, during the design of the FDS for a new mission, "upstream" requirements (the technical requirement specification) and "downstream" requirements (the interfaces specification), while achieving the definition of roles that can be fully understood by the operational teams; and details how each of these components was designed to fulfill the objective of maximizing reusability by isolating the specific elements as closely as possible.

1. Introduction

A "**Flight Dynamics Software**" (FDS) is a part of a "Ground Control Segment" for a space system. This software is mainly in charge of managing the satellite's orbit; its primary functions are: orbit restitution, orbit control maneuvers computation, and programming of the satellite platform and attitude guidance.
The CNES SIRIUS in-house project team designs and develops FDS for several missions and aims at building a reusable architecture applicable to all of them.

The used methodology is based upon a "top-down analysis", started by defining the main and general "use cases" of the system and by identifying the spectrum of all edge cases of the aimed missions.
Starting with the "use cases", since the targeted programming language is Java, an object-oriented one, our approach is entirely using the UML modeling language with a Domain Specific Language instantiation. All datatypes are described as classes, and all use cases are realized by components.
The components are organized in several tidy thematic domains, "guidance and programming" being the one we are going to browse.

"**Guidance**" represents the orders loaded onboard for the AOCS and describing the attitude (and, in some cases, the orientation of mobile parts) to follow. These orders may be of various forms: from Fourier series decomposition of the angular velocity to a simple switch for an autonomous satellite. The word "guidance" also depicts all the methods and algorithms put into play to compute these orders.
"**Satellite programming**" is the field of scheduling and planning all activities to be done on/by the satellite (guidance-related or not: parameters updates, thrusters preheating, …), and then producing the data useful and necessary to prepare the telecommands. First, we have to define the functions called "guidance and programming", before exploring the several specificities relating to every satellite and considering the variable use cases for programming requirements among all missions.

The satellite "guidance and programming" software has to handle two, and maybe three, main objectives, which are the following **features and main functions**:

- computing and providing the data interface named "functional requests" containing all the information needed to produce the telecommands to send to the satellite;
- maintaining a prediction of the future spacecraft attitude, either commanded from ground or eventually generated onboard by autonomous capabilities;
- computing and booking of all time slots to reserve for some activities.

DOI: 10.13009/EUCASS2022-4691

2. Requirements

Whatever the mission is, the designer of a FDS has two main sources of specifications:

- the technical requirement specification,
- the interfaces specification.

The first source defines the functional needs applicable to the FDS. These needs are about *what* the FDS shall be able to have the satellite "physically" done (what type of activity, what guidance/pointing, what state change, at what date, according to which constraints).
The second source is the interface specification, which defines how the commands shall be handed over to the control center, *i.e.* the format they should comply with. The resulting file is called "functional requests", it's a precursor of the telecommands finally sent to the satellite after encoding.

Typology of "upstream" requirements

The use cases for programming requirements among all missions are variable. All aspects of a mission may have an incidence on these programming requirements:

- heliosynchronous orbits have a periodic rhythm of eclipses;
- the types of the orbit control maneuvers may result in many ways to produce a sequence;
- some payload may need some specific calibration sequences;
- any onboard subsystem could require parameters loading and updating;
- some satellites are autonomous, others are not;
- the operations' sequences often differ between "begin of life" LEOP operations and "routine and daily" activities;
- the satellite's pointings might be different.

So these use cases are mission-dependent, or "mission-related". We will consider them as "upstream" requirements.

Functional requests, as "downstream" requirements

"Functional requests" are an attempt for an onboard/ground interface standard. Each functional request type addresses one function ("select a given guidance law", "turn on the thrusters", …) and defines the required parameters for it. They are what a FDS produces and provides for the telecommands preparation and sending.

Some functional requests have been defined in order to be reusable and reused, but despite this effort, they are often defined differently from one satellite to another. Their definition depends on the onboard software architecture and the platform operability:

- the way to program an orbit control maneuver varies: "start date and delta-V", "median date and number of pulses", …
- many data may be required before starting an orbit control maneuver,
- any onboard equipment should need some parameter update.

The defined "functional requests" are what we could call the "downstream" requirements.

Constraints

A satellite's programming must comply with certain constraints, which may vary from one satellite to another:

- some optical instruments must be not dazzled by the Sun or the Moon;
- the allowed kinetic momentum or torque should not be exceeded;
- any onboard parameter may have minimum or maximum values;
- some activities should be scheduled only in eclipse;
- the programming by a Mission Center might be taken into account.

There are almost as many constraints as use cases, each mission having its own constraints.

The validity of a programming plan is an important concern regarding the system, as the violation of a constraint can lead at best to a temporary unavailability of the mission, at worst it can damage some equipment on the satellite.

DOI: 10.13009/EUCASS2022-4691

Therefore, checking the constraints after programming requests are computed is a function in its own right in the FDS. This guarantees the full robustness of the programmation provided to the control center, even if the constraints are already integrated beforehand to compute the parameters of the activities to program.

Integration into operations sequence

A classic operational sequence uses several components of a FDS in an established order:

- (External) data consuming,
- Orbit restitution,
- Maneuver computation,
- Programming,
- (External) data producing.

The purpose of the programming components is to take into account the freshly restituted orbit, to program the execution of any incoming orbit correction maneuvers on the programming horizon (or other attitude-related operation such as inertias' calibration or calibration of a facility) along with the corresponding attitude guidance, and any on-board configuration updates.

Multi-mission design

As already mentioned, in summary, the challenge of a multi-mission design is to take into account all possible specificities, whether they are upstream (*i.e.* mission use cases) or downstream (*i.e.* onboard/ground interface requirements).

3. Data structures & *metamodel*

Before exploring the architecture of the "guidance and programming" software, it is mandatory to introduce some useful notions and especially the "metamodel" used to model the components.

Database

All components of a FDS software get their parameters and the data to process from a single database, a "Semantic Graph Database".

Each data in the database is of a complex datatype, a class in UML. A datatype may be structured and may gather "sub-data", each one of them often being itself of a complex datatype.

In this database, the relationship of a datatype towards an attribute may be "association" or "composition". A "composition" relationship defines an attribute, which is *contained*, which lives in the data. The "association" relationships define attributes, which are outside the data, which are stored by themselves elsewhere in the database, and are just *referenced* in the data.

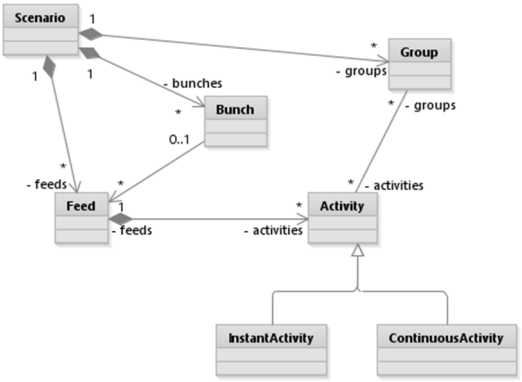
Thereby, an element in the database is of a given type. This type defines the attributes (which are of given types) which may be contained or referenced.

Datatypes

The main datatype (and by the way the more complex) is in any case the `Scenario`. The idea of what we named a `Scenario` is to describe the satellite's state evolution along time. There is at least one `Scenario` in the database: the "nominal" one.

A `Scenario` is composed of `Feed`s. A `Feed` is a generic timeline dedicated to one thematic from one point of view, and for one satellite. For example, an `AttitudeFeed` describes an attitude sequence, an `AOCSManagementFeed` describes AOCS modes' switches and parameters' updates. An `AttitudeFeed` may depict the attitude as programmed or "by default". A `Feed` is composed of `Activity` elements: an `Activity` may be, for example, an attitude pointing during a time interval, or an instant event such as a parameter update.

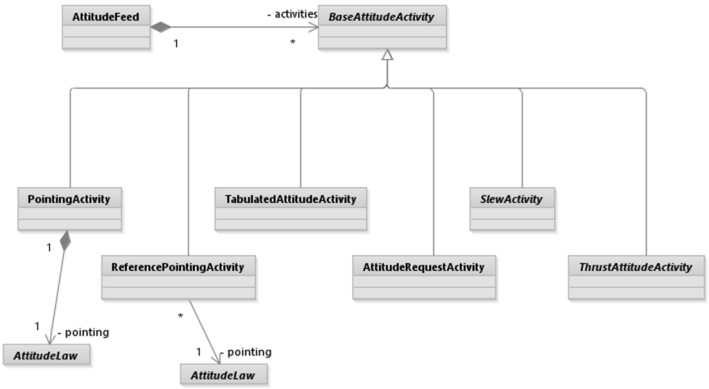
DOI: 10.13009/EUCASS2022-4691



Scenario — UML generic class diagram

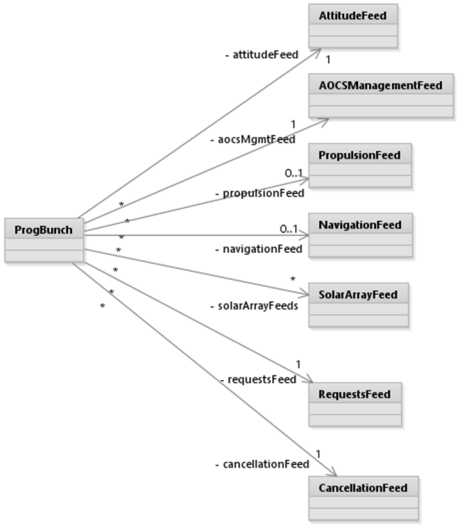
Originally, this has been designed to represent the “programming plan”, and then expanded to many other topics: there are also `TrajectoryFeed`s, `ManeuversFeed`s, `TankStateFeed`s, …

Each type of `Feed` contains activities of specific types as for `AttitudeFeed`:



AttitudeFeed — UML class diagram

The `Feed`s in a `Scenario` are organized in `Bunch`es. The “programming plan” is one of them: `progPlan:ProgBunch`; it is composed of several `Feed`s:



ProgBunch — UML class diagram

DOI: 10.13009/EUCASS2022-4691

As we will see, this “programming plan” is the receptacle of the “functional requests” whole generation.

A `Scenario` has `Group`s. A `Group` enables the capability to gather some `Activit`ies. The relationship between a `Group` and its activities is bidirectional: thereby an `Activity` can access to the others. That concept is very useful to maintain a consistency and some “sense”.

Another important element is that in database there may be some data that are lists of references on `Activit`ies laid in a `Scenario`.

The classic graphical representation of a `Scenario` and some of its `Feed`s is the following figure:

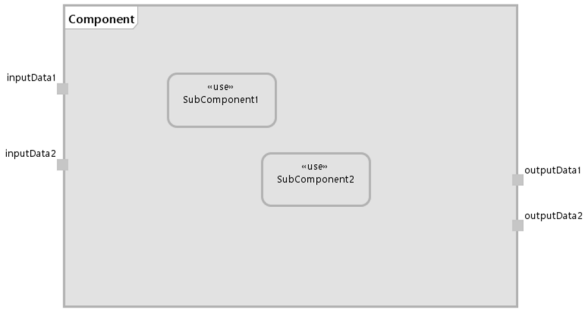
Scenario & programming plan

In that figure, it can be noticed (and reminded) that the `SCENARIO` is one data (of `Scenario` datatype) in the database. The dots are “instant activities” and the rectangles are “continuous activities”.

The notion of “component”

A component has a “written” functional role, some inputs, and some outputs. Each input or output data is of a precise datatype, and should be bound from the database. A component may also define usages of other sub-components.

A component realizes a general and primary use case. We could say that a component leads to the definition of an interface (in object-oriented sense).



Component — metamodel

DOI: 10.13009/EUCASS2022-4691

Each component definition leads to one or more implementations. An implementation of a component offers an algorithm, a way, a manner to fulfill the role of the component itself. An implementation, in addition to the input and output data defined by the component, defines its own parameters, used by the algorithm.

All the subtleties of the design is to decorrelate an implementation from the multiple implementations of the used sub-components. Later, each application is assembled from the choice, for a given mission, of the component implementation and the implementations of the called sub-components.

4. A partitioned functional architecture

The idea here is to show, by going through the sequence of the programming generation why and how it was cut into different steps, into different components.

Use case: “Generate the programming plan and produce the functional requests”

The entries are called “instructions”

All demands to be programmed on a given horizon are prepared by other components. All these instructions are of some types of `Activit`ies, laid in the used `Scenario`.

There are two main types of activities to program:
- the orbit control *maneuvers*,
- and a variety of possible needs to carry out any other operation (for example a calibration, a satellite mode switch), which we call *instructions*.

The maneuvers to command may also be of different forms depending on the propulsive system, which comes out as derived types of an abstract common orbital maneuver type.

There is a large variety of these instructions from one mission to another. The instructions may require various implementations:
- update a given set of parameters onboard,
- load new EOP data,
- run a sequence of several pointings in order to calibrate inertia,
- exit the “safe mode”,
- warm start an equipment such as GNSS,
- …

Each maneuver/instruction type carries the specificities and a use case of the mission. Even if all datatypes are available, only a few concern the mission. The maneuvers and instructions carry what we called the “upstream” requirements.

Since a `Scenario` reflects the past (*i.e.* contains the history), not all the maneuvers or instructions have to be programmed. Those to process are referenced in a list in the database: `PROG_ENTRIES`. This list references all the maneuvers and instructions to program, and has to be completed by their producing components.

DOI: 10.13009/EUCASS2022-4691

`PROG_ENTRIES` references list

Two levels of representation

The need to comply with the two sources of requirements lead us to partition our architecture in two levels of representation, for both data and functionalities.

The data are structured as follows:

- a “physical” representation of *what* the FDS-commanded satellite activity shall be;
- a “commanded” representation, *i.e.* the functional requests (the “*how*”).

The two levels are linked between them to help processing along and ensure a history of FDS programming.

To produce these two levels of representations, the functional architecture is also partitioned in two main programming functions:

- Prepare a “programming plan” (produces the “physical” representation) from information provided by other domains (mainly orbital maneuvers computation) or from mission specific instructions.
- Compute the functional requests (“commanded” representation) from the “physical” representation taken from the programming plan.

Planning and scheduling — `ProgPlanner` component

The first step to generate the “programming plan” and produce the “functional requests” is the preparation by planning and scheduling. This preparation is about the comprehension and the planning of all given `PROG_ENTRIES` (reminder: the maneuvers and instructions). Each activity referenced in `PROG_ENTRIES` is processed and converted into all necessary activities on the `progPlan`.

That step of the sequence is realized by the `ProgPlanner` component. This component fulfills several roles:

- **Hierarchization of the entries**. The set of programming instructions, carried by the `PROG_ENTRIES` list, is prioritized: each instruction is assigned a priority level (this allows, for example, to give priority to the realization of an orbit correction maneuver over another instruction considered less important). This prioritization is not hard-coded, but is described by parameters.
- **Planning**. Following that priority order, each programming instruction is processed and broken down into as many on-board activities as necessary. For example, still in the case of an orbit correction maneuver, the following are planned: modes switches, pointing changes, preheating and configuration updates of the propulsion system, and the thrust itself.

DOI: 10.13009/EUCASS2022-4691

- **Calculation of particular pointings**. Some instructions require the calculation of a pointing or some attitude slews, linked to the orbit and the scheduled date-time.
- And **distribution of activities by onboard subsystem**.

Some examples:

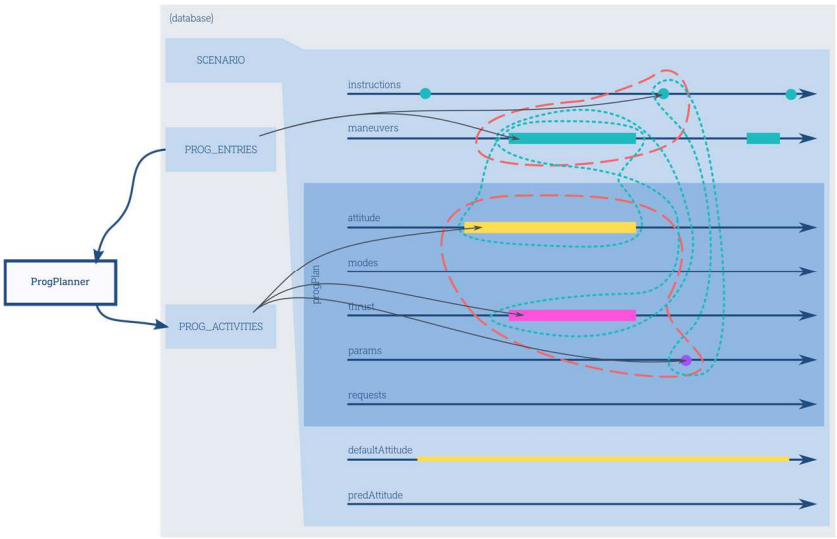
- An instruction to realize a inertia calibration will lead to a sequence of `AttitudeActivities` on the `AttitudeFeed` of the `progPlan`.
- An `InitDateAndDeltaVManeuver` will be processed into a pointing (an `AttitudeActivity`), some AOCS modes switches (activities on the `AOCSManagementFeed` of the `progPlan`), some parameters updates (activities on the `PropulsionFeed`), and an activity to order the thrust start (activity on the same `PropulsionFeed`).

When creating all these activities, `ProgPlanner` puts in place two functional links between activities:

- A first link to signify the cause and effect relationship between the original instruction and each of all the activities carrying it out. It would be useful for the cancellation use case.
- A second link to signify the inseparable character of this set of created activities. It means that one of them cannot be programmed without the others. It is possible that several `PROG_ENTRIES` are also “indissociable”. In this case, all their realizing activities are grouped in the same “indissociable” group.

These two links are formally groups within the scenario. Among other things, they allow the algorithms to immediately identify which activities are to be deleted in case of cancellation, in particular when cancelling the original instruction.

We stick to this level, *i.e.* we do not directly translate these programming instructions into functional requests in order to maximize the potential reuse of the implemented processes and, above all, to realize only what we called the “upstream” requirements.



`PROG_ACTIVITIES` references list

At the end, the `PROG_ENTRIES` list in the database is updated (all processed activities are removed), and all created programming activities are appended to the list called `PROG_ACTIVITIES`.

Requests parameters computation — `Requester` component

The next step to generate the “programming plan” and produce the “functional requests” is the computation of the functional requests parameters. This computation gets the `PROG_ACTIVITIES` (produced at the first step) list as input. All these `PROG_ACTIVITIES` are translated into “functional requests” defined for the mission.

Thus, it translates the programming plan into a format that conforms to what we called previously the “downstream” requirements.

DOI: 10.13009/EUCASS2022-4691

At first glance, one activity in `PROG_ACTIVITIES` should be converted into one “functional request”. For example, an `AttitudeActivity` leads to a guidance request.

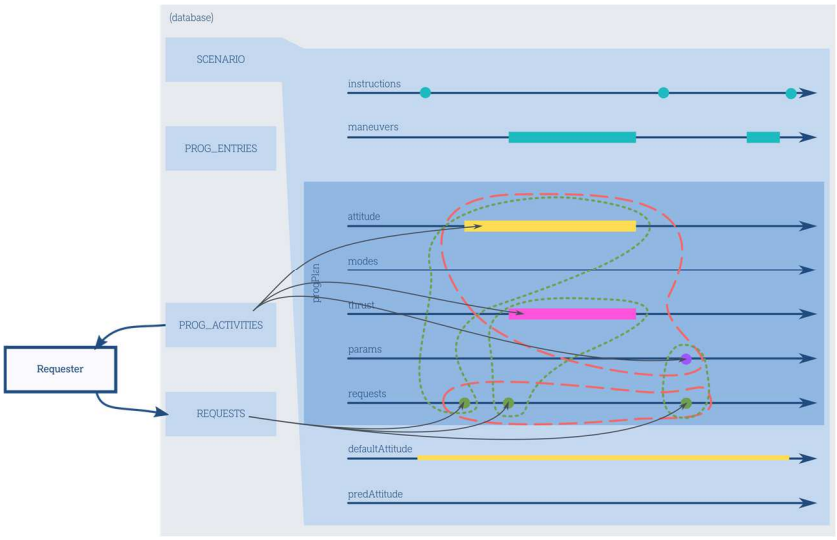
According to the AOCS requirements, the attitude activities, expressed as attitude laws, may be converted into guidance profiles (such as harmonic or polynomial profiles) by that component.

This component sets and maintains the same types of links between activities as `ProgPlanner`:

- Each processed input activity is grouped with the computed request activity, in order to signify the cause and effect relationship between them.
- The inseparability of the input activities is also maintained: all computed requests activities form inseparable input activities are grouped into an “inseparable” group.

In the end, this makes it possible, for example, to find all the requests produced from an instruction, and reciprocally to find what a given request is used for by going back to the original instruction. And this allows to immediately identify all the requests that are inseparable from a request that would later encounter a problem, and cancel them.

As said above, all computed “functional requests” meet the *Interface Requirements*.



`REQUESTS` references list

At the end, the `PROG_ACTIVITIES` list in the database is updated (all processed activities are removed), and all created activities are appended to the list called `REQUESTS`.

Verification and validation — `Checker` component

This step could be optional, but is made necessary in the case of high risks due to bad programming.

The `Checker` component is able to verify, for example, that neither an optical payload nor a stellar sensor is dazzled, or that the angular momentum is within its envelope. The component also validates the respect of the formats, the respect of the chronologies, *i.e.* validates the correct process of the preceding components. Even if, by construction, the programming plan is supposed to be correct (since this is the subject of the software validation performed in the development phase), this component allows for a double contextualized validation and thus allows for compliance at a higher criticality level.

Three types of checks are performed:

- event checks, such as no dazzle;
- mission specific checks, such as respect of chronologies;
- integral checks, such as sufficient solar panel illumination over a given horizon.

For any error detected, the offending request(s) are identified and referenced in the output data and in a list adjacent to the others.

DOI: 10.13009/EUCASS2022-4691

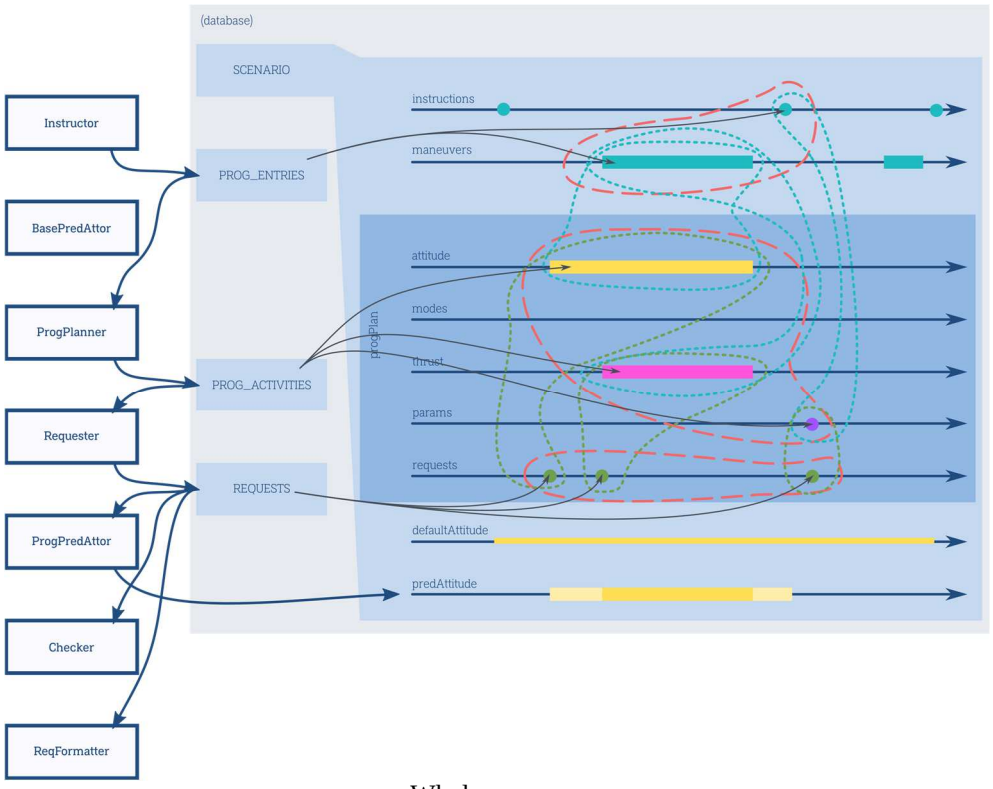
Final formatting of functional requests — `ReqFormatter` component

This last step adds to the thematic parameters of the functional requests all the additional elements necessary for the handling of these requests by the control center (IDs and status).

It is the final step in the generation of the programming plan and the production of the functional requests.

TL;DR

Below is the entire sequence and all the activities, groups and activities references lists involved on a single diagram.



Whole sequence

Use case: “Predict the attitude”

Blend several attitude representations

As seen above, an attitude sequence (pointing segments, slews) is described by an `AttitudeFeed`. Each feed represents the attitude from a given point of view. Thus, several `AttitudeFeed`s coexist in a scenario:

- an attitude feed predicted from the programmed guidance,
- a feed of restituted attitude,
- an attitude feed predicted from the programming of mission and payload activities,
- an autonomous attitude feed (for satellites with autonomous guidance capabilities),
- a default attitude feed.

To give only a directly usable image of the attitude, it is necessary to combine this set of feeds. This is the role of an `AttitudeBlend`: it allows to arrange each of them according to a parameterized order of importance.

When the blend is questioned to obtain the attitude, it addresses the first of its feeds, if it has no information to return an attitude, it passes to the next, and so on.

Thus, each of the feeds needs to be updated as soon as possible.

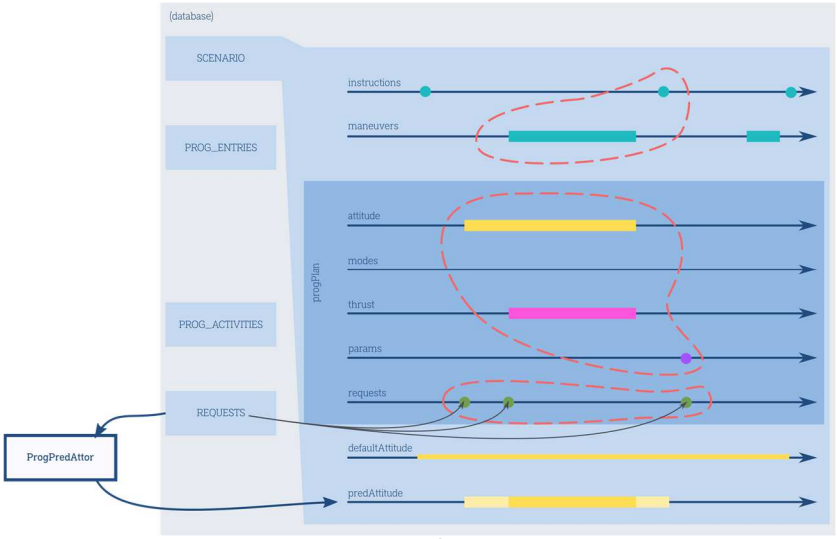
DOI: 10.13009/EUCASS2022-4691

Predicting programmed attitude — ProgPredAttor component

One use case of attitude prediction is to predict the attitude as programmed by the FDS; this is the purpose of the `ProgPredAttor` component.

It consists in taking each activity programmed (present in the `progPlan`) in the prediction. There are two possible options: to consider either the level as output from `ProgPlanner` (and thus the attitude feed of the programming plan) or the level as output from `Requester` (and thus the guidance requests).

In the case of autonomously guided satellites, and more particularly those that perform their attitude slews autonomously, the implementation of `ProgPredAttor` is in charge of computing and adding the attitude slews predicted according to a specific ground model.



`ProgPredAttor`

Predicting autonomous or default attitude — BasePredAttor component

Another use case of attitude prediction is to predict the attitude of an autonomously guided satellite or to set a “default” attitude and is realized by the `BasePredAttor` component.

Such an autonomous guidance can consist in alternating pointings between eclipse phase and daylight phase.

An existing implementation of `BasePredAttor` thus allows, by describing attitude switches, to define towards which pointing law the satellite should switch to when a given event is occurring (orbital event for example), and with what kind of slew – or no slew.

Use case: “Cancel any activities or requests”

Several situations may require the cancellation of a complete or partially elaborated programming plan or functional requests:

- It may happen that the functional requests produced cannot be uploaded on board, and this for multiple and diverse reasons.
- The need may arise to cancel a programmed maneuver without cancelling the whole programming.
- The programming sequence components may have a problem (failure due to a bug, violated constraint, …); in this case, it is necessary to go back and restore the programming plan.

All these needs are covered by the `Eraser` component. It expects one or more activities as input. These activities can be an instruction or a maneuver as potentially given to `ProgPlanner`, or an activity as produced by `ProgPlanner`, or a functional request activity as produced by `Requester`.

DOI: 10.13009/EUCASS2022-4691

For each of these activities, the role of `Eraser` is to remove from the programming plan all cause or consequence activities (thanks to the groups successively set up) and all “indissociable” activities (also thanks to the groups).

If requests are identified as having already been produced, `Eraser` also produces the request for their cancellation.

`Eraser` can also be used in a more classical way by asking it to erase the programming plan over a given time interval. Of course, it respects the logic of deleting all activities that are causal and/or consequential and “indissociable”.

5. A versatile architecture

The idea here is to show, through the interesting examples of `ProgPlanner` and `Requester`, how the various specificities (mission or other) that arise are addressed.

Factory design pattern

> *Wikipedia: “In class-based programming, the factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created.”*

This design pattern is used several times to address the same problem: using a specific algorithm to answer a specific need within a high-level component.

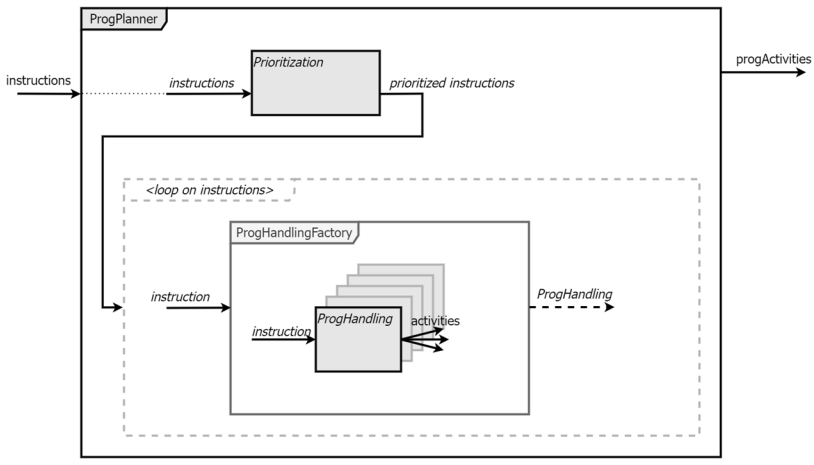
The main example is `ProgPlanner`. This component, as we have seen, does indeed have a high-level processing logic, but it is capable of handling very specific instructions and/or maneuvers.

At the planning step, when it comes to converting each instruction/maneuver into a set of activities, there is a need to use a specific algorithm to meet the mission specificities as we have seen. For this, `ProgPlanner` addresses itself to a *factory* by giving it directly the instruction/maneuver to be processed and by obtaining in return the implementation able to process it.

Such an implementation responds to a unique sub-component interface named `ProgHandling`, whose defined role is indeed to convert the given instruction/maneuver into the set of activities to be programmed in order to realize it (as described in the previous chapter).

This *factory* is described by the model used for the definition and design, in particular by the modeling of the links between the type of instruction/maneuver and the processing implementation. When configuring a FDS for a given mission, the `ProgHandling`s implementations needed for the mission are identified and assembled in the *factory*.

Thus, the specificities are carried by a couple {*instruction* + `ProgHandling`}, the `ProgHandling`s being isolated algorithms each responding to a same interface.



`ProgPlanner` — component simplified diagram

DOI: 10.13009/EUCASS2022-4691

Multiple and ordered factories

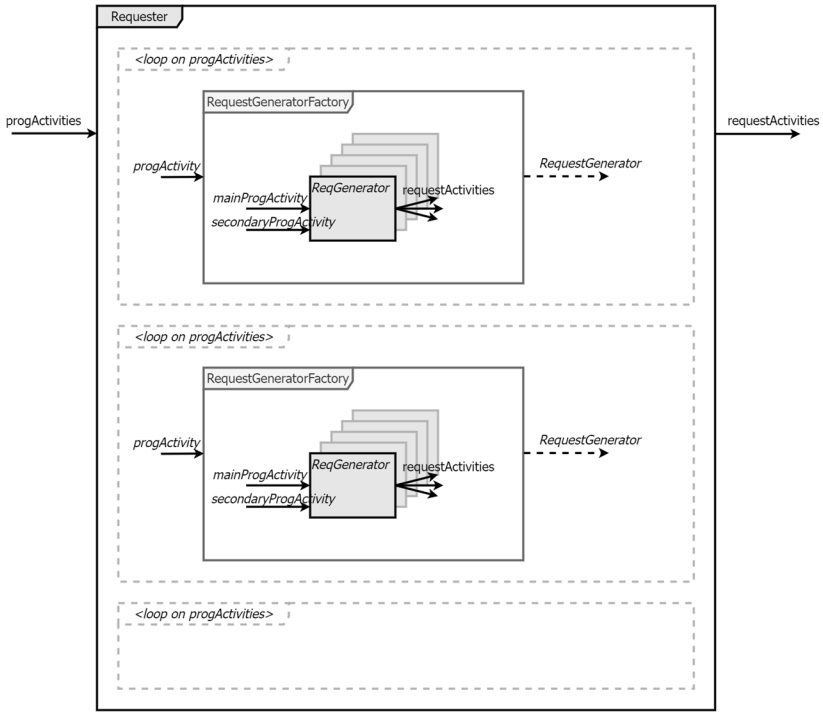
The *factory* design pattern is not suitable if potentially the same input (or at least inputs of the same type) should lead to different processings depending on the context, or if two inputs should be considered together.

This is potentially the case in `Requester`. Indeed, a given type of activity (an activity to be programmed produced by `ProgPlanner`) may, depending on the case, give cause to different functional requests, and two activities may have to be processed together to produce two mixed functional requests. For example, a maneuver functional request (for onboard-ground interface reasons) may contain the parameterization for a thrust, but also the parameterization indicating the pointing mode towards which the satellite must switch at the end of the thrust; Thus, to produce this query it is necessary to consider both the attitude activity and the thrust activity produced by `ProgPlanner`; and this becomes more complex when the attitude activity carries information for the thrust attitude but also for the end-of-thrust attitude (this happens when the thrust duration is not precisely predictable).

Our solution to this problem is to still use the *factory* design pattern, but in a more segmented way. Such a `Requester` *factory* returns a `RequestGenerator` implementation for a given type of activity (to be programmed).

A `RequestGenerator` thus expects as input a so-called *main* activity to be programmed, and produces the consequent functional request activity(ies). But it is able to identify by itself (thanks to groups of activities) one or more so-called *secondary* activities to be programmed which it then references as outputs so that `Requester` also considers them to be well processed and does not seek to process them individually later. In the previous example, the main activity is the push activity, and a secondary activity identified is the attitude activity.

The remaining difficulty is to order the processings of the activities so that `Requester`, again in the previous example, does not start by processing the attitude activity individually when this one in particular should be processed together with the thrust activity. For that purpose, `Requester` does not have one instance of this *factory* but several; each assembled differently with different `RequestGenerator`s. These *factories* are arranged by configuration (not by specific code) in an ordered list. In this way, `Requester` first uses the first *factory* (in our example, the one containing the `RequestGenerator` dealing with maneuvers) on each of the activities to be programmed, and then the next one in the list (in our example, a *factory* containing the `RequestGenerator` capable of handling “normal” attitude activities) on the remaining activities.



`Requester` — component simplified diagram

DOI: 10.13009/EUCASS2022-4691

Note that this semblance of prioritization is here carried by the configuration, and does not require any specific algorithm.

And let us insist on the fact that `ProgPlanner` and `Requester` are two components which implement two different strategies, because `Requester` has to handle *secondary* activities (a `ProgHandling` only takes one instruction, a `RequestGenerator` has to be able to take several activities).

Specificities carried by the parameters themselves

Sometimes, more simply, the mission specificities do not require a specific algorithm, and one can manage to define and design a relatively generic implementation whose parameterization can itself carry these specificities.

The component in charge of predicting a default attitude, `BasePredAttor`, can be parameterized by a system of switches: a switch gives the attitude law on which to switch at the occurrence of such or such event, specifying if a slew is to be inserted to start or end at the time of the event and according to such or such parameterization.

Another example, the hierarchical step implemented by `ProgPlanner` can be reduced to a single implementation associating by parameterization a priority level to a type of instruction.