Model-based specification of the flight software of an autonomous satellite

Jeremie Pouly, Sylvain Jouanneau

Model-based specification of the flight software of an autonomous satellite

Jeremie Pouly (CNES) and Sylvain Jouanneau (ALTEN SO)

Contact Author: Jeremie Pouly
Affiliation: CNES (France)
Address: 18 avenue Edouard Belin, 31401 Toulouse Cedex 9, France
Email: Jeremie.Pouly@cnes.fr
Phone: +33 5 61 28 16 67
Keywords: UML, code generation, model simulation, RTSJ, Topcased, autonomy, satellite

Abstract

In the framework of the AGATA program, we applied a model-based development process founded on a living UML specification to produce the RT-Java code of the AGATA onboard software. Derived from classical V-shaped production cycle, our development process benefited from several model-based engineering methods, such as model-debugging and automated code generation. Our resulting Y-shaped production cycle enabled an incremental development process that allowed us to start software validation early in the process. Despite the complexity of the AGATA onboard software we thereby manage to achieve its functional validation and were able to evaluate RT-Java (Real-Time Java Specification – RTSJ) for real-time space applications.

1. Introduction.

1.1. The AGATA program

The AGATA program, led by CNES and ONERA, aims at developing a ground demonstrator for autonomous satellites including a space segment and a ground segment. This generic demonstrator can be instantiated for different missions with little effort and it will be used to test different on-board algorithms for mission planning, smart and reduced ground communication, or advanced fault detection. It aims at demonstrating that, while functionally fulfilling the mission with great success, even for worst cases the autonomous on-board software is guaranteed to comply with hard real-time constraints. However autonomy means complexity and the more autonomous functions implemented on-board, the harder to design and validate the flight software. In order to address this complexity for the AGATA on-board software we chose to combine different technological breakthrough: new modular architecture, model-based development, model simulation,…

1.2. Autonomy brings complexity

In fact 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. It is also very difficult and time-consuming to design meaningful testing plans of the whole system during the early phase of the process. Moreover the system complexity makes it really hazardous to design the whole system from scratch at once.
Classical development processes such as the waterfall or V-shaped development cycle, which work well for where requirements are easily understood, allow not enough flexibility to design complex autonomous systems. In these processes, adjusting the scope of the software is difficult and expensive, and since the software is only developed during the implementation phase, no early prototypes software are produced and potential problems or feasibility dead-ends cannot be foreseen. Finally the V-shaped development cycle doesn't provide a clear path for problems found during the testing phases of a highly autonomous system: the system complexity requires to deal with each part of the system separately as much as possible.
The spiral development process which emphasizes risks analysis is not appropriate either since it is very hard and therefore very expensive to implement a risk analysis on such a complex system: since exhaustive testing is impossible, it will not be possible to fix all issues and to cover every requirements with appropriate classical testing.

1.3. Towards model-based engineering

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. Our process was therefore derived from the V-shaped development cycle with improvements and automations in the bottom part of the process. The resulting Y-shaped production process enabled an iterative and incremental development cycle with a model-based approach based on UML 2. 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.
- Follow an iterative an incremental model-based development process.
  - Incrementally specify the whole system in a high-level language, such as UML :
  - Validate the UML specification as early as possible in the process using model simulation.
  - Use automated code and tests generation to incrementally implement and validate the flight 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 a 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.

2. A generic software architecture

2.1. AGATA decisional architecture

As stated above, before starting software development, a substantial amount of work has been done designing and validating the foreseen architecture dedicated to autonomy. This new architecture, which separates software functions and standardizes interfaces between functions, is the first step to allow the validation of the complex AGATA generic on-board software for autonomous satellites.

During the early steps of the program a generic decisional architecture has been developed by ONERA to specify the decisional mechanisms of the on-board software. In this modular structure, each component satisfies a specific function and several implementations of the same block may easily be tested. Every component has the same internal architecture allowing both reactive – synchronous – and deliberative – asynchronous – treatments. This ensures a strong partitioning between computations subject to hard real-time constraints on one hand and anytime algorithms such as on-board planning on the other hand.

2.2. A modular approach

As the definition of a complete autonomous system may be very complex, the idea is to describe it as a modular structure, in order to validate each module separately before validating the whole system. 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 parts 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 receives 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).

2.3. AGATA software architecture

Even if the modules are all built following the same pattern, we can distinguish different types of modules. Low-level modules controlling hardware parts are called monitors, the decisions they can make are limited to local control loops: they process the data that are used by higher-level modules. Medium-level modules are in charge of the different functions of the satellite, whereas the highest-level modules control the global behavior of the satellite [1]. This decisional architecture was then derived into a software architecture taking into account the constraints associated to embedded critical real-time software, and language specific constraints (number of software tasks in RTSJ).

Especially for higher-level modules, the decision-making process combines two tasks: a reactive control task and a deliberative reasoning task [2]. 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 that 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.

3. The AGATA development process

3.1. Preliminary – and necessary – works

As described earlier, the generic modular architecture designed for the AGATA onboard software enables the design and validation of different software functions separately. In parallel, the autonomy functions that meant to be integrated in the AGATA software were tested and validated theoretically before the design of the AGATA software. Those autonomy functions along with the concept of decision-making process for the generic architecture modules were prototyped with a synchronous language (Esterel) to validate the new decision mechanisms and autonomy algorithms before they were applied to the real AGATA onboard software [3]. These preliminary works produced a solid basis on which a new development process adapted to autonomy could be designed.

3.2. Model driven engineering

As stated before, the AGATA model-based development process is an iterative and incremental development process based on a Y-shaped cycle, with the objective to incrementally design and validate one by one each software function. We chose to use UML 2 to benefit from semi-formal specification methods. The UML model of the AGATA onboard software is used as the only specification for the software, no external text specifications are used. Our development process is centered on this UML model. To shorten iterations, we also use auto-code generation in order to reduce the validation effort related to implementation. The guideline is to validate as much as possible of the software as early in the development process as possible, even directly at model level when applicable.
We also decided to take advantage of the AGATA demonstrator to evaluate RT-Java as a potential candidate for implementing the onboard software of tomorrow’s satellites, and therefore we decided to perform real-time validation of the AGATA onboard software using RTSJ.
Our model-based development process allows 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 have been validated progressively on the auto-code through successive increments and iterations. The software has been incrementally developed by short specification steps, each leading to incremental OBSW prototypes that have been fully validated

3.3. UML specification

The AGATA OBSW has entirely been specified in UML 2 using Topcased. We have centered our development process on this UML specification that is used to generate documentation as well as OBSW code, or even tests code.

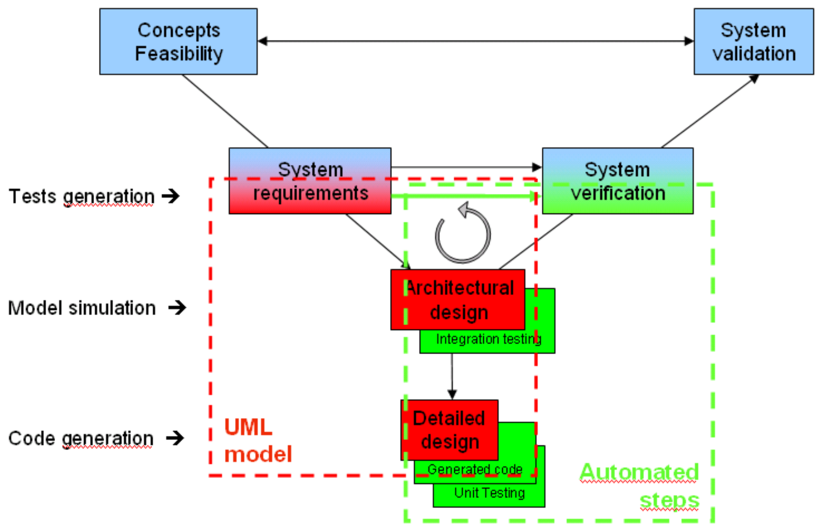

Fig.1: AGATA OBSW model-based “Y-shaped” development process

As shown in the figure above, the UML model of the AGATA onboard software is the heart of our development process. The UML model contains both parts of the system requirements, as well as the architectural and detailed design specification, though different types of diagrams. To specify real-time properties, we used a custom profile, specified by IRIT for CNES in 2006 and called "Autojava" that allows to add real-time behavior to UML components.
Based on the UML specification of the AGATA onboard software, our process contains the following key-steps :

- Model simulation to debug the model, check as soon as possible if critical software functions behave the way they are supposed to and verify that communication between every component in the system is correct ("integration testing" level in the V-shaped development cycle).
- Tests code generation for functional "black-box" validation of the final software ("system verification" level).
- Software code generation to ensure that the final software code is correct (there is no need for "unit testing" as soon as the auto-code generator is qualified).
- Documentation generation to produce the documentation associated to software development (ECSS-E40).

Our process is completely integrated in the Topcased open source UML editor [4] and our development platform is therefore available to anybody interested. Some modules used are part of the basic Topcased distribution (such as Gendoc or TocasedSimu), and the others can be freely downloaded and installed from the Topcased platform.
The next section will describe in more details each one of these steps especially how it fits in the AGATA development process and how it helps developing the AGATA OBSW.

4. Producing the AGATA OBSW

4.1. Model simulation

The first advantage of our model-based approach is the capability to validate the software specification really early in the process (before any code is written or generated) through model simulation. This allows to debug the model – and therefore the specification – and make sure that the final system will behave the way it is supposed to. With model simulation one can make sure that all states in the system are reachable or detect that a guard in an activity diagram will never be met in given circumstances although it should be. Thanks to model simulation such problems can be easily detected and corrected during the specification phase and will not spread to implementation and late validation as it would with classical development processes.
Model simulation also proves to be useful for software integration: running model simulation verifies that communication interfaces between components are correctly implemented by each components, and that connections between components are correctly configured, otherwise the simulation engine detects that a given communication link is not valid.
We used the TopcasedSimu engine developed by ATOS for Airbus and CNES to perform model simulation. This Topcased component animates state machines and activity diagrams to simulate the behavior of the system. It allows the user to send all available events to check how the system will reacts. It offers a classical debugger interface with the ability to execute simulation step-by-step, function-by-function or all at once. Complex and critical functions of the AGATA OBSW were therefore specified using activity diagrams and animated, along with state machines, before code generation to validate the behavior of the system.
The TopcasedSimu component, freely available in Topcased, will continue to evolve in 2012 to support new functionalities such as random or predefined model simulation to avoid asking the user to send external signals and to allow in particular to run non-regression scenarios.

4.2. Tests generation

With the "uml2test" generation module available in Topcased (developed by Astrium for CNES) we have been able to generate "black-box" validation tests from the UML specification of the AGATA OBSW. The "uml2test" module is based on the UML Testing profile and defines a set of stereotypes to specify which system is being tested ("SUT" stereotype), how the operator can send stimulus into the system and how she can observe the behavior of the system [5].
This module generates tests code from a UML test behavior model of the on-board software inherited from the software specification model. This test behavior model mostly expands the software specification model with these stereotypes and with OCL constraints that are added to specify pre-conditions, post-conditions and invariants associated to software functions.
The "uml2test" module interfaces directly with the test bench to execute the tests on the real onboard software. It offers a straight-forward Eclipse interface to command and monitor tests execution while the on-board software is being executed on the test bench. The test bench interface of the "uml2test" module is specific to the test bench, but it has been designed to be almost generic and minimize the parts that have to be redeveloped for new test bench. Although it was initially developed for the Simops test environment (designed for the Pleiades OBSW by Astrium), it required only a few weeks to be adapted to BASILES test environment (AGATA OBSW simulator).

Finally "black-box" tests generated by the "uml2test" module for the AGATA onboard-software have been executed while the software was running on the test-bench. It pointed out lacks in the telemetry cover since some telecommands executions could not be monitored or validated, and it brought us to complete the telemetry definitions.
Although a substantial effort has to be performed to define and implement the test behavior model, the consistency of the final tests plan is improved and, since all the information related to the tests is in the UML model, potential late software modifications will have a reduced effect on validation plans.
We also observed that specify functional tests of the software in a semi-formal language such as UML early in the process helps detecting specification lacks or mistakes. Even before tests generation, it improves the whole process by bringing the designers to ask themselves the rights questions.
The next step to be explored for automated test generation would be to clearly define the "black-box" tests interface for a given on-board software (with generic and specific services) in order to simplify tests development and sequencing. Future projects will also study the connection between scenarios developed for model simulation and those used to generate tests for software validation.

4.3. Code generation

The focal point of our iterative and incremental development process is our auto-code Java/RTJava generator that is integrated in Topcased and uses the "Autojava" profile described in section 3.3. Auto-code generation allows to substantially shorten iterations length by reducing coding and low-level testing time, which enables an iterative and incremental approach.
The RT-Java code generator, based on RTSJ, was initially specified by IRIT for CNES and was then adapted to Topcased environment by Obeo. It allows to generate classical Java code to be executed by standard Java VM, or to generate RT-Java code to be compiled and executed by RTSJ-compliant virtual machine such as JamaicaVM (or AeroVM which is the adaptation of JamaicaVM for ERC32 micro-processor made by Astrium for ESA). Those two versions of the auto-code generator share the "Autojava" profile and can be used separately, depending on the target. In Topcased, the RTSJ generator module is named "uml2rtsj" whereas the classical Java generator is called "Autojava". In the following section both versions of the auto-code generator will be referred to as "Autojava generator".
From the UML specification, the Autojava generator produces the software static architecture along with a middleware to handle communication between software components, including asynchronous communication between tasks. Specific RT-Java code is generated from model information associated to the Autojava profile, such as tasks and ports properties or communication buffers dimensioning. Some of this information associated to the Autojava profile is also used to generate the communication middleware for classical Java. Thanks to Acceleo technology, software code is kept unchanged from previous version and only the new methods and the overall behavior have to be validated.
In the first version of the Autojava generator, code was entirely generated from pseudo-code in the UML model (available with the TAU editor) but we decided to abandon that for direct java coding. In fact TAU pseudo-code has limited expressivity compared to Java and we often had to modify the generated code manually.
Using the Autojava code generator in our model-based development process we have been able to incrementally produce the functional version of the AGATA software. Finally the RT-Java code of the AGATA OBSW has been executed using AeroVM on an ERC32 simulator integrated in the real-time satellite simulator and we were able to perform an evaluation of RT-Java for space application. Although inferior to C, RT-Java proved to be in the same order of performance for its application in the AGATA OBSW (the RT-Java code is actually partially compiled in C-code by AeroVM for execution time optimization purpose and partially interpreted for memory footprint purpose).
Although the current Topcased Autojava generator proved to fulfill its goals, future version will support different communication middleware generation and thereby enhance a building block approach. The idea is to add the capability to specify what kind of middleware to generate for each communication link. The new RT-Java generator currently being developed implements standard communication middlewares in order to encourage future integration of COTS developed outside AGATA. It remains compatible with both classical Java generation and RT-Java generation and, upon completion, will be contributed to Topcased.

4.4. Documentation generation

The last tool used in our model-based development process is the UML documentation generation module "Gendoc2" [6]. This Topcased module uses templates written using Microsoft Word or Open Office to specify how to organize the final document, and then it populates the template with the information of the UML model. As they are based on office software applications, templates are really easy to write and documentation generation, which is based on Acceleo, is straight forward.
Specific documentation of the AGATA OBSW has been generated using the Gendoc2 documentation generation module available in Topcased. However this documentation is not fully compatible with the ECSS E40 recommended documentation since it is missing some information required by ECSS E40 that do not exist in the AGATA OBSW UML model (nothing was manually added to the generated document).

4.5. Formal methods

Although no concrete application of formal methods could be done in our iterative and incremental model-based development process, current works are exploring how to integrate model-checking in our process. Most formal methods are really powerful when applied to simple systems since they produce an analytical solution but they are unable to converge on larger systems.
Nevertheless model-checking remains a promising lead for further improvement of our development process.

5. Conclusion

In conclusion, in the framework of the AGATA program we were able to demonstrate the efficiency of a model-based incremental development process, and the maturity and malleability of Topcased to support such a process. In fact model-based engineering relies largely on tools, such as code generator or model-debugging modules. In our incremental Y-shaped development cycle we used several tools that were developed or evolved for the AGATA OBSW and are now freely available in Topcased. We used documentation generation, model simulation and automated test generation to validate the model before the final functional validation and RT-Java code automated generation to produce the AGATA OBSW from the Topcased UML specification. Our model-based development process proved to be efficient and particularly adapted to the ambitious objective of producing the OBSW for an autonomous satellite.

6. Future works

Future works will address the challenging objective to target the generated RT-Java code of the AGATA OBSW on an evaluation board featuring an ERC32 micro-processor integrated to the satellite simulator. This final step in the AGATA program, scheduled for 2012, will demonstrate that our iterative and incremental model-based development process allowed us to design from scratch the OBSW of an autonomous satellite with limited effort.

7. References

[1] 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
[2] 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
[3] M.C. Charmeau, J. Pouly, E. Bensana, M. Lemaître, “Testing spacecraft autonomy with AGATA”, *iSAIRAS*, 2008
[4] Topcased “The Open-Source Toolkit for Critical Systems” *http://www.topcased.org/*
[5] P. Hyounet, J. Pouly “UML for Validation: Experimenting automatic test generation for flight software validation”, *ERTS*, 2010
[6] Topcased gendoc2 module, *http://gforge.enseeiht.fr/docman/view.php/102/4043/TPC_GenDoc2_v1.1.0_tutorial.pdf*