JAVA MULTI-MISSION SIMULATION FRAMEWORK: EVOLUTIONS AND IMPROVEMENTS (SESP) 2015

Pierre Bornuat [(1)], Thierry Warrot [(2)], Olivier Podevin [(1)]

[(1)] *CS Systèmes d'Information (CS SI), 5, Rue Brindejonc des Moulinais - BP 15872 - 31506 Toulouse Cedex 05 - France, Email: pierre.bornuat@c-s.fr / olivier.podevin@c-s.fr*

[(2)] *CNES, 18, avenue Edouard Belin - 31401 TOULOUSE cedex 9 – France, Email: Thierry.warrot@cnes.fr*

ABSTRACT

CNES (*Centre National d'Etudes Spatiales – France*) has a large mission engineering experience accumulated through different Earth Observation missions. Turning towards the future, CNES has invested in a new Java-based Mission Simulation framework. This framework, called ALIS, which stands for "*Atelier Logiciel pour l'Ingénierie Système*" (*Simulation Framework for Mission Engineering*), models a mission simulation loop which is composed of : mission parameters, ground and in-flight constraints such as number and location of images to capture, orbit, on board memory capacity, number and location of TM (*TeleMetry*) ground stations, platform agility capacities, etc., in order to assess the performances of the global acquisition / reception of the acquired images. Current development and maintenance of ALIS framework is held by CS SI since 2011.

This kind of simulation, which addresses large data management and computer time consuming, is a major concern.

This paper presents the different capabilities such a framework offers and how it is and will be used at CNES. Moreover, among recent technical concerns, two of them have required particular attention and design efforts:

- the computation capacities strengthening for high CPU-intensive simulations,
- the need to rapidly incorporate new functionalities to satisfy a wider scope of potential users.

1. INTRODUCTION

After an introductory focus on how mission simulation is understood and applied at CNES (*Centre National d'Etudes Spatiales – France*) and on the framework architecture of ALIS (*Atelier Logiciel pour l'Ingénierie Système - Simulation Framework for Mission Engineering*) to detail technological topics, this paper highlights two technical topics showing:

- how, from a standalone simulator's infrastructure, ALIS has moved to a distributed architecture, in order, in particular, to run simulations on distributed environments (client/server clusters, etc.). Intrinsic problems and obvious technical profits are detailed;
- how the use of COTS (*Commercial Off-The-Shelf software*) with a high readiness level offers a large flexibility and enrichment of the framework.

As a conclusion, the paper draws the full benefits such a framework can offer to entities dealing with end to end mission performances.

2. MISSION SIMULATION AT CNES

2.1 Context

Earth observation systems are composed of satellites, ground stations, networks, etc. Once launched, a satellite flies imperturbably on its orbit, but nowadays, satellites are "agile", which means they can change their orientation (not their trajectory) quickly to take pictures.

Few of them can compute by themselves orientation and time to take pictures, but this is generally done on larger computers, on ground, in the "operational mission planning software" hosted at the "Mission Programming Centre". This kind of software must be strongly robust to be operational 24 hours a day, 7 days a week.

On the contrary, mission simulation software are study software which aim at simulating the "operational mission planning software" before it is actually built. Therefore, they do not need to be as robust as the operational software, but they need to be **versatile** to allow trying and testing many algorithms and parameters, in order to define, with the System customers, the best optimized set of options.

2.2 Taking shots from space

Contrary to a basic camera, which is based on a CCD matrix (*Charge-Coupled Device*), a satellite's camera is only a single line of CCD detectors. A picture is taken by, first switching "on" the CCD line, second by leaving the satellite move forward on its orbit, and last by switching "off" the CCD line. In true life, in order to get high quality pictures, satellite orientation must continuously and very precisely be controlled during the shot, which needs very complex mathematic computations. However, the principle remains as described above.

Thus, to take a picture with a satellite, one must define the satellite orientation, when to switch "on", when to switch "off" and how to rotate in between: and do so for each taken picture. This corresponds to the “Acquisition Kinematics Plan”.

Satellites now hold a large number of pictures in memory but the latter is limited. It is therefore necessary to download these pictures on ground stations. But if a satellite is on the opposite side of the Earth from the station, it must wait until it arrives above the ground station to begin downloading pictures. Thus, “mission planning software” must compute all downloads’ begin and end times: this constitutes the “Acquisition Download Plan”.

These kinds of plans must be uploaded to the satellite when it passes above a ground station to prepare the satellite’s work for the next day.

2.3 Mission Programming engineer's tasks

We just talked about how to take pictures. But what picture shall we take first? What if the region to observe is larger than the CCD line? If there are lots of pictures to take, which is always the case, what is the optimized scheduling to take them, according to their number, their priorities and taking into account customers’ rights? Is there still enough free memory and electrical power on board to make a shoot or is it time to point solar arrays to sun direction? What if there are clouds on the interesting part of the picture? What is the average delay between a picture request deposit and its obtaining? How many 3D pictures can be taken during 6 months?

All these questions must be answered by mission programming engineers and/or implemented in the final “operational mission planning software”.

Hereafter are some other goals a mission programming engineer has to handle:

- Before developing the “operational mission planning software”
    - Organize experimentation campaigns with customers, to specify needs and present the system and its functioning (ex: principles and tuning of the system sharing rules, negotiation principles, prioritization helpers, etc.),
    - Study and find the best algorithms and parameter set,
    - Size the system (ground networks, satellites’ configuration, satellites’ agility, etc.),
    - Generate mission contexts to check provisioning (AOCS - *Attitude and Orbital Control Subsystem*, power…),
    - Estimate mission performances (capacity on a specific region, capacity on one orbit, etc.),
    - Estimate temporal performances,
    - Teach future users how to use the System,
    - Communicate or advertise about the System (using videos showing the system while working).
- After developing the “operational mission planning software”
    - Validate that each service and function of the “operational mission planning software” calculates correctly. Simulator is, in this case, used to produces reference data and compare them with the one computed by the operational software),
    - Check produced mission plans (through cartography and 3D visualizations, chronogram displays…).

Combination is so enormous and complexity so high that, in order to perform his work, the mission engineer needs to use mission planning simulators.

2.4 Mission planning loop

The following diagram (*Fig. 1*) summarizes the main stages of the “mission planning loop” which are scheduled everyday (sometimes several times a day) to prepare the satellites’ work plans for the next few days.

There are mainly two programs: Users’ interacting software and Satellites’ working plans computing software.

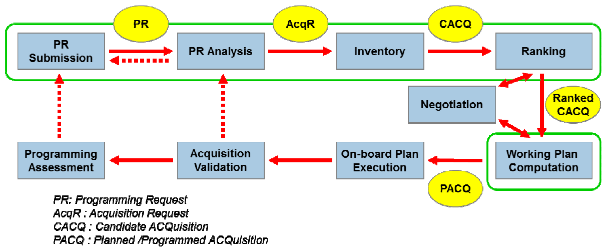


Fig. 1 – Mission Planning Loop

The first step of users’ interacting software consists in the “**Submission**” of zone imaging requests by users.

Each zone is then “**Analysed**”, i.e. virtually cut in small rectangles (AcqR: *Acquisition Request*) whose width corresponds to the CCD line width.

For the few next days, satellites orbits are well known. Thus it is possible to collect all AcqR underlying satellites trajectories. The third stage, “**Inventory**”, builds this AcqR list and computes time slots available to take pictures according to constraints: for instance, a same AcqR must be acquired several times with precise geometric constraints in order to provide 3D pictures.

The fourth stage, “**Ranking**”, can be done manually and/or automatically. It consists in sorting the candidate acquisitions to meet users’ priorities. These sorted lists are then sent to the **satellites’ working plans computing** software (fifth stage).

All that is left is then to upload working plans on board, during next ground station flying over.

2.5 What is a mission simulator?

What has been presented in the previous paragraphs corresponds to “operational mission planning” software.

But within a simulator, many other tasks shall be performed: simulate electric and memory consumption, simulate picture rejection according to cloud probability, compute statistics about overall acquisition performances on several months, etc.

All these computations shall be done as quickly as possible. We do not need real time simulation (which would take several months due to satellites’ speed), but accelerated time.

Note that in this kind of simulator, we assume that satellites do exactly what they are intended to do. We do not try to take into account AOCS inaccuracies and do not simulate on board software. Thus, only topics impacting overall system performances such as memory and power are taken into account.

3. ALIS INFRASTRUCTURE AND MAIN CHARACTERISTICS

3.1 Project's figures

ALIS framework covers about **23 Use Cases**, ranging from simulation objects and stages configuration, simulation stages execution, Acquisition Requests deposit and analysis, manual prioritization of Candidate Acquisitions, Acquisition Plan computation and visualization.

ALIS is defined by about **1170 requirements**.

Final product is composed of about 3 200 Java classes, 25 000 methods, **500 000 code** and comments **lines** (excluding COTS).

Validation efforts rise up to approximately **300 validation tests**.

3.2 ALIS technologies

On the technological side, ALIS is a Java/Eclipse RCP based platform which uses a large number of Open Source components as WorldWindJava (NASA’s cartography component), Hibernate & hSQLDB (relational databases), Xstream, Spring, commons.math Apache library, JfreeChart, etc.

3.3 ALIS architecture

ALIS architecture is designed to separate generic functions and domain specific ones. On Fig. 2, the blue set holds “Flight Dynamics” specific components and the yellow set "mission planning” specific ones. The orange set contains generic components that may be used "as is" to build a non "mission planning specific" simulator. If a new simulation domain which has to be shared appears, and several simulators are planned to be developed in this domain, a new violet set could be added to ALIS.

This being said, ALIS main generic components are: the "Base" that handles data configuration files, SGO and MDF which are generic GUI (*Graphical User Interfaces* - see focus on next paragraphs), FDS library, cartography wizards (display orbits, ground station visibility circles, clouds, etc.), 3D animation (data preparation for 3D animation in CNES VTS viewer).

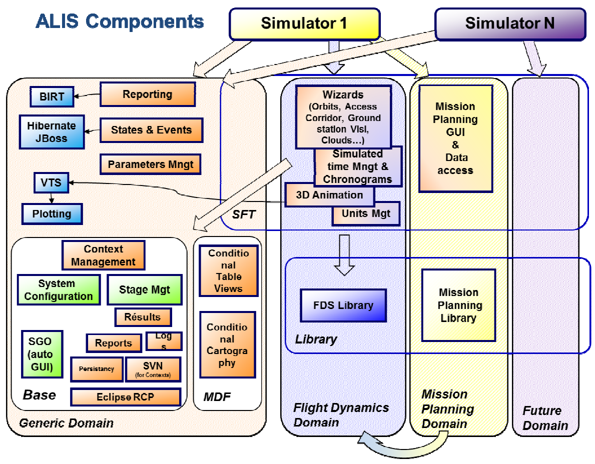


Fig. 2 – ALIS components architecture

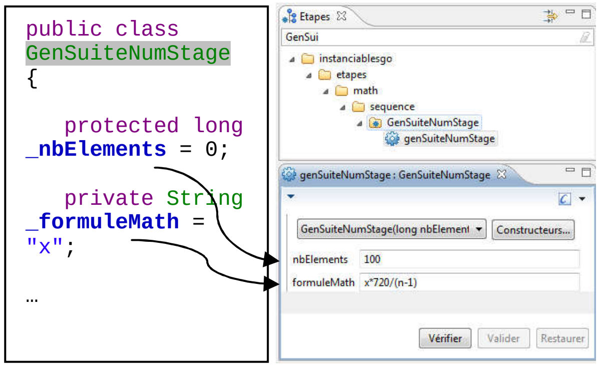
3.4 ALIS versatility

This paragraph makes a focus on SGO and MDF, two major features which provide high versatility to ALIS.

These two features were created by Capgemini and completed by CS SI.

3.4.1 SGO

The SGO (*Show Generic Objet*) component allows automatic generation of GUI directly through source code inspection as seen on the following example:



```
public class
GenSuiteNumStage
{

   protected long
_nbElements = 0;

   private String
_formuleMath =
"x";

…
```

nbElements 100
formuleMath x*720/(n-1)

Fig. 3 – SGO automatically generates GUI

The advantage of this feature appears when you want to quickly switch between two algorithms. To simplify, imagine a stage that performs an arithmetic operation. The operation can be an addition or a product. In such a case, the Strategy design pattern advises us to define an Operation Interface implemented either by an Addition or a Product class. In this case, SGO will build the folowing GUI (*Fig. 4*):

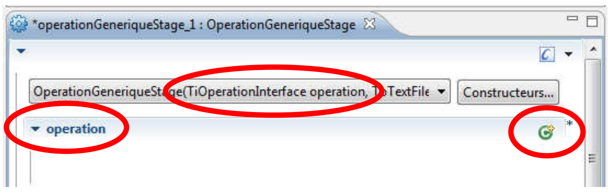

*operationGeneriqueStage_1 : OperationGeneriqueStage
OperationGeneriqueStage(TiOperationInterface operation, ToTextFile
Constructeurs...
operation

Fig. 4 – SGO: stage definition needs an operation which may be...

… defined by clicking on the green 'class create' button. This leads to a pop-up allowing to choose one among all simulator's classes implementing TiOperationInterface:

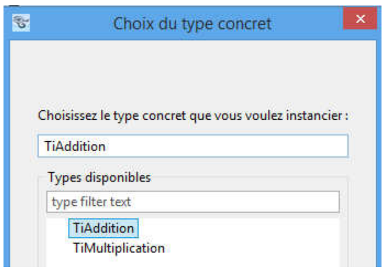
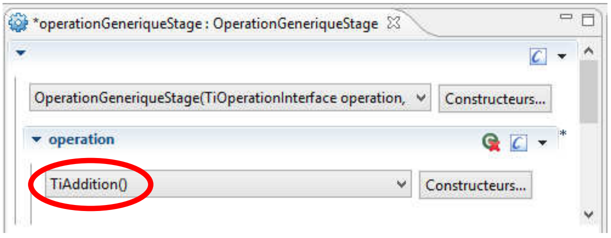


Fig. 5 – SGO: operation implementation selection

The stage is now configured to perform an addition.


*operationGeneriqueStage : OperationGeneriqueStage
OperationGeneriqueStage(TiOperationInterface operation,
Constructeurs...
operation
TiAddition()
Constructeurs...

To perform a product, you just have to replace Addition with Product (Multiplication in french) and restart the run.

This example is trivial, but if you replace OperationInterface by ZoneClippingInterface, Addition by SouthNorthClipping and Product by AlongSatelliteTrackClipping and you will understand the power of this feature.

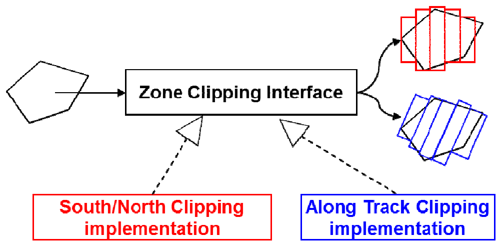

Zone Clipping Interface
South/North Clipping implementation
Along Track Clipping implementation

Fig. 6 – SGO: the power of quickly replacing implementations

3.4.2 MDF

The second key feature of ALIS is MDF (*Model Driven Framework*). This component allows users to define themselves conditional formatting, filtering and sorting in tables and cartography, through two automatically generated configuration wizards:

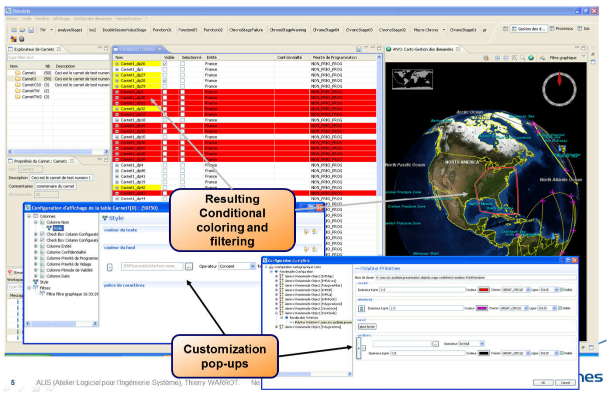


Fig. 7 – MDF: user autonomously customizes displays

4. IMPROVEMENT OF COMPUTATION CAPACITIES

4.1 An increasing need for high performances…

Simulations become more and more CPU-consuming (*Central Processing Unit*), as algorithms tend to model more complex systems and situations, and process larger volumes of data, as computers' capacities increase permanently.

ALIS simulation framework was initially designed as a standalone simulator's infrastructure. What has decided CNES to reconsider ALIS architecture originated from the SSA (*Space Surveillance and Awareness*) simulator project, conducted at CNES.

SSA simulations use a very large number of space objects (more than 20 000, orbiting around the Earth), a wide quantity of orbitographic data (ephemeris, optical measurements, etc.), SSA algorithms (collisions and fragmentations detection), and synchronous and asynchronous processes integrating these algorithms as well as surveillance sensors scheduling.

Consequences are:

- Powerful computation capabilities are required to absorb CPU-consuming algorithms, which are to be executed online or on a batch mode, either on a single PC, or on a Linux cluster;
- Several different databases are required to manage the different data types (reference space objects, simulated space objects, sensors, etc.).

Similarly, other simulators in the field of Earth observation, based on ALIS, such as mission simulators developed for French Defence, tend to process larger lists of image acquisition requests, and implement more complex mission algorithms (analysis, ranking, mission plan computation, etc.).

4.2 …Resulting in a deep architecture re-engineering

In order to satisfy computation needs, it has become necessary to perform deep modifications to the existing ALIS architecture. As said before, ALIS was initially designed as a standalone application, including GUI, data access and server layers in a single process. Moreover, the simulation execution engine was originally highly embedded with the GUI functional layer; which has widely impacted on modifications required for the infrastructure.

ALIS brings many useful features to the SSA simulator, such as the capacity to link synchronous and asynchronous processing. However, up to now, ALIS has not provided the possibility to run simulations on a cluster, possibly using a command line mode; all simulations executions were possible only locally and via a graphical user interface. Moreover, the SSA simulator needed to be able to disconnect the GUI while still running simulations on cluster, as these simulations may last much longer than the authorized duration before simulator GUI is disconnected from the cluster host. This major constraint has conducted to propose a major architecture re-design: moving from a standalone process to a distributed application, including separation of the database server from the standalone application. These changes have been designed to satisfy SSA simulator requirements, but are also applicable to existing and future Earth Observation simulators, bringing them potential performance improvements.

4.3 New ALIS distributed architecture

A simulator based on ALIS shall run 1) on a single laptop, when simulation design is the main concern, 2) on a standalone PC, located in a cluster, when specific simulation processes have to be dispatched on several processors, or 3) on a distributed hardware platform, when computation performance is sought without benefitting from a cluster's computation power.

Moving to a distributed architecture implied mainly to solve the question of inter-process communications, and therefore to select a technology adapted to existing technical environment (Windows/Linux), which would also be acceptable with respect to development costs.

Rapidly, RMI (*Remote Method Invocation*) has been retained as the best solution, due to its high level of integration with java. Other technologies were considered, mainly CORBA (*Common Object Request Broker Architecture*), but were finally rejected as they less matched to technical requirements.

Originally, ALIS execution daemon, which runs simulations based on a simulation context and a Simulation Object Model (SOM), was included in the same process as the user interface, as shown on *Fig. 8*.

ALIS architecture re-design was made such that execution definition and planning would remain on the GUI side, and execution itself would be done on the server side.

Allowing distributed execution of simulations consisted mainly in creating separate processes into which ALIS execution daemons could run. Exchanges between ALIS GUI process and ALIS server processes are based on RMI. However, ALIS architecture assumes that a shared storage is available through NFS to enable different processes to load context data using coherent paths.

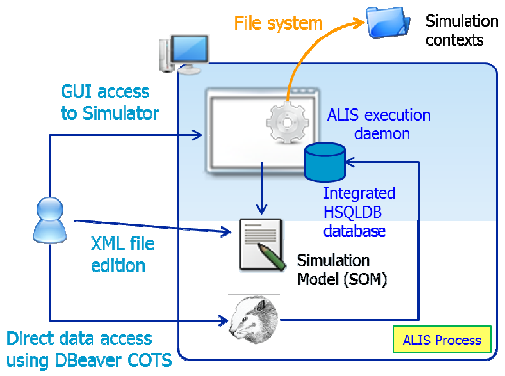


Fig. 8 - Original ALIS architecture (simplified)

Next paragraph provides more details on processes architecture and interactions. Fig. 9 also shows possible use of command line accesses to execute and supervise simulations, and clear separation of database server from MMI process.

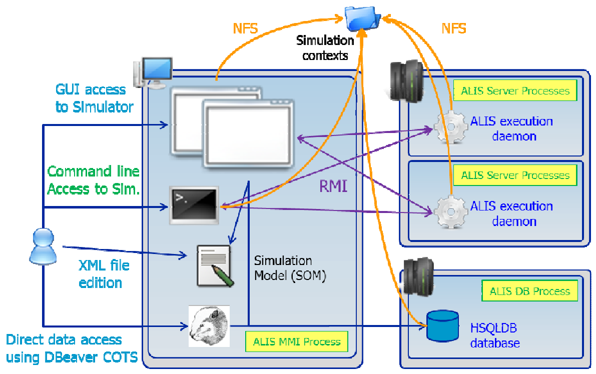


Fig. 9 - ALIS distributed architecture (distributed hardware configuration)

Fig. 10 shows how ALIS distributed architecture is transposed on a single PC. The only difference is that all processes are hosted on a single machine; this allows ALIS simulator's use on a laptop without requiring a different architecture.

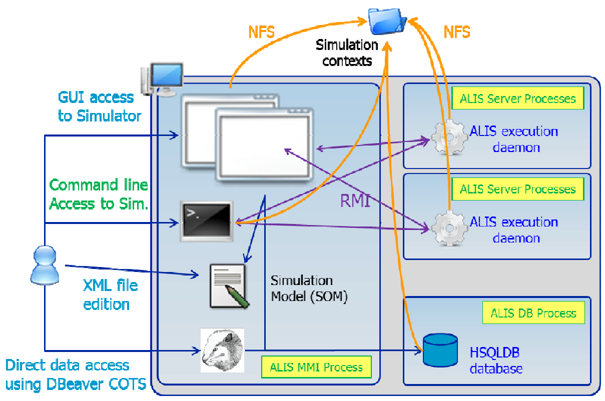


Fig. 10 - ALIS distributed architecture (single PC)

4.4 Close-up on processes interactions

On the server side, three kinds of processes are implemented:

1. RMS process (Runtime engine Manager Service): this process is always associated with a single context. It executes all processing in separated threads, one per execution request proceeding from a client (command line or GUI process).
2. DMS process (Daemon Manager Service): this process offers the remote services front-end for the GUI process. It is in charge of launching the RRS process if it is not running. It is in charge of launching an RMS process for each context opened if such a process does not already exist.
3. RRS process (RMI Registry Service): this process contains the RMI registry in which are registered: the registry itself, the Daemon manager service (DMS) and one entry for each running Runtime engine manager service (RMS).

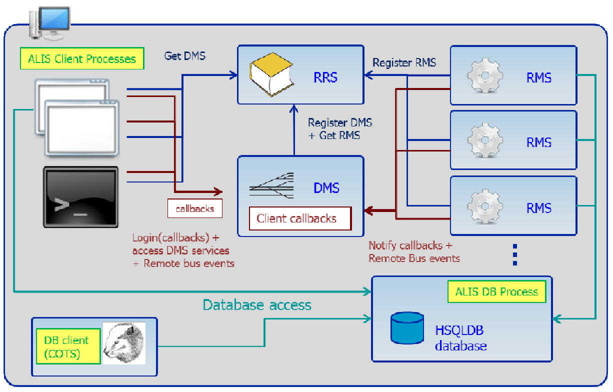


Fig. 11 - ALIS inter-process communication flows

Process synchronization: ALIS execution daemons isolation required a deep and complete analysis of data exchanged between the different processes, and how data should be synchronized. One constraint was in particular that data exchanged via RMI should be serializable, which was not the case for all concerned data, because of their intrinsic specification.

Data synchronization between processes has been achieved using specific mechanisms. In a first implementation, these mechanisms relied on the particular methods of marshalling and unmarshalling used to store and load data models.

This implied that data model was loaded by both processes when launched. Then, if the model was modified in one process, it stored the model on disc before sending a RMI notification to the second process. Then the second process reloaded the model from shared storage. A problem with this approach is that the model save and reload are time consuming, even if the model is quite small, because marshalling was customized to produce highly human-readable XML format files.

A second implementation has then been developed to increase synchronization performances. This new mechanism is able to marshal any data model into an intermediate object which is always serializable. This serializable object is then transmitted to another process through RMI. This second process then unmarshals the intermediate object into the data model using reverse automatic mechanism. This second approach is far more efficient than the first one, and does not rely on any specific data model marshalling/unmarshalling mechanism.

This second approach having the advantage to be totally generic, it has been applied since then to several non-serializable data model issues.

Event bus service: an event bus service has been implemented in ALIS distributed architecture, to complete standard Eclipse listeners. Each process contains such a bus. Moreover, these event buses are interconnected to transmit specific kinds of messages from one process to another. These remote events are typically log messages, but also remote notifications replacing direct calls from the non-distributed architecture.

4.5 New / changed functionalities related to simulation distribution

Externalization of database server: as introduced above, the database server needed to be separated from the GUI process, in order to allow it to run after simulator GUI was disconnected. This has been introduced in ALIS distributed architecture, along with server management functionalities (start, restart, stop). As some issues have been encountered under Windows OS with shutting down the Database server, a specific development had to be done to bypass this limitation. The DB process, containing the DB server, now contains a second server which can receive external queries to shut down, restart or get the current DB running status.

Simulation executions supervision: before switching to ALIS distributed architecture, processing supervision was done using Eclipse API and in particular Jobs and ProgressMonitors. These core elements still remain in the distributed architecture to avoid a major refactoring, but they have been adapted as processing supervision is done in GUI process and processing itself is done in an RMS process. Using the same core Eclipse elements in the new architecture implied to 1) duplicate the supervision services layer in both GUI and processing processes, 2) create on-the-fly proxy Jobs on GUI process while creating the real Job on the processing process, 3) create on-the-fly proxy ProgressMonitors on the remote processing entity while creating the real ProgressMonitor on GUI process and 4) transmitting all achievement notifications and Job status from the supervised processing to the supervising GUI process.

4.6 Technical issues

RMI in an Eclipse-RCP / Spring context: several technologies used in ALIS software such as RMI, Spring, XStream and more usual ones such as standard resources access are not Eclipse plugin-based and rely only on Java classpath mechanism to solve access and class dependencies.

As ALIS software is developed under Eclipse in an OSGi environment (Open Services Gateway initiative), all dependencies are solved using OSGi Manifests and all these technologies can raise class and resource access issues due to a lack of interoperability between these two dependencies resolution paradigms.

Fortunately, Eclipse provides a mechanism to bypass this issue. Indeed; in an Eclipse Manifest for plugin A, if we declare plugin B as a “Registered Buddy”, then Plugin B gains visibility on plugin A. Few other conditions also apply to make it all work.

The main issue here has been to analyse which plugins were involved in the access problem, as classloaders used while the exception occurs are not always those of the class in which the exception actually occurs.

Dynamic services management via injection: cross-functional services are only visible from the final simulation application and not from the underlying ALIS framework. A problem is that the RMI distributed services layer is provided by ALIS framework and not by the simulation application which is specific to each implementation.

To solve this issue, Eclipse API offers an implementation (named Equinox) of OSGi mechanism called “Declarative Service” which enables injecting an implementation only knowing its interface in a different way that the Spring technology does.

Configuration of these components is completely integrated into Eclipse; services are lazy-loaded (i.e. loaded only when necessary) and their life-cycle is bounded to their associated bundle’s lifecycle (the one in which they are defined).

This technology enables to extend ALIS framework remote services offered by the GUI, DMS and RMS processes, without the framework knowing anything from those components.

User Preferences’ distribution: before distributing ALIS architecture, Eclipse User preferences were used in a classic way. Distributing the architecture required to manage such preferences to give access to RMS processes. A second issue was to minimize the implied refactoring. A third issue was to provide a generic access to the preferences, whatever the process from which the query would originate. We chose to create a static front-end, available to all processes, using a Spring injection mechanism. The implementation is different for each process, as preferences are 1) Eclipse-based in the GUI process (as before), 2) based on a specific proprietary PreferenceStore implementation in RMS process (as preferences are compartmentalized for each processing) 3) also different in the command line process which can possibly get back all preferences from a GUI preferences storage set.

4.7 Near future evolutions and consequences

Current ALIS release has been tested in a single PC hardware configuration, which corresponds to initial SSA simulator requirements and needs. However, architecture design and implementation are made so that deployment on a truly distributed hardware configuration will require only little complementary coding and validation works. Indeed, ALIS is ready to use IP addresses instead of “localhost” configuration to identify server processes localization.

It is also important also to emphasize the fact that a consequence of enabling distributed simulations may have an influence on how simulation stages are designed, as well as how algorithms are coded. Indeed, distributed simulations may now be executed on a Linux cluster, thus benefitting from parallel processing. This possibility may not be applicable to all algorithms; for example, an acquisition inventory algorithm may not be easily parallelized due to its intrinsic structure. However, this possibility should be taken into consideration when designing an algorithm, as parallelization could provide significant computation power. This should also be done considering algorithm complexity, as parallelization could increase complexity too much in comparison to computation gains. Indeed, simulation is an activity which requires being able to modify algorithms easily and rapidly, depending on studies to realize and sought optimizations.

5. ALIS VERSATILITY: EMPHASIS ON COTS INTEGRATION

Either after users’ requirements or developers’ propositions, enrichment of ALIS framework has been and is still required to supply more and more new services, to ease quicker implementations of new simulators. As development cost is a strong driver, and as services quick availability is expected by users, use of COTS with a high readiness level is at the core of design concerns.

But, before handling COTS integration, a short focus is made on an important question: COTS licenses.

5.1 Licenses

COTS may be free of charge or not. They also may be open source or not. Finally, they may be libraries, RCP plugins or main programs. All combinations of these three axes are possible, even though open source software is usually free of charge.

In all cases, COTS are distributed under licence, which may have a copyright © or a copyleft 🄯. As the copyright restricts redistribution of software, on the opposite, copyleft (a play on words) encourages people to distribute creations, but it may impose obligations which may sometimes be restricting for a company. Indeed, there are three levels of copyleft:

- permissive licences (BSD, MIT, Apache),
- weak copyleft licences (LGPL, CeCILL-C),
- strong copyleft licences (GPL, AGPL, CeCILL).

Imagining a COTS A, that you modify a little to become A', and B your application using A'.

With a **permissive** licence, A' and B may be distributed under any licence (permissive or not).

With a **weak** copyleft, A' will inherit the weak copyleft, but not B. Precise conditions are given in each licence.

With a **strong** copyleft licence on A, B inherits the same strong copyleft. Thus, if A is a free open source, and you want to distribute your application B, then you must distribute it as a free open source!

Two points however are to be reminded:

1. Copyleft applies only if you distribute B outside your company.
2. In this case, you must distribute the A' and B source code **only** to recipient users (users for which software B is developed).

5.2 Levels of integration

COTS may be low-level libraries as well as high-level components with graphical user interface (GUI).

With low-level components, integration is usually quite simple, but it remains one’s responsibility to implement GUI using it... and this may represent a large effort.

Using an OSGi framework like Eclipse RCP, it becomes possible to integrate not only low-level COTS, but also high-level GUI, which allows you to fairly quickly add high readiness level services.

ALIS integrates open source COTS ranging from low-level libraries to high-level GUI. As low-level or medium-level COTS (such as XStream, Hibernate, Spring, JFreeChart, WorldWindJava) have been integrated early in ALIS development, because they were essential components, we only provide hereafter samples of high-level COTS integration in ALIS.

5.2.1 Simplest integration of an RCP plugin (SVN)

When an RCP plugin is completely ready to use, you just have to add it to the "run configuration" of your application. The plug-in will either add its own menu to the main menu or it will be accessible through the "change perspective button".

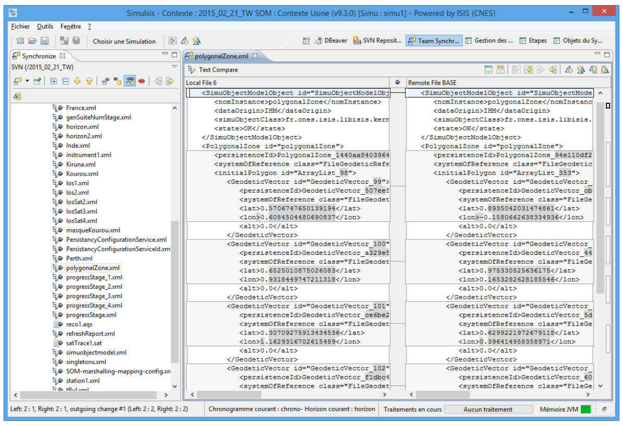
In ALIS, SVN (*SubVersion*) support has been added using this way.

SVN is a source code version and revision control system. Almost every java developer has already used the "SVN Team Synchronize" RCP plug-in in the Eclipse Java compiling environment. It allows developers to synchronize local source code with a shared remote SVN Repository.

The source code of ALIS is of course managed using this plug-in in the Eclipse IDE (*Integrated Development Environment*). But why integrate this plug-in in ALIS?

A simulator is a software that manages a lot of data files which are all related to the same simulation. This set of files (which includes several databases) is stored in a directory tree called an ALIS Context. To be able to share a Context, to allow replaying a few months later the exact same simulation or to compare the current Context with a reference Context, a version and revision control system is a good solution. Though these files are data files and not source code files, they can be handled the same way in a SVN Repository; therefore it was decided to integrate the "SVN Team Synchronize" plug-in in ALIS framework in order to be used in a simulator.

The following figure shows this plug-in used in a simulator to compare a local version of a PolygonalZone object with the one of the SVN Repository. We can easily see that coordinates have changed… without coding any source line.



Local File 6:

```
<SimuObjectModelObject id="SimuObjectModelObj
   <nomInstance>polygonalZone</nomInstance>
   <dataOrigin>IHM</dataOrigin>
   <simuObjectClass>fr.cnes.isis.libisis.kern
   <state>OK</state>
</SimuObjectModelObject>
<PolygonalZone id="polygonalZone">
   <persistenceId>PolygonalZone_1440aa8403964
   <systemOfReference class="FileGeodeticRefe
   <initialPolygon id="ArrayList_98">
      <GeodeticVector id="GeodeticVector_99">
         <persistenceId>GeodeticVector_507ee5
         <systemOfReference class="FileGeodet
         <lat>0.5706747650139194</lat>
         <lon>0.6084504480690837</lon>
         <alt>0.0</alt>
      </GeodeticVector>
      <GeodeticVector id="GeodeticVector_100"
         <persistenceId>GeodeticVector_a329e5
         <systemOfReference class="FileGeodet
         <lat>0.6525010875026083</lat>
         <lon>0.9318449747211318</lon>
         <alt>0.0</alt>
      </GeodeticVector>
      <GeodeticVector id="GeodeticVector_101"
         <persistenceId>GeodeticVector_ce6be2
         <systemOfReference class="FileGeodet
         <lat>0.30709275913434536</lat>
         <lon>1.1629316702615489</lon>
         <alt>0.0</alt>
      </GeodeticVector>
      <GeodeticVector id="GeodeticVector_102"
         <persistenceId>GeodeticVector_f1dbc4
         <systemOfReference class="FileGeodet
```

Remote File BASE:

```
<SimuObjectModelObject id="SimuObjectMode
   <nomInstance>polygonalZone</nomInstanc
   <dataOrigin>IHM</dataOrigin>
   <simuObjectClass>fr.cnes.isis.libisis.
   <state>OK</state>
</SimuObjectModelObject>
<PolygonalZone id="polygonalZone">
   <persistenceId>PolygonalZone_94e110df2
   <systemOfReference class="FileGeodetic
   <initialPolygon id="ArrayList_353">
      <GeodeticVector id="GeodeticVector_
         <persistenceId>GeodeticVector_cb
         <systemOfReference class="FileGe
         <lat>0.8935042031474861</lat>
         <lon>-0.1580662638334936</lon>
         <alt>0.0</alt>
      </GeodeticVector>
      <GeodeticVector id="GeodeticVector_
         <persistenceId>GeodeticVector_44
         <systemOfReference class="FileGe
         <lat>0.975330525636175</lat>
         <lon>0.1653282628185546</lon>
         <alt>0.0</alt>
      </GeodeticVector>
      <GeodeticVector id="GeodeticVector_
         <persistenceId>GeodeticVector_5d
         <systemOfReference class="FileGe
         <lat>0.6299221972679118</lat>
         <lon>0.396414958358971</lon>
         <alt>0.0</alt>
      </GeodeticVector>
      <GeodeticVector id="GeodeticVector_
         <persistenceId>GeodeticVector_60
         <systemOfReference class="FileGe
```

Left: 2 : 1, Right: 2 : 1, outgoing change #1 (Left: 2 : 2, Right: 2 : 2)

Chronogramme courant : chrono- Horizon courant : horizon

Fig. 12 – Comparing Context files with SVN

5.2.2 GUI COTS integration with GUI Adapters (BIRT Sample)

As seen before, a mission planning simulator may be used to produce, collect and analyse statistics about algorithms and parameters. Mission Engineers often export data manually for further analysis in Excel. As it is a good solution for short studies, it becomes boring for repeated analysis like long term simulations.

That is why it was decided to integrate BIRT (Business Intelligence Reporting Tool) in ALIS.

BIRT purpose is to update report templates according to data produced by a simulator and stored in a database, a csv file or anywhere else, in order to produce synthetic reports containing diagrams and texts (Fig. 13).

BIRT is an **optional** open source RCP plugin. This means that user can decide, at launch time, to use it or not. This allows quicker launch time and less memory consumption if BIRT analyses are not required.

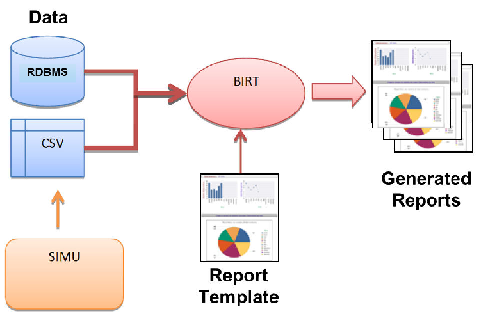


Fig. 13 – BIRT usage to produce reports

As shown in Fig. 14, BIRT is accessible in the application through a built-in Eclipse RCP Perspective. An Eclipse Perspective is a memorizing of the UI state (i.e. position of Views, visibility of Menus, etc.).

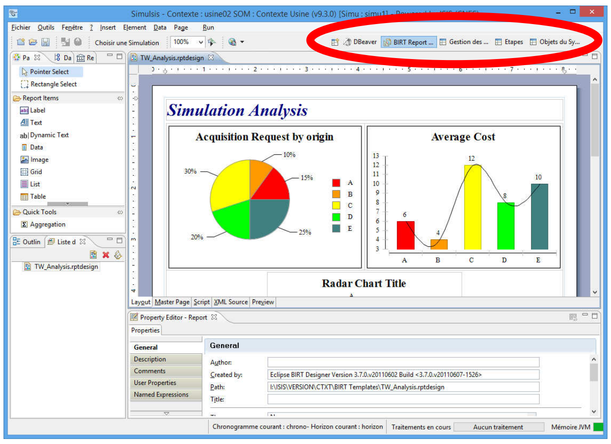
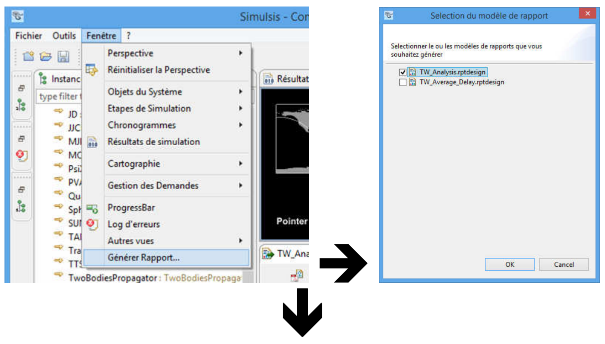


Fig. 14 – BIRT Report configuration high quality Perspective

Even though BIRT Perspective can be used "as is" in its Perspective (as shown on Fig. 14), for a better user experience, it was decided to enhance integration with some improvements:

- Ability to display Reports in other Perspectives using specific menu and selection pop-up window.



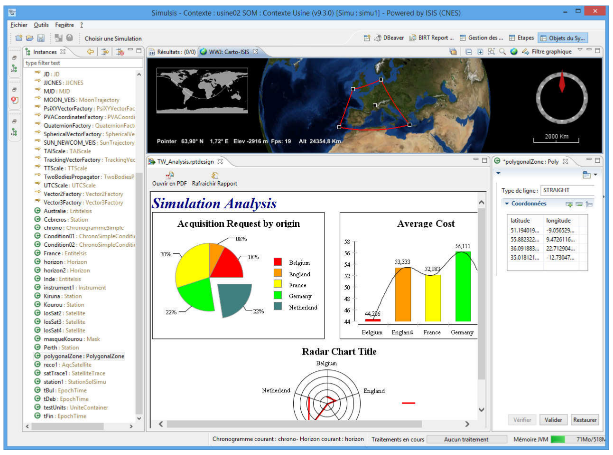


Fig. 15 – ALIS shows Report in simulator's Perspective

- Ability to update reports' views and/or generate pdf exports during stage runs (useful for long term simulations):

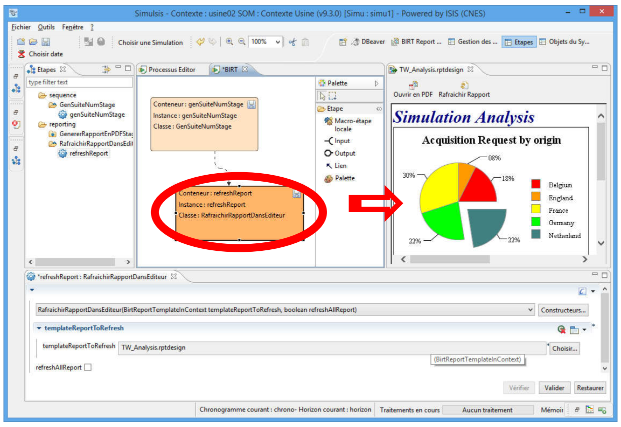

Simulation Analysis
Acquisition Request by origin
Conteneur : refreshReport
Instance : refreshReport
Classe : RafraichirRapportDansEditeur

Fig. 16 – Update reports views and/or generate pdf export

6. CONCLUSIONS

We have seen in this paper that a mission simulator is a program that simulates operational software which computes satellites working plans. Due to very high algorithms and parameters combinatory, a mission simulator is essential to System provisioning. CNES has developed a simulation infrastructure named ALIS, which provides services to build mission simulators.

Complexity of System studies requires high computation performances. ALIS architecture has been adapted to make it distributed and usable on an adapted hardware platform, a Linux cluster for instance.

Moreover, as new user needs frequently arise, it has become necessary to rely as often as possible on high-level COTS, which induce reduced integration costs.

Thanks to ALIS, whose open architecture allows use in other domains (SSA, etc.), CNES can bring its expertise efficiently, and proceed to deep and extended system-level studies, while CS SI brings its skills and experience to ALIS development.