# SIRIUS-DV: The New Flight Dynamics Algorithms for the Future CNES Missions

By Iván LLAMAS,[1)] Yannick TANGUY,[2)] Michel LACOTTE,[2)] and Jean-Jacques WASBAUER[2)]

[1)] *GMV, Madrid, Spain*
[2)] *CNES, Toulouse, France*

The SIRIUS project aims to develop a set of Flight Dynamics (FD) products that will be used operationally in the control centers of the upcoming CNES missions. It mainly covers three different layers: the mathematical low level libraries (PATRIUS), intended to be used either in an operational environment or in expert studies; the flight dynamics algorithms, implementing the operational functionalities (SIRIUS-DV); and the FD applications, that include the assembly of the algorithms to build stand-alone applications – with dedicated Graphical User Interfaces (GUI) - and the infrastructure services (such as time, messages, logging, …) needed in an operational FDS. This paper focuses in the second layer, the software applications implementing the flight dynamics algorithms

**Key Words:** SIRIUS, flight dynamics, scenario

## 1. Introduction

The development of a Flight Dynamics System (FDS) involves different layers, from the most basic mathematical libraries up to the FD applications themselves running in an operational control center, with support functions (logging, time, visualization…).

After analyzing the lessons learned over the last few decades in the development of such systems, CNES has decided to try a different approach in the development of their new products, making a clear and clean separation between the development of the computational layer and the development of the supporting services. Two different contracts have been issued, with a very limited interaction between them, mostly given by the definition of the interfaces (both for the data and the services). Figure 1 shows this division.

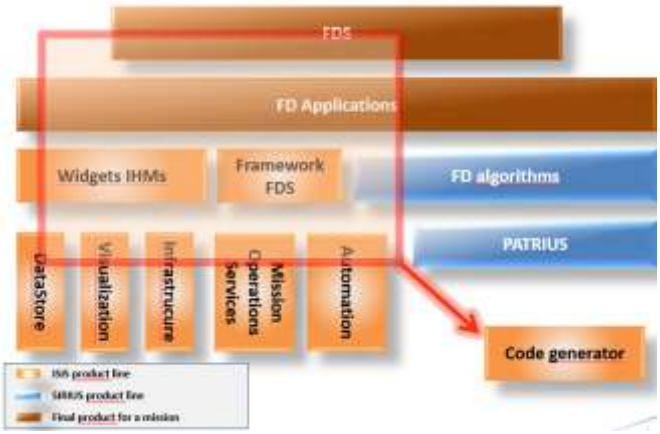


Fig. 1. Elements to Build a FDS.

Hence, to build the FD application, clear interfaces between all the elements are defined by CNES, constituting the logical architecture of the system. Even if this paper is mainly focused in the development of the "FD algorithms" part, it's very interesting to explain here at least how the computational part (from now on called "FD service") of the FD applications will be created.

Each FD service can call other services as part of its own computation; however, it will only know their interfaces, the implementation of the called services being therefore totally exchangeable. For instance, the service in charge of the orbit restitution will call the propagation service and also the function to resolve the linear system, but the implementation of both sub-services will be totally unknown for the parent service.

All the services are stateless, so their results are completely defined by the provided inputs, allowing – among other things - an easier automation. The role of each service is therefore fully defined by its interface (inputs and outputs), and each possible implementation of the service shall comply with this role. Moreover, the interface of each service is very simple, since its defined by one single method, whose arguments are the list of inputs that are common to all the possible implementations of the service, a structure containing the specific parameters of each implementation (which is abstract, so as to guarantee the exchangeability and which any parent service won't be allowed to change or manipulate) and the response structure containing all the output data.

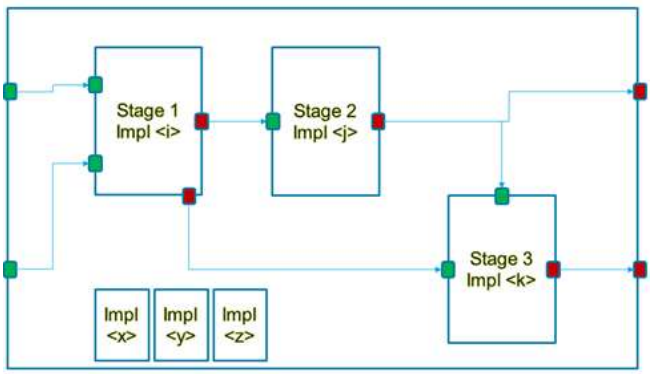


Fig. 2. Assembling a FD application.

The Figure 2 shows an example on how a FD service with

three different stages (each one being a FD service) would be assembled. For a given mission, some specific implementations of each service (in the figure, <i>, <j> and <k>) will be used, but any of them could be easily replaced by any other implementation (<x>, <y>…) that complies with the defined interface.

This architecture of the FDS allows on one hand a high degree of flexibility (all the implementations of a given service are exchangeable) and scalability (the effort to add new functionalities is reduced), and on the other hand allows a clear separation between the development of the computational layer and the rest of the system. Moreover, the choice of technologies also guarantees its non-obsolescence up to - at least - twenty years from now.

## 2. Technical Domains of the FD Algorithms

The FD algorithms are divided in several technical domains, which are briefly explained in the following sub-sections.

### 2.1. Conversions

This domain handles the conversion of orbital and attitude data within the FD. It contains functionalities to perform the conversion of the dates (format, time scale…), of the orbital parameters type (cartesian, keplerian, circular, equinoctial…), of the orbital parameters frame (IERS 2003 and 2010 conventions are supported), the conversion to mean elements, the conversion of attitude parameters type (quaternion, Euler angles, matrix…) and of the attitude frame.

### 2.2. Ephemeris Generation

This domain is in charge of providing the functionalities to compute the dynamic state of the satellite at a given date, at a given event or over a period of time. The concepts of the core service of this domain – the Productive Propagator – are explained in section 5. In addition to the productive propagator (that, as we'll explain later on, is a very generic state generator), this domain also includes a service in charge of generating the nominal ephemeris both for orbit and attitude that will afterwards be converted into CCSDS OEM/AEM format.

### 2.3. Events

This domain regroups the functionalities involved in the computation and treatment of the events and phenomena needed for the mission. A wide range of events and phenomena are available (orbital elements, antenna visibility, attitude events, events between different satellites, programming events, external events to the FD…) and due to the architecture of the system, more can be easily added.

It also handles the post-processing of events (combination of several events) and phenomena (logical operators, such as the union, intersection…).

### 2.4. Scenario

It includes several services in charge of handling and analyzing the data scenario (the details on this data are found in section 3): generation of N dispersed scenarios (the number and parameters to disperse are completely defined by the operator in the inputs, the service being totally unaware of the modifications to be done a priori), creation of several alternative scenarios from a point on (to simulate, for instance, a switch into survival mode), comparison between different scenarios (orbit, attitude…).

### 2.5. Orbitography

One of the core domains of any FDS, it includes all the functionalities needed for the orbit determination.

The most relevant services are the one in charge of computing the theoretical measurements (and their partial derivatives with respect to the estimated parameters), all the measurement functions (angular, one-way and two-way Doppler, PVT, GNSS), the measurement treatment (to verify their validity), the measurement simulation, the least-squares filter (LSF) solver, the orbit restitution service (based in a least-squares method), the orbit determination qualification (to check the results of the determination) and the collision risk assessment.

### 2.6. Orbital Maneuvers

Another main domain of all the FDS, it involves all the functionalities linked to the maneuver computation, from the definition of the maneuvers strategy to the computation of the station keeping maneuvers (in longitude, local time and eccentricity), the simulation of a user-defined maneuver, the optimization of the rendezvous maneuvers, the computation of collision avoidance maneuvers or the end-of-life strategy computation.

It also includes other supporting services as the one in charge of treating the TM to build the observed maneuvers, and that responsible of computing the propulsion parameters.

### 2.7. Guidance and Programming

This domain handles the computation and prediction of the attitude (attitude laws, slew computation…) within the FDS. It also includes the functionalities required to program the satellite, such as building the programming plan, the verification of mission constraints and the computation of the allowed slots to perform FD activities.

### 2.8. Mission

This domain includes services such as the one in charge of computing the reference orbit using analytical models (to simulate the orbit computed on-board when the spacecraft is in autonomous mode) and the one responsible of computing the trajectory to follow in order to go from an initial orbit to a targeted one. It also includes some services that are in charge of simulating the collision risk trajectory analysis when the satellite is controlled by the on-board.

### 2.9. Interfaces

In charge of consuming the external information (EOP, satellite data base, TM…) to create the data handled by the FD services. This domain is also responsible of the generation of the operational products (orbit/attitude files…) that are exchanged with the other subsystems and/or entities.

### 2.10. Calibration

This domain is in charge of performing the calibration and computation of the satellite inertias and also the calibration of the thrusters, mainly by processing the TM information received when these activities are carried out on-board.

### 2.11. Scenario Processings

In charge of handling the different parts of the data scenario, such as trajectory, attitude, maneuvers, MCI (mass, center of gravity and inertia), thrusters, tanks and solar arrays. This

domain is explained in more detail in section 4.

## 3. The Data Scenario

As already mentioned, the scenario is the central data of the FD algorithms. It contains all the information (both past and future) of the satellite dynamic state for one (and only one) spacecraft, including its programming plan. It is static and it is only defined by the time (i.e., the transitions between the different activities are not given by events). We can define several scenarios for a given satellite, the nominal one and as many alternative scenarios as desired (for instance, to simulate a switch to survival mode, to simulate the effects of not performing a maneuver…). The nominal scenario represents, at a given date, the best knowledge that we have of the whole mission, in the past (the trajectory coming from the last orbit determination, let's say) and in the future (the propagated trajectory). It must be remarked that this data has no associated operation (its treatment is performed by the processings, section 4), and it represents the evolution in parallel (but not independent) of the different domains (trajectory, attitude, maneuvers…).

The most basic elements of the *scenario* are the *activities*, each one of them associated to a given domain. For instance, we can define trajectory activities (numerical propagation, keplerian propagation, GPS almanac propagation, PV ephemeris treatment…), maneuver activities (impulsive maneuver, continuous maneuver…), attitude activities (reference attitude, slew, guidance laws…) and so on. The activities contain all the information needed to simulate the model that they represent, and they are all independent from one another (they can however make reference to another feed/blend). We'll also need to handle coherent sets of activities (for instance, different maneuvers activities – impulsive/continuous…- refer to the same maneuver), hence there's also a notion of *group*.

The activities are organized (in chronological order) in feeds, which represent a temporal axis that may potentially cover the whole lifetime of the satellite. Each feed is dedicated to a given domain, defined by the type of activities it can include (for instance, the maneuver feed can only contain maneuver activities). We can however define several feeds for a domain (following the same example, there can be a feed for the impulsive maneuvers, another one for the continuous ones…). The activities in a feed cannot overlap, but they may allow gaps within (again, the maneuver feed allows it since not at every date we'll have a maneuver, while the trajectory feed won't allow it since we need to know the trajectory of the spacecraft at every moment). The feeds can also be grouped in bunches, a simple aggregation of coherent feeds (for instance, the orbital bunch groups the trajectory, the mass and tanks feeds, while the maneuver bunch groups all the maneuver feeds).

Finally, the higher elements in the hierarchy are the blends, whose objective is giving the best global vision of the state for a given domain via a synthesis of the different feeds. Several ways are defined to handle the synthesis of the feeds into a blend: by fragments, the blend is composed of time intervals, each one pointing to a single feed (this is the implementation used for the trajectory); by priority, the blend is constituted of all the possible feeds of the domain, the feed to be used at a given date is defined by the available information at that date and a priority order which is part of the definition of the data (used for the attitude); by selection, the definition of the blend contains the activity to choose (for instance, used in the maneuver blend, where all the types of maneuver activities are present and the operator selects the one to be used depending on the need).

The following figure (Figure 3) shows a simplified example of the concepts explained in this section. We have a trajectory blend with two fragments, the first one pointing to the feed number 2 and the second one pointing to the feed number 1. This means that for any date within the interval of the first fragment, we'll use the information of the Feed 2 to obtain our best knowledge of the satellite trajectory (respectively with the second fragment and the feed 1). Each trajectory feed contains several activities and in particular we can see that the "Numerical Activity 1" (NA_1) has a reference to the "Impulsive Maneuvers feed", while the "Numerical Activity 2" (NA_2) points to the "Continuous Man Feed". This means that in the simulation of the NA_1 we'll consider the maneuvers present in the impulsive feed, while in the simulation of the NA_2 we'll take into account the continuous ones. We also show a group for the second maneuver activities.

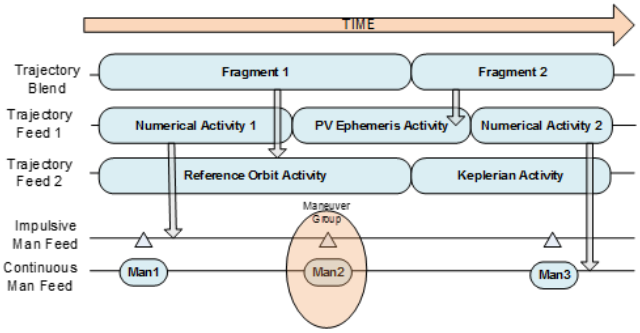


Fig. 3. Blends, Feeds and Activities.

To go a step further away in the explanation of this example, the operator could potentially define up to three different ways to compute the trajectory of the satellite. He could decide to use the trajectory blend, so the satellite will follow during the first fragment the trajectory defined by the trajectory feed 2 (in the example, the trajectory will be provided via a reference orbit) and will follow the trajectory of the first feed during the second fragment (in this example, via an ephemeris file up to the end of the PV Ephemeris activity, and via a numerical propagation afterwards). But he could also just decide to simulate the satellite's trajectory using the first feed (so it will follow the sequence Numerical Activity 1, then the PV ephemeris, then Numerical activity 2) or the second one (so the trajectory will be defined by the reference orbit plus a keplerian propagation).

It must also be noted that the contents of the scenario (list of blends, bunches, feeds) are defined in an abstract way, so each mission can (and must) decide its own structure of the

data.

As an additional remark, one could argue that treating and handling such "a big object" as the scenario (which, let's remember, contains the information of a whole mission, so up to several years of data) is not optimal from a memory consumption point of view. To tackle this issue, the mechanisms of lazy data loading are widely used in the system. The concept behind is just that we don't load the data in memory unless they're needed; this is quite adapted to the typical FD operations, where usually only the data of some days are needed at the same time to perform the computations. So, even if the scenario we receive as input gathers the information of tens of years, this information will not be loaded in memory. In fact, the scenario being composed of several blends/feeds/bunches, it will at first only contain the references to those elements. This depends on the implementation, but for the sake of the argument let's assume that they are just files, so the data scenario that a service receives in input will only contain some dozens of file names, hence the memory consumption would be negligible. Afterwards, the service will need to actually handle the data over the computation interval, but this interval will be at most of some days, and even in that case, the data are loaded in memory using a cache of blocks each one covering a reduced time span, and which are only loaded if (and when) needed.

## 4. Scenario Processings

The scenario processings cover the functionality needed to simulate the different elements that compose the scenario. Given that the scenario is organized in activities, feeds and blends, the same division is followed in the processing (there's one processing per activity type, one per feed type and one per blend type).

They're also divided in domains (trajectory, attitude, maneuvers, tanks, solar arrays, MCI and thrusters), each one defining the interface that the processing must implement. As mentioned before, there can be dependencies between the different processings. For instance, the processing of the trajectory feed is based in the processing of the activities within the feed; but the processing of a given trajectory activity may also depend, for instance, on the processing of a maneuvers feed and on the processing of the mass characteristics feed. The following sub-sections give an overview of the processing for each domain.

### 4.1. Trajectory Processing

Basically, they provide the trajectory state (position and velocity) of the spacecraft at a given date. In the case of the blend processing, the computation date defines the fragment to be used, and hence the associated feed; it then invokes the feed processing to obtain the trajectory. Respectively, in the case of the feed processing, the computation date allows the selection of the activity to be used to simulate the trajectory of the satellite. Please, remember that we must be able to know the satellite's orbit at any moment, so if no activity exists at the required date, an error will be raised. Once the activity is selected, the feed processing invokes the processing of the activity. The activity processing is finally specific for each type of activity. For a numerical activity, a numerical propagation with the defined integrator, forces and models is performed to obtain the trajectory state at the desired date; for the keplerian activity, a simple keplerian propagation is performed; for the ephemeris activity, the trajectory state is obtained by interpolation in the ephemeris data contained in the activity, and so on.

We can introduce here the concept of complete and incomplete activities. A complete activity contains all the elements needed to compute its state at any given date of its interval, while the state of an incomplete activity is given by the previous one. In the case of the trajectory activities, the distinction is quite simple and easy to understand: a complete activity contains the initial orbit (not necessarily at the begin date of the activity interval) that must be handled to get the trajectory state at any given date within the activity limits; an incomplete activity does not contain the initial orbit, so in order to get its state at a given date, we should first process the previous activity (what we call "synchronization") to get the orbit at the initial date of the current activity, and then perform the propagation up to the desired date within the current activity interval. Even if this might seem somehow odd at first, it is in fact a very easy and efficient way to ensure the trajectory continuity in the transition between activities.

### 4.2. Attitude Processing

They provide the attitude state (orientation, angular velocity, and – eventually - its derivatives) of the spacecraft at a given date. In the case of the blend processing, the different feed processings are invoked at the computation date in decreasing priority order (so if the one with the highest priority doesn't exists for the date, the next one is invoked and so on). From the feed processing on, the same logic as that explained in the trajectory processing applies here (but accounting for the fact that an attitude feed may have gaps in the activities it contains). Once again, the actual treatment of each activity depends on its type (interpolation in a given attitude ephemeris, computation of the attitude from the defined guidance law…).

### 4.3. Tanks Processing

They provide the state of the tanks (propellant mass, temperature, pressure) of the spacecraft at a given date. The same logic as that of the trajectory processing applies here for all the levels (blend, feed and activity).

### 4.4. Solar Array Processing

They provide the state of the solar arrays (orientation) of the spacecraft at a given date. Same logic as that of the attitude processing applies here.

### 4.5. MCI Processing

They provide the MCI (mass, center of gravity and inertia) state of the spacecraft at a given date. Same logic as that of the trajectory processing applies here (except for the fact that there's no blend processing, given that only one MCI feed is defined).

### 4.6. Maneuver Processing

They provide the delta-V produced by an impulsive maneuver or the force produced by a spread maneuver over a given interval. In this case, the blend processing knows the type of maneuver that must be handled, and it therefore

invokes the corresponding feed processing. The feed processing then invokes the processing of the activity that is active at the computation date/interval. It may happen that no activity (that is to say, no maneuver) is modeled at the computation date, and in that case the feed processing will send a null output (but no error is raised).

### 4.7. Thruster Processing

They provide the thruster state (force, flow rate, throughput) for a given thruster at a given date. Same logic as that of the trajectory processing applies here.

## 5. Productive Propagator

The productive propagator is one of the core elements of the FD algorithms. Its function is the production of any parameter at a given date or over a given interval using the dynamic state of the satellite provided by the treatment of the scenario (hence, computed by the scenario processings). It relies on the PATRIUS propagator [1] in master mode, and in particular it makes use of two mechanisms: *EventDetector*, so as to compute the required parameters when an event is detected and *StepHandler*, so as to compute the parameters at certain dates.

Its inputs are the computation interval (optional), a list of date descriptors (each one containing a list of output parameters descriptor), a list of event descriptors (each one also containing a list of parameter descriptors) and an optional scenario state (whose role will be explained later). The date descriptors define the dates at which the parameters will be computed. Several types are defined (and others can be easily included) such as fixed dates defined by the user, dates in a given interval with a fixed step, dates coming from an input set of records (TM, ephemeris…) and dates associated to a given phenomenon (with a given step within the phenomenon period). The event descriptors define the events at which dates the operator requests the generation of the output parameters (for instance, computation of the local time at each equator crossing). Finally, the output parameters descriptors define the parameters that will be computed (PV elements in a given reference frame, the satellite angular momentum…) and also provide the description of the required scenario descriptors (trajectory feed, maneuver blend…). The Figure 4 represents this architecture of data.

Respectively, the outputs of this service are a list of output ephemeris (one per date descriptor and per event descriptor) containing each one a list of records (one per output descriptor) composed by the output descriptor itself (so as to be able to identify which parameters it contains) and the list of values of each computed parameter (X, Y, Z…) and, optionally, a list of scenario states (which can be used as input for further calls to the service, see above).

Each output parameter descriptor is associated to a given part, the element in charge of actually performing the computation of the required parameters. Each part is created via a dedicated Factory that uses as input the output parameter descriptor. Hence, the data itself defines the expected behavior, allowing a generic treatment within the Productive Propagator (that is to say, the algorithm of this service doesn't need to "know" which parameters it is actually computing). Each part must implement an interface with several methods, basically a "compute" one (in charge of performing the actual computation) and another one in charge of giving the list of descriptors (feeds/blends) of the scenario (trajectory, attitude…) containing the data that it needs to perform its computation.

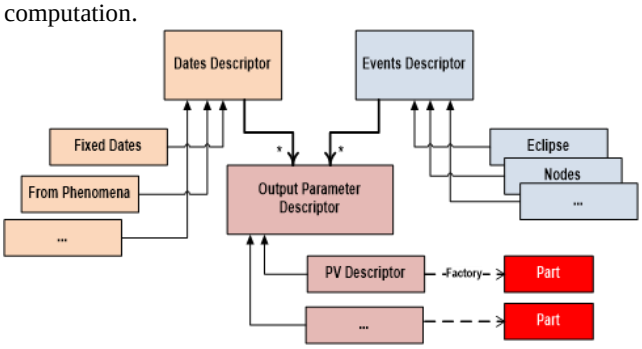


Fig. 4. Productive Propagator Input Definition.

The duality and separation of roles between the simulation/propagation of the dynamic state (performed by the scenario processing) and the output parameters computation (performed by the associated part) must be remarked. For a given execution of the productive propagator, the output parameters that are to be computed define the elements of the scenario (trajectory state, attitude state, thruster state…) that are needed and also the descriptors that will be used to compute them. For each date (defined either via the dates or event descriptors), the propagator provides the scenario state at the date by using the corresponding scenario processing. Once the scenario state is known for the date, the different computation parts are invoked using this state as input and they do their job filling the output parameters structure.

Some paragraphs above we mentioned that this service may receive a scenario state as input, which is the data structure that groups all the different states (trajectory, attitude, MCI, and so on…) computed by the different processings at a given date. This scenario state is used (if provided as input) as initial state for the processing computations, so instead of using the state contained in the different activities, the scenario processing will eventually consider the information coming in this structure. This mechanism is implemented in order to improve the time performances of the computation, in particular when several calls to the productive propagator service are performed within another FD service. For instance, in the computation of the impulsive maneuvers strategy we will need to compute several maneuvers, each one of them depending on the previously computed maneuvers (since the trajectory of the satellite changes). For simplicity's sake, let's imagine that all the maneuvers will lay within the time interval of a single trajectory activity (a numerical one), whose state (i.e. orbit) is defined at a given date T. To compute the first maneuver, we will need to know the trajectory just before the maneuver date (T1), so a propagation will be done from T to T1. Once the first maneuver is computed, we go on and try to compute the second maneuver; to do so we'll need to know the state just before the date of the second maneuver (T2), so a propagation will be needed up

to T2. If no additional information is provided, and since the trajectory activity is "complete", the default behavior will be to perform a propagation from T (the date of the activity state) up to T2. However, we've already computed part of this trajectory (up to T1, the information being included in a scenario state at this date) just before, so we can use it to speed up things and just do the computation over the reduced interval (from T1 to T2).

Coming back to the duality between the processings and the parts, let's consider an example where the operator wants to compute the local time at each equator crossing and also the ephemeris of angular momentum of the satellite over an interval at fixed dates. For the first output, an event descriptor (equator crossing) will be used while for the second one a date descriptor is needed. The first output requires the knowledge of the satellite's orbit and to do so it will use the information of the trajectory blend. The second output requires knowing the attitude (angular velocity) and inertia of the satellite, so it will use an attitude feed and a MCI feed. For each date, the computation of the scenario state (in this case, trajectory, attitude and MCI states) will be performed by the associated scenario processings (trajectory blend processing, attitude feed processing, MCI feed processing). Afterwards, the two parts responsible to fill the output parameters will be invoked: the first one will use the trajectory state (position and velocity) to compute the local time; the second one will use the angular velocity and the inertia data from the attitude and MCI states to compute the angular momentum. For instance, it's the part's responsibility to perform any frame/type conversion needed.

Once again, this architecture provides a very flexible and easy way to evolve the system, since we can add as many output parameters (i.e. computation parts) as desired without modifying at all neither the scenario processing nor the implementation of the productive propagator (provided that the part is compliant with the defined interfaces). In addition to the high capacity for evolution, this architecture guarantees that adding a new part won't have any side-effects in the rest of the system, so we can be sure that it will still work exactly as before (hence, reducing the validation effort and cost at system level).

## 6. Development Process

The development of the FD algorithms relies on a data model managed by the CNES domain experts and which is updated gradually as the development advances. It contains the definition of all the data that are used in the algorithms, the definition of the interfaces (inputs/outputs) of each algorithm and the software requirements that the different implementations must meet. Using this model as input, the implementations of both the data and algorithms interfaces are automatically generated (using a code generator that is also part of the SIRIUS line of products), which serve as starting point for the development carried out by the team.

The SIRIUS-DV algorithms are developed in Java using an Agile/SCRUM methodology with sprints (realization iterations) lasting four weeks. The functionalities to be developed in a given sprint are presented (at the beginning of each sprint) to the team by the CNES domain experts. During the sprint a constant communication flow is established between both parties in order to ensure the understanding – and hence the quality – of the tasks to be done (the development team being physically located at CNES premises).

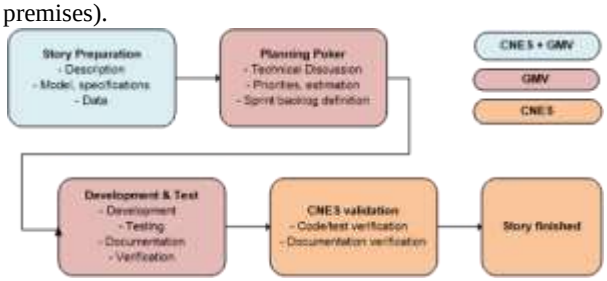


Fig. 5. Story Life Cycle.

The lifecycle of each story is presented in Figure 5. First of all, the story must be prepared, the model and specifications are modified by CNES experts and verified by both CNES and GMV teams, so as to be sure that all the required elements are ready to start the implementation. Afterwards, the stories are included in the sprint backlog during the planning poker meeting (considering the estimated effort and the relative priorities). Afterwards, the implementation, testing, documentation, validation and verification activities are performed by the GMV team. Once this step is done, the CNES experts verify that all the produced elements (implementation, tests, and documents) are in line with the expectations. They provide comments which are answered and/or implemented by the team. Once everything is in line, the story is declared as finished.

At the end of each sprint, those functionalities that are finished are presented to the users by the development team, so a fully usable product is available once a month, with increased functionalities over time.

## 7. Conclusions

This paper has given an overview of the development of the new FD algorithms that will be used in the upcoming missions operated by CNES. The different domains in which the system is divided have been described. The main data treated in the algorithms, the scenario, which contains all the information (past and future) of a given satellite over its whole lifetime has been presented. The mechanisms to handle this data, scenario processings, have been described with more detail. The architecture of one of the main FD services (the productive propagator) and its relation with the scenario processing has also been discussed in the paper.

Finally, a brief description of the development process, based in an Agile/SCRUM methodology has been provided.

## References

1) PATRIUS User Manual (SIRIUS-SUM-DV-10037-THA).