SIRIUS Model-Driven Software Product Line for Flight Dynamics Systems

By Pâmini ANNAT,[1)] Romain BERNARD,[2)] and Jesús ESTEBAN DONES[3)]

[1)]*Centre National d'Etudes Spatiales, CNES, Toulouse, France*
[2)]*EXMS, France*
[3)]*Thalès Services, Toulouse, France*

The SIRIUS product line defines the next generation of Flight Dynamics Systems for the upcoming CNES missions such as SWOT, MICROCARB and MERLIN. A feedback study on a decade of FDS development and operations by CNES drove to the conclusion that the next generation of FDS software should be implemented with increased reusability and modularity. A model-driven approach based on DSLs (Domains Specific Languages) has been identified by CNES as a natural candidate to implement this business capitalization and provide a new way of designing FDS software. This paper presents the rationale behind the customized model-driven strategy specifically implemented for the SIRIUS product line. Then, it defines the family of software the SIRIUS product line is dedicated to, what kind of problems it can solve, the overall model-driven development process and the various managed artifacts (models, requirements, code...).

Key Words: Flight Dynamics System (FDS), Model-Driven Engineering (MDE), Domain Specific Language (DSL), Graphical User Interface (GUI), Software Product Line (SPL)

1. CNES Context

CNES, the French Space Agency, is in charge of the overall system design of multiple LEO (Low Earth Orbit) satellite missions for scientific purposes (such as SWOT, MICROCARB, MERLIN) and defense purposes, from space segment to ground segment.

Besides the space segment, the other critical part of a mission is its Control Ground Segment (CGS). As a part of the CGS, the Command and Control Center (CCC) hosts the Flight Dynamics System (FDS) which is dedicated to orbit determination and propagation, Launch and Early Orbit Phase (LEOP) / Station Keeping (SK) / End Of Life (EOL) maneuvers, guidance and programming, platform calibrations, mission specific calculations.

Within the Orbital Systems Direction (DSO) of the Toulouse Space Center (CST), the department in charge of the development and maintenance of flight dynamics libraries and operational Flight Dynamics Systems has decided, through the SIRIUS project, to set up an innovative product line in order to define the next generation of Flight Dynamics Systems for the upcoming LEO missions. The SIRIUS product line is carried out through 3 layers of development:

- Base level Flight Dynamics library for studies, mission analysis and operational products (PATRIUS),
- Flight Dynamics (FD) Algorithms for each thematic domain to be covered (orbitography, maneuvers, guidance and programming…),
- FDS operational systems defined for a given mission and to be operated from satellite BOL (Beginning of Life) to EOL (End of Life).

This paper focuses on the design process for the 2nd and 3rd layers of development (FD Algorithms and operational FDS). The 1st layer is addressed in Ref. 8 and the 2nd layer is complementary addressed in Ref. 2.

2. FDS production challenges

Due to long term lifetime of satellites missions, usually around a decade and sometimes even more, the past product line based on obsolete informatics languages and classical development cycle had to be replaced. The experience feedback over a decade has shown a double challenge to be fulfilled by the FDS software:

- Reusability from one mission to another with a common basis of FD algorithms;
- Adaptability to each mission's specific constraints and requirements.

The algorithms commonality is highlighted especially on LEO missions for orbit determination, propagation, events calculation, standardized interfaces of the FDS within the CCC.

The algorithms variability is essentially due to difference between satellite platforms for guidance and programming requirements, mission performances and precisions requirements, orbit phasing, mono or multi satellite missions, satellite autonomy level.

A feedback study on a decade of FDS development and operations by CNES drove to the conclusion that the generation of FDS software for the next decades should be implemented with increased reusability and modularity, maximizing the assembly of well-defined and validated Flight Dynamics (FD) services from one mission to another.

Thus, the reusability of algorithms and associated Graphical User Interfaces (GUI) would not only be a benefit on development and maintenance costs, but also a simplified training for FDS operators who work on different missions.

3. SIRIUS product line objectives

The SIRIUS Software Product Line (SPL) is designed for FDS production. It provides FD components that can be decomposed in interchangeable FD services; a Datamodel that can be edited through UML concepts in a collaborative way;

and applicative GUIs that represent parameter sets, driving the underlying functional services.

The SIRIUS model-driven product line follows the 3 main principles applying to SPLs: "*exploring commonality among products to proactively reuse software artifacts, encouraging architecture-centric development, and having a two-tiered organizational structure*".[1)] It also implements or allows all 17 success rules detailed in Ref. 4 that every SPL should implement, and take inspiration from industry's accumulated experience and guidelines. But most of the existing documentation about SPLs does not consider model-driven techniques. We will try in this paper to demonstrate that the model-driven orientation can solve problems and limitations of traditional SPLs approaches.

The SIRIUS product line is designed to minimize the design and the development effort for each new orbital mission. It is designed to provide the following expected benefits:

- Maximize the reuse of well validated FD software components from one mission to another, thus reducing the cost of development and maintenance.
- Capitalize knowledge and validated software components from one mission to another, through an off-the-shelf library of reusable FD components.
- Facilitate the maintenance of the next FDS generation.
- Harmonize FDS operability concept to optimize operator daily tasks on several missions.

4. SIRIUS product line strategy overview

The chosen product line strategy is based on two main complementary axes:

- The identification of dedicated software architecture to provide interchangeability of FD services implementations, and a way to assemble the complex functionalities an FDS requires.
- A domain-specific model-driven approach that perfectly fits to this defined architecture, and provides support, from early functional requirements identification to code generation, through various kinds of models (Requirements, FD analysis, FD components design, FD services assemblies, data widgets, FD applications GUI).

4.1. A dedicated software architecture

With the reusability objective in mind, the first activity was to define precisely a software "logical architecture" that could support:

- FD services loose coupling,
- FD services interchangeability, even if each FD service implementation can define its own set of extra parameters,
- FD service composition/decomposition in order to assemble a FD service and its required sub-services,
- A set of behavioral expectations on datatypes (which exceeds the scope of this paper),
- The ability to edit and launch assembled FD services with default generic GUIs.

We ensured the feasibility of all these assumptions with the realization of proof of concepts and prototypes, illustrating effective services interchangeability and assembly.

The logical architecture is a cornerstone of our product line, since the defined concepts are designed to express simple solutions to the variability-intensive challenge. It is also required by our model-driven process, because design models and metamodels are naturally based on the logical architecture concepts.

4.2. SIRIUS model-driven approach

Traditional development approaches mainly rely on the production of semi-structured word-documents that describe the functionalities, requirements, designs, tests, quality of the software to implement. Associated with processes and quality standards, they have proven their utility to reach development process milestones and control the implementation of systems of arbitrary complexity.

With model-driven approaches, word-documents are replaced by a set of models that capture the specification information, and provide the ability to generate the required and usual word-documents, that still exist for purpose.

In a model-driven development process, the ability to produce the usual high quality documents is primordial. If this requirement is not reached for any reasons (incapability to mix manual documentation inside generated documents, incapability to generate documents in an iterative and incremental manner, incapability to respect custom styles and stylesheets…), the document will finally be produced by hand, and the model would become unreliable, since not any more synchronous with the model.

The model-driven approach has been identified for SIRIUS SPL as a natural candidate to implement this business capitalization and provide a new way of designing FDS software. It can be of course applied to other technical domains.

Many different "model-driven" philosophies exist (MDA, MDD, MDE, MDSD…), each of them focusing on different objectives and ways to use model-driven techniques. Using a model-driven approach is not a pledge of success by itself and many reasons can lead to a project failure or investment loss.
The main characteristics of our model-driven strategy rely on the following subjects:

- DSL vs GML : Domain Specific Languages vs Generic Modeling Languages,
- Custom DSL tooling,
- Models are the source,
- Pragmatic modeling.

4.2.1. DSL vs GML

Domain-Specific-Languages have many advantages against their Generic-Modeling-Languages (e.g. UML) counterparts. They can express more with less, because they define a concise and precise vocabulary (e.g. FDApplication; FDAssembly, FDService, FDData…) that has much more semantic strength than any generic concept (e.g. class, interface, operation…). As a consequence, models can be simply expressed so as to facilitate their exploitation by generators.

The essential downside of the DSL orientation is the obligation to invest in a custom Domain-Specific-Tooling and assume its maintenance. On the contrary, GMLs provide a generic toolbox that can be immediately reused (profiles, generators), but it rarely meets the specific requirements of a precise technical domain (custom architecture, space standards constraints, code efficiency or testability…), because generic generators exploiting generic models can never fit perfectly to everyone needs.

In order to succeed, a DSL modeling approach should propose a modeling mindset framework that is suited to the way FDS thematic experts think. It should reuse the existing technical vocabulary whenever it is possible, propose a modeling process that relies on available user's expertise and knowledge, and "invent" some necessary concepts, watching out they are simple enough to be clearly understood and used.

4.2.2. The SIRIUS custom toolbox

Adopting a new model-driven design process, changing the way people were used to work, is a classical issue. To ease the acceptation of a model-driven process, it has been important to develop a SIRIUS custom toolbox to give new abilities to users, and get the maximum value from the models.

The DSL custom toolbox is of great importance for several reasons:

- First of all, it provides services to the product owners with custom validity rules applied to the edited models, allowing the users to early discover and fix problems, lack of information, or inconsistencies. It also helps the users to focus on capturing the essential information through a graphical and user-friendly tooling, reducing the requirements writing activities to the core information to be delivered to code developers.
- As stated previously, it should provide the ability to generate the required project's documents, in order to allow the model to stay reliable.
- Regarding design models, the toolbox should provide custom code generators. The reason is very similar to the documentation generation's topic: if the design model does not generate the code, that is to say that the code is implemented manually, and as a consequence, the model does not represent the reality and is not reliable anymore. Generating code from models is a "*sine qua non*" condition to get the full promises of model-driven development.
- More generally, and in contrast with the traditional document-oriented process, the model-driven approach gives many opportunities to easily produce value-added information from the model that can help users to conceptualize more easily.

As a conclusion to the DSL tooling subject, the more value is obtained from the model, the higher satisfaction and acceptance factor can be provided to its users.

For the SIRIUS product line we have decided to use Rational Software Architect (IBM) as our modeling environment software, customized with a series of update sites that implement SIRIUS modeling toolbox.

4.2.3. Models are and remain the source

As already stated before, models must be the source of - almost - all other project's artefacts. If they do not, their destiny is a predictable death. On the contrary, if the model is always the source (Fig.1), it leads to an immediate benefit:

- It capitalizes reliable and valuable information, since it is the source for all word-documents and software code.
- The software documentation is always perfectly synchronized with the reality of the implemented system.
- Technical evolutions, maintenance and modernization activities can be obtained with little effort by simply modifying code generation templates accurately and regenerating the entire software.
- Once the code generation tooling exists, functional evolutions can be obtained with the minimum effort, by simply modifying models and regenerating the code.

These advantages allow short cycles of development that are perfectly fitted by Agile/Scrum development with iterative, incremental and adaptive methods.

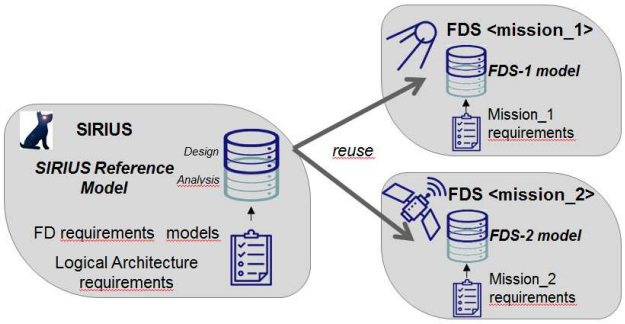

Fig. 1: SIRIUS models organization

4.2.4. Language choice

Java has been chosen for its development easiness thanks to multiples integrated environments, its large users community, its serviceability and its portability. Java is used for the 1st level (base Flight Dynamics library PATRIUS, [8]), the 2nd level (FD Algorithms, [2]) and a part of the 3rd level (FDS operational software) of the SIRIUS Product Line.

4.2.5. Pragmatic modeling

We do not model every detail of FDS software, because models must be kept simple and handy. For instance, we do not model algorithms details but only algorithms interfaces (in/out data analysis), and the way algorithms call each other (dependency analysis). Given this simple model, code generators are able to provide automatically all the required code about algorithms, except Java operations written manually by code developers.

More generally, every single piece of information captured by the models must be balanced in order to ensure it is useful to produce project artefacts, and that the effort of capturing the information is globally valuable.

When implementing a model-driven environment, it is not possible to anticipate all the concrete problems that may arise

during the project lifetime, nor define the exact required DSL vocabulary (metamodel) at the first iteration. As our understanding of the problem evolves, the required underlying metamodel must take new aspects or details into account. Some of these evolutions can be done without impact, and some of them will require model migrations. Because models are fully-structured information, this can generally easily be achieved with some throwaway migration program.

4.3. SIRIUS product line variability management

The model-driven orientation provides some advantages regarding the subject of variability management. Indeed, it allows decoupling different variability events into issues that can be managed separately:

- In respect with architectural concepts, mapping rules to transform model artefacts into source code are captured in specific code generators. Since software components are massively generated, it is possible to apply architectural evolutions to all existing products with simply modifying code generators and/or models. This is a real advantage in comparison with traditional product lines where maintaining an architecture across many products is often an issue, as shown in Ref. 3,5).
- FD functionalities are capitalized in models, and can evolve easily through simple model edition. Thanks to code generation, the evolutions are automatically implemented and propagated to all software components.
- Each generated component can be extended with additional manual code, for FD algorithms details with equations specified in dedicated requirements documents. Actually, the technical architecture is designed to facilitate the generated code manual extension wherever needed [6)].

5. Overall development process & artefacts

The SIRIUS model-oriented development process deals with the following main artefacts:

- Models for specifications and requirements.
- A “SIRIUS Reference Model” hosting a set of reusable off-the-shelf FD components. The Analysis phase defines the available shared FD interfaces, while the Design phase lists FD services implementations for these interfaces.
- FDS models, specifying which FD functionalities should be reused and providing FD service assembly configurations that meet mission specific requirements.
- GUI models.

5.1. Dealing with requirements

Each new orbital mission comes with its own project requirements to be covered by the FDS development. It is then necessary to take into account FDS mission requirements, since the beginning of the product life cycle.

In the SIRIUS product line, we have decided to import as requirements models the high level applicable documents. It is then easier to produce in the same modeling environment a work of precise traceability between mission requirements and reused FD services provided by the SIRIUS product line.

This traceability is facilitated with the custom tooling that is able to identify changes in high level requirements from one edition to another, and that is able to produce coverage and reusability documents to justify how the mission needs are fulfilled.

5.2. Modeling reusable components

The reusability of components is considered at the very beginning in our analysis models. Each identified service step in an algorithm is marked with a reusability status, which states if the discovered service is expected to be fully reusable, or if it will require various implementations depending on orbits, satellite platforms or mission.

Additional mechanisms are provided by the “logical architecture” which allows to transform some “variable” identified service steps into fully reusable components (in return with some details injected during the assembly phase of the service).

Having potential reusability in mind during the analysis phase allows the specification of algorithms that organize their intrinsic variability.

5.3. Modeling analysis: the functional needs

The main goal of the analysis model is to identify the top-level thematic functionalities required for the realization of the FDS based on the SIRIUS product line. It represents our way to express the product line feature model.

In order to make these top-level functionalities emerge, we start by defining the list of use cases FD software should fulfill (e.g. estimate the spacecraft orbit; compute the station keeping maneuvers strategy…). The goal of the use cases is to identify, specify and organize the functionalities of the FDS from the point of view of their users (Fig. 2).

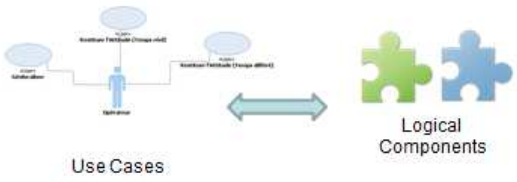

Fig. 2: Use cases and Logical components definition

As a result of this analysis, a set of “Logical Components” is identified. These components are considered as our product line “Feature Model”.

As shown in Figure 3, for each “Logical Component”, we decomposed with the help of UML activity diagrams (Fig. 3)

- high level algorithms with lists of subcomponents necessary to perform its computation,
- input and output data,
- data transfer between the different components.

We especially identify how data are consumed and produced in the algorithm process. Each identified algorithm step also identifies a service interface that can lead to various concrete interchangeable implementations in the design model.

It is important to note that the aim of this process is not to

detail the algorithmic details of each component but to identify its building blocks from the point of view of reusability.

Once the analysis phase for a “Logical Component” is complete, the design phase starts with the realization of this component into an actual FD service that will be used in the operational FDS.

5.4. Modeling design: FD service implementations

The design phase is aimed to specify the implementation details of the thematic components and sub-components defined in the analysis model.

Starting from a “Logical Component” in the analysis model, our DSL custom toolbox allows us to define its corresponding counterparts in the design model:

- An interface that defines an operation with the same inputs and outputs defined in the analysis model.
- The dependency towards the interfaces of the subcomponents declared in the analysis model.
- Each interface may define a set of parameters that is common to every service implementing it.
- As explained before, changes in the features model are detected and fixed in the design model with the help of a dedicated tooling.

For each one of these interfaces several completely interchangeable FD services implementations may be defined. A key feature provided by the logical architecture is the possibility to define, for each implementation of a given interface, as many specific parameters as needed in order to answer to its own algorithmic specificities.

In order to facilitate the identification of the target implementation that meets the requirements for a given mission, each interface can define some segmentation parameters (orbit type, types of tracking measurements…). Every service implementing this interface must declare the target values for those segmentation parameters.

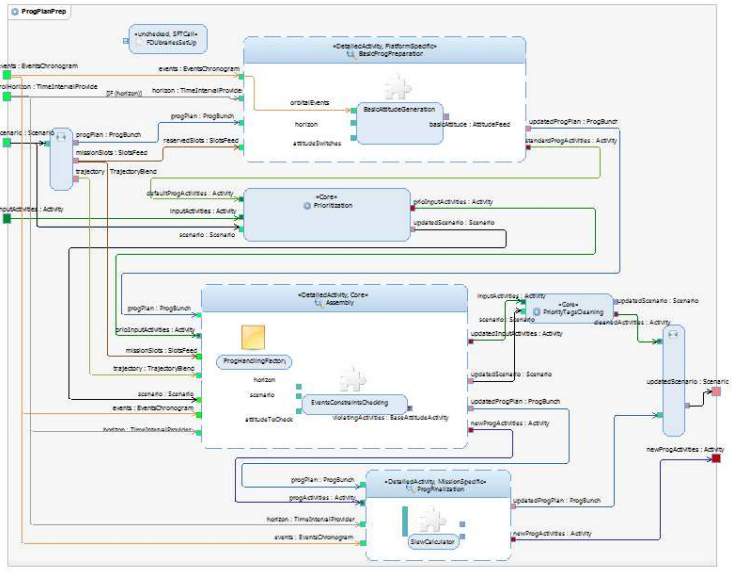

Fig. 3: Example of Feature Model for Guidance and Programming

This design process is completed with technical requirements that allow us to specify items that exceed the scope of the model: implementation details, algorithmic schemes, performances and validation strategy. These technical requirements are taken as an input in the FD algorithms development,[2].

5.5. Analysis-Design synchronizations

The analysis part of the SIRIUS Reference Model indirectly participates in the code generation, since it defines the FD interfaces that are implemented by real software components in the design part. The analysis model defines a contract that FD components commit to fulfill.

As a consequence, FD services implementations in the design model need to be synchronized with the analysis specification and both models need to be consistent with each other, even if they evolve. We implemented a specific tooling that detects the potential inconsistencies, and helps with applying the necessary corrections.

5.6. Modeling the datatypes

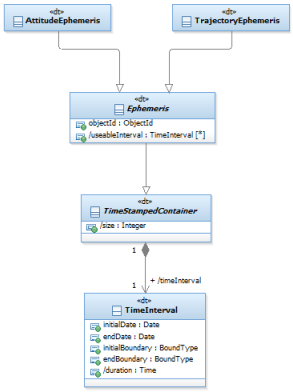

Fig. 4: Example of Ephemeris data

A central process in both the analysis and design of the FD services is the definition of the specific data structures that are to be manipulated inside each service and exchanged between them (Fig. 4).

Our DSL custom toolbox extends the generic concepts (class, attribute, inheritance…) in order to support our specific needs in terms of validation rules, physical quantities or data volume.

Amongst all datatypes defined in our model, there is one that deserves particular attention. The *scenario* is the core data of the FD services and FDS as it contains the models to compute the spacecraft dynamic state (trajectory, attitude, maneuvers, propellant mass…) over its complete timeline. This datatype is thus used and updated by most of the components in an FDS.

The model is used to generate the interfaces and one of the implementations of these datatypes. In fact, due to specific requirements in the different phases of the development process, we currently use two distinct implementations of the datatypes. During the functional validation and pre-integration of the FD services, a completely generated implementation based on EMF is used, whereas during the integration of an FDS a different implementation is used because of different design choices. Therefore, the code of the FD services may only manipulate each datatype via its interface, guaranteeing

that distinct datatypes implementations are totally interchangeable.

5.7. Modeling the datatypes widgets

In addition to the data structures, we also use the model approach to design the set of widgets associated to each datatype. The model being the common source for both the datatypes and the widgets, it allows us to assure their consistency.

We consider a widget as the elemental artifact used to graphically represent a datatype. Like datatypes, their associated widgets are provided by the SIRIUS product line and are reusable from one mission to another. The modeling of the widgets is supported on a set of basic graphical elements (textbox, combobox, date, table…) used to represent the elementary attributes of datatypes. The widgets associated to complex datatypes are constructed from these basic building blocks and the widgets defined for low-level datatypes.

5.8. Modeling an FDS

The final objective of the SIRIUS product line is to provide the means to build operational FDS.

Just as for the SIRIUS product line, the FDS model is divided in two different parts: analysis and design.

In the analysis part of the FDS model, we select from the SIRIUS Reference Model the logical components to reuse in order to fulfill the mission.

In the design part, we define the assembly specification for each of these logic components. The FD Assembly configuration consists in choosing the FD services from the design part of the SIRIUS Reference Model for each interface defined in the component. In order to facilitate the selection of these FD services amongst all the implementations made available by the product line for a given interface, the target values of the segmentation parameters defined for the mission act as a filter used by our DSL custom toolbox to propose only the FD services from the Reference Model compliant with the segmentation values defined for the mission (metadata attached to each implementation that help to decide if it is suited to the product specific requirements,[7]).

Actually, as a “Feature Model”, the FD Assembly can be represented with a tree of service interfaces, each of them requiring a binding with an implementation for the feature to be executable. It allows raising the abstraction level in a way that this configuration process can be carried out by Flight Dynamics experts instead of experienced developers,[3] since segmentation parameters are well defined.

The FDS design model also contains the specification of the operational sequences of the FDS components for ground segment automation in order to fulfill the operational needs of the mission.

Finally, based on the same metamodel used to specify the widgets, the model is also the source for the specification of the graphical user interfaces (GUI) of the FD applications that compose the FDS. The modeling of GUI consists mainly in choosing the widget used to represent each input, parameter and output declared in the assembly of services and in deciding how to organize the layout of these widgets : number of tabs, sections, columns... This approach has allowed our flight dynamics experts and operators to get highly involved since the early specification of the GUIs and work on a model whose graphical layout is very close to the desired in the final product.

6. SIRIUS model-driven artefacts

The SIRIUS model-driven product line is designed to produce numerous artefacts, from documentation to software code.

6.1. Code generation

The Design part of the SIRIUS Reference Model is used to generate in Java the code of:

- FD Interfaces: services, data (100%)
- FD services (almost 100%, except an operation to implement manually).
- Data implementations (100%, EMF based).
- Graphical widgets to represent the SIRIUS datatypes.
- Data editors that help us to define validation data for pre-integration activities and to test the developed FD services.

An FDS model (Fig. 5) is the input to generate:

- FDS service assemblies.
- Automatic Sequence of assemblies.
- GUIs for the FD applications.

We do not have yet the final code generation rate, for a deployed product, since our first product will be integrated and delivered later this year. But we already use and test extensively all our product features, which represent the biggest part of the application code (FD Data, FD services, FD Assemblies). For this product core scope, the source code corresponds to 3 million lines of code, automatically generated at 93%.

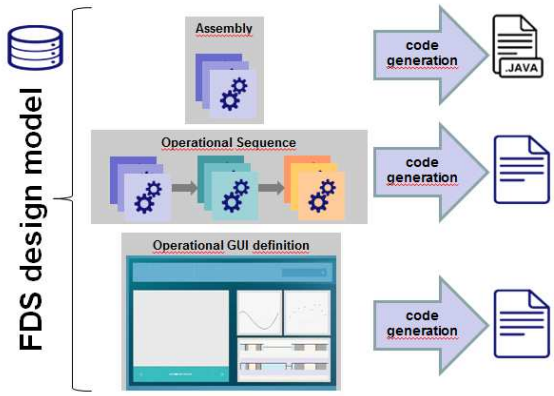

Fig. 5: Code generators from FDS design model

6.2. Documentation generation

Each model can lead to documentation generation. We implemented more than 20 documentation generators in the SIRIUS product line, from regular requirements specifications to design documents or requirements traceability matrices.

For now, about 150 documents are generated for the project, with a total of 20000 pages of documentation.

6.3. Automatic migration of validation data

As stated before, we generate datatypes implementations in order to support a set of activities (FD services development and validation, pre-integration activities for FDS software).

Evolutions in SIRIUS datatype models will lead to evolutions in the underlying java classes, and *in fine* lead to impacts on the validation data. Because FD services are implemented using an Agile methodology, we generate new versions for the totality of the SIRIUS code every month, and consequently we need to carry out a validation data migration to take the changes into account.

We developed a custom tooling to assist this migration, which is able to compare two versions of our datatypes implementations, and generate a fully operational migrator from the comparison. Few migration cases will eventually require a complementary manual coding to succeed in the migration, but it is unlikely. Most of the generated migrators should work without any manual intervention.

7. Conclusion

In this paper we described the choices made for the SIRIUS model driven product line, by avoiding the risks of GML for our specific Flight Dynamics technical domain, by giving a crucial importance to the model as a reference and by considering the code generation and the document generation as an important part of the product life cycle.

This methodology has already reached its final goal with the FDS definition of the first defense missions, and is beginning for next missions as SWOT.

References

1) Amir Molzam Sharifloo: *Requirements verification of Variability-Intensive Systems*, Politecnico di Milano, 2014.
2) Iván Llamas, Yannick Tanguy, Michel Lacotte, Jean-Jacques Wasbauer, SIRIUS-DV: *The new Flight Dynamics algorithms for the future CNES missions*, ICATT, 2016
3) Devesh Sharma, Aybüke Aurum, Barbara Paech: *Business Value through Product Line Engineering – A Case Study*, 2008
4) Faheem Ahmed, Luiz Fernando Capretz, *A Framework for Process Assessment of Software Product Line*, Journal of Information Technology Theory and Application, p138-143, 2005.
5) Sami Ouali, Naoufel Kraiem, Henda Ben Ghezala, *Framework for Evolving Software Product Line*, International Journal of Software Engineering & Applications (IJSEA), Vol.2, No.2, April 2011.
6) Andreas Metzger, Klaus Pohl, *Software Product Line Engineering and Variability Management: Achievements and Challenges*, June 2014.
7) Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, *Beyond Boolean Product line Model Checking:Dealing with Feature Attributes and Multi-features*, International Conference on Software Engineering, June 2013.
8) Houdroge R., Claude D., Anton J., Sabatini T., Cardoso P., Mercadier G., Trapier T., Tanguy Y., *The Sirius Flight Dynamics Library for the next 25 Years*, ICATT, 2012