# Testing spacecraft autonomy with AGATA

Marie-Claire Charmeau[1], Jérémie Pouly[1], Eric Bensana[2], Michel Lemaître[2]
[1]*CNES, Toulouse, France,* [2]*ONERA, Toulouse, France*
*Marie-Claire.Charmeau@cnes.fr*

## Abstract

*AGATA is a research and demonstration program led by ONERA and CNES on the topic of autonomy for space systems. Current work consists in the development of a testbed, including a spacecraft simulation and ground segment tools for operations. The on-board software of the spacecraft is based on a new architecture dedicated to autonomous systems and a new process of software development and validation is being tested with the objective of mastering more easily the complexity added by autonomous functions in a spacecraft architecture. The first step of this process is the description of a decisional architecture which has to be translated into a real-time software architecture before being implemented and tested on the testbed. The process is largely model-based and partly automated by a Java code generator. In this paper, we describe the simulation framework and the process used for the on-board software development and validation.*

## 1. Introduction

There is a gap between in-flight experiment of spacecraft autonomy and its operational use on a space project. Even flight-proven algorithms may not easily convince project managers that they can be used safely and without additional cost. This is why CNES and ONERA started a few years ago a research program dedicated to spacecraft autonomy the purpose of which was to make on-board autonomy a ready-to-fly technology. Parts of the activity consist in defining algorithms for on-board planning or advanced FDIR and implement them in a specific architecture. This architecture is based on decisional components with both reactive and deliberative behavior. How to translate this decisional architecture into a flight-software architecture, while respecting industrial constraints and safety and validation requirements is the challenge that we address through the AGATA demonstration program.

## 2. The AGATA program

### 2.1. Partners and context

In July 2004, CNES and ONERA signed an agreement to start a common research program on autonomy for space systems. Since then, a team of ONERA researchers and CNES engineers have been working together with the same objective: federate research activities on autonomy through a demonstration project. This program is composed of research studies on topics such as architecture, autonomous planning and diagnostic, software development and validation methods. The results of these studies are applied to the AGATA (Autonomy Generic Architecture - Tests and Application) lab bench which is being built for this purpose.

The LAAS-CNRS plays a part in the research activity by providing its experience in decisional architecture for robotics and diagnosis algorithms. Industrial partners are also involved in this program. Their point of view is particularly useful and helps to produce a final software architecture compliant with industrial requirements.

### 2.2 Objectives

The main objectives of this program are to:

- study feasibility of high autonomy levels for space systems (except launchers)
- demonstrate the interest of autonomy and its impact on ground operations and operator skills
- define and test a process for the development and the validation of software dedicated to autonomy
- develop a rapid prototyping tool to evaluate autonomy concepts for future projects.

The demonstration testbed provides the framework to reach these objectives.

### 2.3 The ground demonstration testbed

The aim of the demonstration is to study the whole system consisting of:

- a space segment with one or several spaceships or one spaceship and several ground devices. In case of rovers, the aim is not to study their own autonomy, but just to model them as part of a larger system;
- a ground segment which can be split into ground stations, a control center and mission centers.

The satellite's behavior and its environment are simulated. The focus is put on the on board software, which can be either simulated as a functional software or integrated in the simulator as a real-time flight software. Hardware definition level in the simulation is precise enough to test FDIR algorithms and redundancy will be taken into account too in order to test reconfiguration scenarios.

For the ground segment, which has not been developed yet, we intend to use as much as possible recent generic tools developed by CNES for its control centers. In order to get a feedback on the impact of autonomy on ground operations, real operators will be asked to control a simulated autonomous spacecraft.

Several mission scenarios have been selected as candidates for a demonstration of the advantages of autonomy [1]. The first mission, currently the baseline for the studies, is an Earth monitoring mission.

## 3. HOTSPOT, a virtual mission

### 3.1. Fire and volcanoes monitoring

This mission is inspired by the ESA Earth Watch program FUEGOSAT. It is an imaginary mission which has been defined to combine the need of a quick response to an event with the need of autonomous planning of on-board activities. The mission objectives are to detect, observe and monitor events such as fires or volcanic eruptions. Once detected, events must be localized and if possible identified very quickly by the satellite before sending alarms to the ground and planning an observation phase of the phenomena. Of course data must be downlinked as soon as possible to provide support to the ground intervention teams. On-board reactive planning can help to associate a long-term background mission which consists in monitoring predefined areas and a detection and alarm mission with short-term requirements. The planning system has to manage energy and memory resources. The mission is performed by means of a constellation of low Earth orbit satellites, but only one satellite is taken into account at present time in the demonstrator.

Priority levels are associated to the monitoring requests. Those subsequent to a new detection have the highest level, and those related to the background mission have the lowest level.

### 3.2. The mission architecture

Two mission centers can send commands to the satellite. The main mission center is dedicated to fire monitoring, and the second one monitors the volcanoes. Both can receive the data emitted by the satellite as they share several ground stations. They can also receive the alarms triggered after the detection of a new hotspot and relayed by a geosynchronous communication spacecraft.

In the current scenario, they do not have a direct access to the satellite for the commanding link. They have to send their command plans through the control center. An alternate scenario would enable this direct access.

The commands send by the mission centers to the satellite are high-level requests. They are related to the background mission only, as the observation requests linked to the hotspots detected on-board are generated by the on-board software. A request contains the coordinates of the area to be monitored, and the frequency of the monitoring task (once a day or once a week for example).

The main mission center also has the possibility to tune the autonomy algorithms with specific commands, and to adjust the visibility level on the satellite's behavior by modifying the telemetry plan.

### 3.3. A generic platform

The satellite platform is a typical platform for LEO missions, without attitude maneuvering ability. A set of sensors, actuators, and communication and power subsystem hardware has been defined for the main platform functions interacting with the mission activities (AOCS, communication with the control centers, power management). Interface with the on-board software has been defined at a functional level. In this simulation tool, low-level communication protocols are not simulated, as opposed to simulation tools that are used for operational tests of real missions.

The occurrence of anomalies or failures can be triggered via the simulation control interface. Communication with ground segment tools is done by the exchange of telemetry and telecommand files, and potentially other files that could be useful for the observation of autonomy functions, such as event files, reports and so on.

## 4. Validation issues

### 4.1. Autonomy-related difficulties

One of the challenges addressed in the AGATA program is to define a development and validation process adapted to highly autonomous systems and compliant with space industry standards. Usual validation methods are inappropriate for such systems because of the onboard decision capability associated with autonomy. Since the system reacts to events in a complex way depending on an unknown context, exhaustive testing is impossible.

Mastering this complexity can only be achieved by applying a step-by-step development and validation process, from high-level autonomy algorithms testing to fully-integrated software validation. Key steps include the following:

- Define a generic architecture adapted to advanced autonomy needs and ensuring a low coupling between application processes so that testing may be done separately for each of them.
- Specify and test autonomy functions at decisional level in order to validate the algorithms before embedding them in the software.
- Prototype the flight software to check, without detailing all parts of the software, that high-level mechanisms trigger expected behaviors.
- Run functional validation tests on the functional simulator containing representative models of platform and payload hardware devices, and including communication with a control centre prototype.
- Run real-time validation tests on the real-time simulator running the on-board software on a calculator emulator and using the same models as the functional simulator.

This process contains two software validation steps: Functional validation is separated from strictly real-time validation. Functional simulations are used to validate the decisional behavior of the spacecraft and its interaction with the ground control, whereas real-time simulations are centered on software behavior with respect to hard real-time constraints.

### 4.2. An adapted validation process

As detailed in the software development section, the leading idea in the AGATA program is to apply an iterative and incremental development process. This guideline will allow to perform part of the validation process directly at model level and use auto-code generation to transfer the validated properties to the software code. Properties that cannot be verified at model level will be validated progressively on the auto-code through successive increments and iterations. This approach allows to focus on specific subsets of the onboard software code that need to be validated over different iterations.

As demonstrated, validation is largely based on the model-based approach and on the incremental and iterative development process. The two guidelines are to validate as much as possible of the software as early in the development process as possible (directly at model level when applicable), and to use auto-code generation in order to reduce the validation effort related to implementation.

This validation process is being tested on the virtual HOTSPOT mission. A generic decisional architecture adapted to autonomy needs was designed and used to validate high-level algorithms. Then, after a decisional software was prototyped with the Esterel language, the functional software, auto-coded in Java language from an UML specification, was tested in a first version of the simulator. This functional simulator will soon be upgraded to enable real-time simulation in order to perform the last validation step to confirm onboard autonomy as a ready-to-fly technology.

## 5. Decisional architecture

### 5.1. A generic architecture

AGATA generic architecture aims at specifying the decisional mechanisms of the on-board software. As the definition of a whole autonomous system may be very complex, we chose to describe it as a modular structure. Modules are built on the basis of a common pattern and connected together to form a global architecture. The objective is to describe the expected behavior of the system in such a way that it is easy to understand and to validate.

Each module is in charge of controlling a part of the system and of handling the data associated to this part. It takes into account requests and information coming from other modules and can send requests or ask information to other modules. To avoid potential decision conflicts it cannot have direct access to the part of the system controlled by another module.

Each module is built on a sense/decide/act pattern. The module maintains its own knowledge of the state of the system part it controls, on the basis of an internal model and data acquired from other modules or from hardware elements (such as sensor measurements).

The module uses this knowledge and the requests it has received to decide on which action to perform (actions include sending a request or a command, changing its behavior, reporting an event, or even doing nothing).

Even if the modules are all built following the same pattern, we can distinguish different types of modules. Low-level modules which control hardware parts are called monitors. The decisions they can make are limited to local control loops and FDIR functions. They process the data that are used by higher-level modules. The highest-level modules control the global behavior of the satellite. One possible decisional architecture based on these modules is a hierarchical architecture with a "mission" module at the top level [2].

### 5.2. Reactive and deliberative tasks

Each module can make a decision and produce new requests addressed to itself or other modules. The decision-making process combines two tasks: a reactive control task and a deliberative reasoning task [3]. The reactive control task analyses the current state of the module and new requests or new events that have been received. It can either react immediately, using pre-defined decision rules or algorithms with short computation time, or decide to trigger a deliberative task. That task runs an algorithm which needs some time to produce an optimized result, such as planning or diagnosis algorithms. It can provide intermediate results and is interrupted by the reactive control task when the final result is due. The longer the computation time allowed, the better the final answer.

The solution delivered by the deliberative task is only an advice, and the reactive control task decides whether to follow it or not. The control task should always be able to make a decision, even without any answer from the deliberative task. This decision may not be an optimized one, but the decision process must not be blocked by the deliberative task.

## 6. Software development

### 6.1. Software architecture

Once the decisional architecture has been defined, the next step is to translate it to a software architecture taking into account the constraints associated to embedded critical real-time software.

The fulfillment of some of the constraints might require to alter the decisional architecture, but the idea here is to comply with the constraints, remaining as close as possible to the theoretical architecture designed previously. For example, good practice development rules in real-time software state that the higher the number of interacting tasks, the harder to prove any determinism of the final software. Therefore, the flight software cannot implement the decisional architecture as given, with potentially 3 to 8 tasks per module and more than 20 modules. Thus the software tasks might be defined in order to fit the decisional architecture with a lower granularity (all tasks defined in the decisional architecture were used as undividable entities, and sets of those tasks were combined as software tasks), and to minimize interactions between each tasks (both in terms of direct communication and access to shared data).

### 6.2. A two-step development process

For AGATA, a two-step development process is followed:

- an early development cycle based on a synchronous language – Esterel – that focuses on the integration of autonomy-related algorithms (such as on-board planning), and aims at prototyping and validating the decisional part of the software, and setting up the communication interface with the simulator;
- a model-based approach that consists in writing a complete formal specification of the software in UML 2.0 and uses an auto-code generator to produce the Java code for this asynchronous software.

Both implementations are coded in Java language using the Esterel to Java compiler for the “Esterel-based” approach and a home-made UML to Java auto-code generator for the “UML-based” approach.

This dual development process produces two different implementations of the same functional flight software, acknowledging the same interfaces, and ready to run in the simulator. Although they are independent, those two versions are also connected since the synchronous “Esterel-based” implementation is used as a testbench to validate new mechanisms and algorithms before they are applied to the UML-based implementation. Common interfaces are defined for both versions so that, beyond compatibility with the same simulator, they can also use common libraries (mathematical calculations, astrodynamics classical functions,…). Furthermore, identical data structures (“classes” in the formalism of UML or Java) are used in both implementations so that they only have to be validated once.

Aside from its own purpose, described later on, the Esterel-based implementation is largely used as an intermediary step in the development of the UML-based one. The leading idea here is to benefit in the UML-based implementation from the preliminary synchronous modeling permitted by synchronous languages like Esterel. These advantages are mainly:

- a sound mathematical semantics for concurrency, enforced by the compiler;
- the support of the two most common synchronous execution patterns: event driven and clock-driven;
- the simplicity of the language and its associated formal model, making formal and practical reasoning tractable, and giving to the Esterel code the flavor of an executable specification. That framework allows to validate before-hand part of the code used in the UML-based implementation, such as data handling classes, as well as prototyping new autonomy-related algorithms.

## 6.3. Esterel-based approach

The Esterel-based implementation greatly simplifies the solving of scheduling problems, as the compiler is able to compute the ordering of the actions to be done during a reaction step. This allows to focus on the behavioral part of the software: given a set of inputs, what should be the outputs in order to fulfill the mission objectives.

This synchronous approach, compared to asynchronous ones, makes it easier to implement and validate the state machines and other behaviors defined in the decisional architecture: They can quickly be tested in the simulation environment, and potential weaknesses in those mechanisms – and their associated improvements – can be highlighted. Another benefit to the use of Esterel, is to separate the behavioral part of the software (coded in Esterel) from the algorithmic part (coded directly in Java). The Java algorithmic part of the software, shared with that of the UML-based approach, is hereby validated for both implementations in this decoupled framework – which decreases the validation effort.

## 6.4. UML-based approach

The UML-based implementation primarily consists in writing a detailed description of the software in UML 2.0. The UML model contains both architectural and detailed specifications (from high-level requirements to the definition of the method bodies described in UML pseudo-code). The two objectives here are to contain the whole specification of the software in a single model instead of several text documents and to allow the use of automated generators to produce either documentation or the software auto-code.

This model-based approach allows an incremental and iterative development process instead of the classical V-shaped development cycle. The software development is divided into several increments (mapped onto software functional objectives), and each of those is then subdivided into a few iterations that contain different subsets of the specification for the current increment. This approach grants a progressive validation of the software, enabling an early and less costly detection of potential mistakes in the specification. However, such a development process relies extensively on auto-code generation to quickly prototype the software after each iteration, which corresponds to evolution of the UML model.

## 6.5. The AutoJava generator

The auto-code generator used in AGATA is a RT-Java generator prototype developed specifically for the project. Although this paper only addresses the functional flight software – i.e. software simplified from most real-time concerns – the long-term objective of the project is to produce a real-time software. Therefore the UML model contains the description of the functional software and the associated real-time specification.

Until recently the UML standard did not contain any framework to describe real-time properties of the objects in the model, therefore, for AGATA, a profile named “AutoJava” was developed for this purpose (currently, real-time properties are only used as hints to help specify the behavior of the functional flight software). The AutoJava profile is associated with a RT-Java auto-code generator (also referred to as “AutoJava”) that can be applied to the UML model completed with the profile for real-time aspects. The RT-Java code generated is then automatically converted to standard Java by a bash script (a few manual modifications remain) and tested into the simulator. This auto-code generation process in two steps was chosen in order to remain compatible with both real-time (in RT-Java) and functional (in standard Java) implementations of the software.

The combining of the AutoJava generator (to generate RT-Java code from the UML model) and the bash script (to convert the RT-Java code into standard Java) produces an automated tool to generate Java code from a UML specification.

This development process is currently being applied in AGATA. Since the theoretical generation tool presented early on is still under development, the generation of Java auto-code from the UML model still requires manual interventions from the developer. Nevertheless, the process is converging and should be operational soon.

## 7. AGATA demonstrator architecture

### 7.1. The software/simulator interface

The AGATA demonstrator is composed of an onboard software and a satellite simulator that includes environment simulation (i.e. inputs to all sensors in the satellite). As stated before, the simulator provides the user with means to control the simulation through environment events such as hardware anomalies or failures and hotspot occurrences. Its interface is defined for the purpose of software validation and it offers all functionalities necessary to verify any functional behavior that needs to be checked. The user interface is a Tcl-Tk layer that sits on top of the C code of the simulator, and it can be easily upgraded to fulfill new simulation control requirements.

Simulator and onboard software communicate through a functional interface that masks low-level communication protocols. This interface defines a set of functions that are available for each side to interact with the other side: The simulator can activate the onboard software at the required frequency, and the software can read equipment inputs from the simulator and send the subsequent outputs. However, whereas the simulator is mainly coded in C language, the onboard software is coded in Java language. Communication between the simulator in C and the onboard software in Java was enabled by the Java Native Interface (JNI) and the Simplified Wrapper and Interface Generator (SWIG). JNI was used to activate the Java code of the onboard software from the Simulator (including the loading of the Java Virtual Machine), and SWIG was used to call C functions on the simulator side from the software.

### 7.2. A simulator-driven demonstrator

The functional interface setup between the onboard software and the simulator is largely based on the simulator internal architecture. The simulator is based on a Kernel that activates various simulation models sequentially at their specific frequencies and manages communication between them. Typical models include:

- standard platform hardware: GPS receivers, Star Trackers, Thrusters,…
- specific mission hardware: a “Hotspotter” that aims at detecting hotspots on the Earth surface, an “Imager” that can take images of areas nearby hotspots, a high-definition transmitter to download pictures, a mass-memory dedicated to mission needs and an alarm transmitter via geosynchronous relay to warn the ground segment upon discovery of new hotspots
- environment: current orbit, position, speed and orientation of the satellite, current date,…

The simulation Kernel activates the simulation models in a static predefined order based on user’s preferences and models frequencies. Most importantly, the onboard software is also viewed as a “simulation model” by the simulator, and is activated intermittently as a whole, at the highest frequency of the software functional tasks. Upon activation, it executes all the actions it needs to run during this cycle whatever the true duration of these actions. When all actions in the cycle have been executed, the onboard software stops and releases CPU resources in favor of the simulator Kernel, allowing it to activate the next simulation model.

### 7.3. Computing power

In the current simulator/software activation principle, the demonstrator may only prove functional properties of the onboard software. These are substantial-enough objectives to start with, and are currently being fulfilled by the AGATA team.

However, final project objectives lie far beyond these and a real-time version of the simulator is being developed at CNES. In fact one of the main foreseen bottlenecks for implementing autonomy concepts is related to the on-board computing system power. The relation between on-board processing power and achievable levels of autonomy will be addressed during the project first by measuring memory or power needs when running the functional version of the software and then with a new simulator based on a calculator emulator in “real-time” interaction with the simulation models via a simplified I/O server. The onboard software will be executed continuously on this emulator and will be submitted to real-time constrains, such as limited CPU resources.

Future developments should include a hybrid version of the AGATA lab bench where real on-board CPU will be used to run the proposed software.

## 8. Conclusion

A first version of the on-board software dedicated mainly to hotspot detection and localization has been integrated in the simulator. The whole process was used for the first time to perform this development. It highlighted some hurdles mostly related to the UML tool and the integration of a functional Java code in the simulator. This first increment in the cyclic development helped to adjust some development rules that will be used for the next increments.

The next version will include the data-handling system and the telecommand and telemetry management software which will enable connection with the ground segment tools.

## 9. Acknowledgment

Part of the on-board software development work was performed by Julien Conte (CSSI).

## 10. References

[1] M.C. Charmeau, E. Bensana, "AGATA, a lab bench project for spacecraft autonomy", *iSAIRAS*, 2005

[2] G. Verfaillie, M.C. Charmeau, "A generic modular architecture for the control of an autonomous spacecraft", *5th International Workshop on Planning and Scheduling for Space (IWPSS)*, 2006

[3] M. Lemaître, G. Verfaillie, "Interaction between reactive and deliberative tasks for on-line decision-making", *ICAPS 2007 Workshop on Planning Plan Execution for Real-World Systems*, 2007

Filename: m102-Charmeau.doc
Directory: D:\F
Drive_new\conferences\iSAIRAS\MemoryStickLoad\Manuscripts\SESSION 22
Template: C:\Documents and Settings\klittle\Application
Data\Microsoft\Templates\Normal.dot
Title: Author Guidelines for 8
Subject:
Author: Larry Bergman
Keywords:
Comments: iSAIRAS 2007 Paper Template
Creation Date: 12/7/2007 5:44:00 PM
Change Number: 5
Last Saved On: 12/7/2007 6:14:00 PM
Last Saved By: CNES
Total Editing Time: 40 Minutes
Last Printed On: 3/4/2008 1:11:00 PM
As of Last Complete Printing
Number of Pages: 7
Number of Words: 4,242 (approx.)
Number of Characters: 24,182 (approx.)