ABSTRACT

This paper presents the work of Astrium/IA (the Avionics Products Directorate of Astrium) with CNES for the definition of autonomous spacecraft architecture.

The current spacecraft system evolution trend consists in moving to board operations usually carried out on ground in order to automate maintenance and mission operations and provide increasing performance and reactivity to the whole system. However, reaching such goals requires a careful definition of hardware and software architectures with the ability to handle the increasing system complexity. This leads to define the notion of autonomous spacecraft systems.

Autonomy is the property of the system that ensures that the spacecraft is able to realize its objectives even in the absence of human intervention. Autonomy can be enforced by giving to the system the ability to take decisions by itself to fulfil its objectives, and by ensuring that it has a sufficient degree of dependability to carry out these decisions continuously. However, compared with current existing systems, an autonomous system introduces a high degree of complexity at software level. It is thus necessary to define a system specific architecture that can address this complexity problem.

The paper demonstrates how an autonomous system can be designed dealing with the complexity of the different actions it must perform.

1. INTRODUCTION

The goals of autonomy can be classified in two main classes:

- It provides a way to carry out new types of missions (such as interplanetary flights where it is difficult to communicate with the spacecraft, or satellite constellation control by minimising command control complexity);
- It provides a way to carry out in a different way current missions, by minimizing the costs of the ground segment and increasing the reactivity of the system.

The reference mission chosen in the framework of the project is the FUEGO constellation. It is composed of twelve satellites whose role is to detect fire occurrences in risky zones. Bringing autonomy to this constellation provides several challenges that can be easily supported by our autonomous architecture:

- The on-board planning and scheduling of mission and platform management activities would increase the reactivity of the system and decrease the amount of work being performed on ground;
- The autonomous coordination between the satellites would also reduce the on-ground activity to synchronise the overall mission objectives;
- The spacecraft ability to detect failures on-board and reconfigure automatically would enhance the overall mission autonomy;
- The on-board configurable functionalities (TM flows, configurable macro-commands or mission plans) help the ground to tune the system.

To achieve autonomy requirements we have defined a system architecture that is based on robotic and artificial intelligence related architectures, and includes the required level of dependability and safety. This approach leads to define:

- Appropriate hardware architecture to ensure sufficient fault tolerance and reconfiguration properties;
• Appropriate software architecture able to handle on-board decisional capabilities, taking into account both the mission goals and the safeguard of the mission (i.e., safety goals must be appropriately included at the decisional level, so that the actions performed by the spacecraft cannot lead to hazardous state);

• A software architecture that deals with the software complexity on-board, by taking into account potential fault propagation between the software components of heterogeneous criticality levels.

2. REVIEW OF DEPENDABLE SPACECRAFT AND ROBOTIC ARCHITECTURES
We briefly describe in this section the various architectures that compose our baseline. It consists in:

• Dependable spacecraft hardware architectures (Rosetta, GUARDS [3]) that contribute to the definition we retain in the context of the project (§ 2.1, § 2.2),


2.1 Rosetta
This hardware architecture includes two pairs of bi-processors (two CDMU). The redundancy scheme used is a cold redundancy, i.e., the second CDMU is switched on in case of failure detection of the primary one. The second CDMU can thus be used as a recovery unit.

However, during the critical phase of the mission, the four CPUs can be used in a hot redundancy scheme, to ensure the continuity of service.

The segregation of the AOCS and DMS software on the two processors aims to provide higher performance. However, from a dependability point of view, such an approach ensures the isolation between the two software entities, providing thus error confinement properties. Consequently, one of the two software entities can fail without disturbing the other one, providing that they are sufficiently functionally independent.

Even if this was not a goal of the Rosetta architecture, these dependability aspects must be carefully taken into account in the context of autonomous spacecraft architectures, where software complexity is a one of the main points to consider.

2.2 GUARDS
To fulfill various requirements and in particular to be able to tolerate both physical and design faults, the GUARDS architecture is structured around three main axes.

The channel dimension C provides primary fault containment regions, and is designed to take into account any kind of faults (including the so-called byzantine faults). According to the number of channels, several kinds of GUARDS instances may be envisaged. Two-channel instances can be used to implement the classic duplication-and-comparison scheme, in order to ensure safety. Three-channel instances can be used to implement the well-known Triple Modular Redundancy scheme, in order to improve reliability or availability without degrading safety. Four-channel instances can provide additional flexibility, by widening the spectrum of applicable interactive consistency algorithms and by allowing one channel to be isolated (e.g. for maintenance purposes).

The multiplicity dimension M defines secondary fault containment regions, by decomposing each channel into several independent processor boards, which can then be used to improve either availability or fault diagnosis capabilities of the channel. The multiplicity dimension can also be used to improve performance (through parallel processing), or to physically segregate software of different criticality.

The integrity dimension I prevents the propagation of errors from low-level integrity software to high-level integrity software. This dimension allows highly critical software and standard software to share the same processor, without forcing the standard software to be validated as if it were highly critical (solution commonly used, but not very cost effective).

The Guards architecture is able to provide any level of dependability required, depending on the type of its instance. It is thus adapted to any mission requirements. Moreover, it addresses the problem of the software faults that is an important notion in the context of high complexity software systems (e.g., autonomous systems).
2.3 **Deep Space One**

The Deep Space One architecture is based on the integration of artificial intelligence related decisional software with well-known real-time software. The major objective of this system was to produce sufficiently reactive decision software to be used on an embedded spacecraft. The architecture developed is composed of the following components:

- A mission manager and a planner scheduler whose goal is to create the activity plan by resolving the constraints (temporal and value prediction) of the predicted system state;
- The plan produced is composed of a sequence of commands executed by the executor that also verifies if they are successful;
- In case of failed command execution, the Mode Identification and Reconfiguration module is used to perform the on-board failure diagnosis and determine what reconfiguration has to be launched to be able to continue the mission execution;
- A monitoring is implemented to provide discrete values of the real-time system state to the MIR module.

The executive module creates both the mission manager/real-time system interface and the ground/real-time system interface. Moreover, the decision software (composed of the mission manager, the planner/scheduler and the MIR module) can be considered as a on-board human like decision capability. It is built such that it can be deactivated if required, producing then a degraded software similar to the current ground-driven spacecraft software system.

2.4 **LAAS Architecture for Autonomous System**

The LAAS architecture defines three software levels:

- **Decision level** that should be considered as the autonomy level and is requiring some decisional aspects (planning, scheduling). This provides high-level software layer for task involving human like decision.
- **Execution control level** that is certainly one of the most critical components of the architecture. It is a purely reactive system, with no deliberation capability, which acts as a time bounded filter between the decisional level and the functional level. It is thus responsible for catching poor or inadequate decision that could come from the decisional level or the user.
- **The functional level** integrates all the operational functions (hardware control, servo-control, data processing,...). It integrates all real-time activities of the system.

**Decision level**

The decision layer is mainly composed of:

- a supervisor whose role is to manage the different actions that are part of the plans it handles;
- a planner that creates these plans on the base of the existing plans and of the knowledge of the system.

Compared to the current existing systems, this decision level can be considered as a new component in the system. It encapsulates all human like on-board decision capabilities.

This part of the software architecture constitutes the main autonomy software component. However, its insertion can impact the overall spacecraft architecture due to its intrinsic nature: it is event driven, it must coexist inside the system with cyclic tasks, and must have the knowledge of the real-time activities of the system.

**Execution control level**

The execution control level can be described by the following attributes:

- It provides the interface between the embedded system and the operator, who send requests to it;
- It receives a flow of requests sent by the decision level;
- It verifies the compatibility between these requests and the internal state of the functional level. Its role will then be to activate subsystem procedures at lowest level.

This software level builds the interface between the ground and the embedded system, and provides error confinement properties to ensure the functional level is correctly activated.

**Functional level**

The functional level contains all the real-time functionalities of the system.

This three-layer architecture appears to be particularly adapted to spacecraft architectures where the two lower layers are already available in the system. The hierarchy emphasizes the interfaces between the software components of the
system and shows their particular dynamic properties. No doubt it provides a way to master the software complexity inside an autonomous spacecraft system.

3. **PRESENTATION OF THE ARCHITECTURE**

The architecture proposed is composed of both hardware and software component definitions. It aims to address the safety and dependability problems and proposes a hierarchical decomposition of the software in layers that determine the interactions between the several components of the system.

### 3.1 Dependability and safety requirements

To ensure that the system is autonomous, we must give it the ability to react to situations where its internal state becomes erroneous. It must thus be able either to tolerate or to recover from a hardware failure and continue to perform its function, at least in a degraded mode. This requires the use of fault tolerance mechanisms that must be applied to both hardware and software levels.

Additionally, the on-board decision level constitutes a high complexity piece of software against which the real-time functional level has to be protected. This implies the definition of error confinement areas to prevent error propagation from this software of heterogeneous critical levels.

### 3.2 Hardware definition

The hardware layer relies on the use of four computer units organized in two pairs of hot redundant units that provide:

- Hardware fault detection by using a duplex and comparison scheme. This provides a good fault detection coverage, ensuring that any failure is detected and that the service will be of integrity;
- Software fault containment areas by distributing the software components on the nodes depending on their criticality. This leads to put the decision software apart from the execution control and real-time layers, following the GUARDS integrity levels management method.

### 3.3 Software definition

The software architecture definition relies on the robotic LAAS architecture [1] and is composed of three different layers:

- A functional real time level, which is responsible for the control loop of the system. The level also realizes the interface between the control system and the hardware functions handled by the system. It eventually encapsulates the Attitude and Orbit Control System, the Payload system, the Thermal and Power systems. Additionally, all basic functionalities like mass memory unit management are part of this software layer, as well as most of the DHS services;
- An execution control level, which is at the centre of the control system. It is in contact with the real time level (that it controls), with the ground (that send requests to the system), and the decisional level that autonomously requests services from the real-time layer through the execution control level. The execution control level is also in charge of preserving the integrity of the system by verifying the correctness of the commands with respect to the real-time system state;
- A decisional level that is composed of planning and scheduling activities linked with platform maintenance and mission related activities.

However, our software architecture takes advantage from the lesson learned from the Deep Space One architecture. In particular, it is inspired from the FDIR and ground interface of this project to define adequate interactions inside the system.

### 3.4 Modelling of the defined software architecture using UML

This functional description leads to the modelling of the software architecture using the UML language.

*Decision Level*

The decision level (Fig. 1) can be either active or inactive, depending on the autonomous state of the spacecraft. If this layer is active, the supervisor processes all ground requests. It can either insert them into the plan of actions, or request the planner/scheduler to generate a new plan taking into account the new objectives. In order to create its plan, the
planner uses a knowledge base composed of unresolved plans, the current known state of the system and states predicted by the planning experts (e.g., the attitude and orbit predictions).

The decision level is in interaction with the other software layers via the supervisor, the FDIR diagnostic system and the state model translator. This last component aims to update the system state known by the decision level in accordance with the real state.

![Decision Level Diagram](image)

**Fig. 1. Decision Level**

This level must be considered as the really new software part of the system, compared to current existing ones. It is the most CPU power consuming part of the system, and the most complex (statically and dynamically). It is although the less critical part of the system, since its failure does not imply a total loss of the mission (this is not the case of the functional level). It is thus natural to ensure the isolation of this software from the other parts. This is provided by the hardware architecture. The execution control level must then filter the errors that can occur in the decision part.

**Execution Control Level**

The execution control level (Fig. 2) builds the interface between the decision and functional levels. The execution controller executes all actions sent by the supervisor. It is composed of a command router and a set of verifiers. Each command is verified before being routed to the appropriate service. This verification can consist either in a set of execution preconditions that would verify the action is in accordance with system state prerequisites, or a simulation of the command to verify the action will not lead to an unsafe system state. Once verified, the command is sent to the real-time subsystems at the functional level. The status of executed command is returned to the supervisor that can, on failure report, request information from the diagnostic FDIR module at decision level.

The verification stage provides an error confinement mechanism that would prevent the functional level to be corrupted by erroneous actions performed by the ground or the decision software.

The interpreter provides high-level subsystem command ability. It both provides macro-command availability to the supervisor and a configurable and flexible interface to the ground. In case of ground manual intervention (e.g., in a non-autonomous mode), the interpreter can be used to build a procedure toolbox, whose functions are executed on board.

**Functional Level**

The function level (Fig. 3) is composed of all the functional subsystems, i.e., thermal, power management, payload and AOCS. It also includes all software equipment components, and their monitoring. A fast reactive FDIR is linked to this
monitoring service in order to update the states of the subsystems in case of error detection. This would prevent further erroneous commands to be executed. The monitoring activity additionally updates the system state as seen by the decision level.

4. CONCLUSION

The step towards autonomy seems to be certain. However, mastering the increasing complexity of the embedded systems requires architecture definitions. The one we previously defined provides various advantages. It covers all the autonomy requirements, and ensures an easy to validate application design. Due to its simplicity and genericity, it can be easily adapted to each type of mission.

In the context of the context of FUEGO, all mission specific components have been easily inserted inside the architecture definition, demonstrating the adaptability and the adequacy of the solution proposed.

Moreover, our architecture is generic enough to be suitable to current architectures. This provides a way to introduce autonomy progressively in the system.

5. BIBLIOGRAPHIE

