Towards the Formal Verification of the functional architecture of Autonomous Satellite Onboard Flight Software

Michel Lemaître, Gérard Verfaillie

ONERA Centre de Toulouse – DCSD/CD
2 avenue Édouard Belin, B.P. 4025
F-31055 TOULOUSE CEDEX 4

Abstract

The AGATA project, jointly conducted by CNES and ONERA, investigates the possibility of advanced autonomy for spacecraft. The flight software (FS) is a key component of such an autonomous spacecraft. We are currently investigating the use of the high level synchronous langage Esterel to design the reactive part of the FS. The choice of such a semantically well-defined language has many advantages. One of those is to open the verification process to formal methods. This article reports the effective use of a formal method and tool to verify some typical properties on the preliminary version of the FS for an autonomous observing satellite.

Keywords

Formal verification ; Functional Architecture ; Synchronous languages ; Esterel ; Onboard Flight Software.

1 Introduction

An increased level of autonomy is demanded for spacecraft so as to meet more and more tight cost requirements as well as effectiveness. See for example [7] for arguments.

The AGATA project, jointly conducted by CNES and ONERA, investigates the possibility of advanced autonomy for spacecraft. A virtual mission, named HOTSPOT [6], has been designed to serve as the basis for the project. The HOTSPOT mission is similar to the Earth Observing One mission [8] or the Proba mission [10]. It brings into play an Earth Observing Satellite, boarding two payload equipments : a detection instrument – the hotspotter – and an observation instrument. Thanks to these equipments, the satellite is able to detect hot spots on the Earth surface (fires, volcanoes, ...) and to observe them later on. It must also satisfy observation demands asked by the mission center on ground.

The on-board flight software (FS) is a key component of such an autonomous spacecraft. The FS must react continuously to events. Main events related to the mission are :

- hot spot detections
- observation orders from the ground
- deadlines for observations or downloadings
- monitoring of equipements (fault detection and recovery)

- housekeeping periodic events : orbit parameters updates, maintenance of the catalogue of observed and detected spots, and so on.

Reaction to events are worked out by a set of *reactive tasks*.

Among other tasks, the FS is responsible for planning all the observing and downloading activities of the satellite. This planning activity is worked out by a *deliberative task*, which may not be as responsive as the reactive one.

The AGATA main purpose is to build the functional architecture of the FS and to verify it as much as possible.

The FS currently in construction follows a modular functional architecture described in [13, 12]. Its development obeys the following principles :

- incremental development (successive versions with increasing functionalities)
- use of a simulator for the equipments and the environment of the satellite (BASILES simulator, developped at CNES).

Verifying a system consists in proving that some given properties of a system are met. There are two main ways to do this :

- simulation
- formal verification (see [5] for a good introduction).

This article investigates the possibilities of formal verification of well-chosen properties, in the context of the FS of an autonomous satellite. It is organized as follows. First we describe a simplified functional architecture of the FS. Then we argue in favour of a synchronous executable language to specify the FS. After having exposed the observer principle, which is at the basis of the verification process, we give several examples of different kind of properties that have been verified on a simplified version of the FS.

2 A FLIGHT SOFTWARE FUNCTIONAL ARCHITECTURE

The Figure 1 depicts a simplified functional architecture of the FS. The FS is interfaced with four equipments :

- a stellar-sensor (or star-tracker) – giving the attitude of the satellite
- a GPS – giving the position of the satellite
- an alarm emitter
- a hotspotter, able to detect hot spots on the Earth surface.

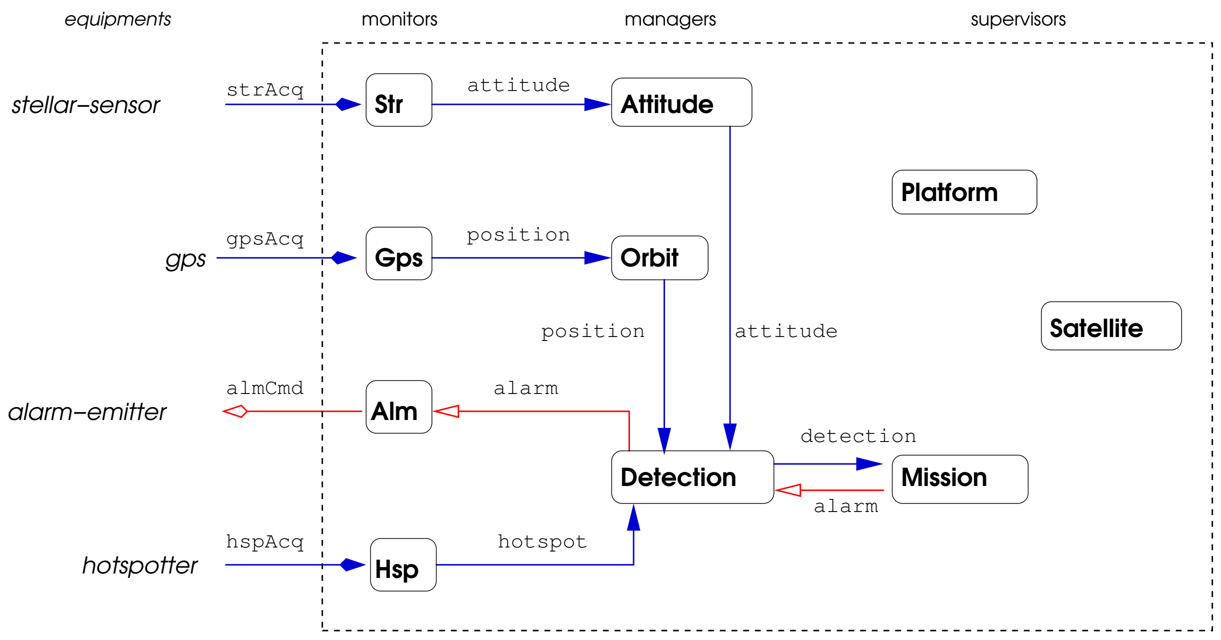

FIG. 1 – Simplified functional architecture of the Flight Software.

Each equipment is controlled and monitored by a corresponding *monitor*. A monitor conducts two main tasks. The first one is to translate low-level acquisition signals into more abstract but logically equivalent ones, and high-level output command signals into low-level ones (see Figure 1, column of signals on the left).

The second task is to perform a minimum level of fault detection and recovery : a monitor checks continuously that its equipment is correctly powered and working. If not, it sends to the equipment a sequence of reset signals until the equipment is again operational. The Figure 2 depicts the control and state signals exchanged between equipments and its monitors. This behaviour is the subject of a vivacity property that will be discussed in section .

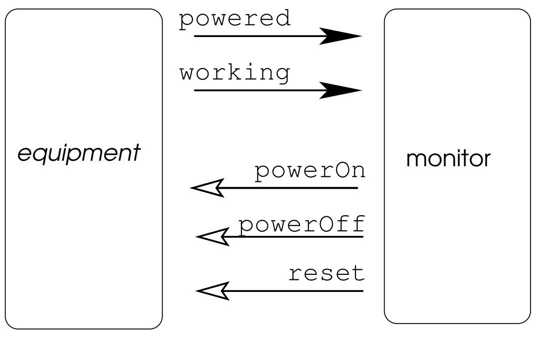

FIG. 2 – Control and state signals exchanged between an equipment and its monitor.

The *managers* and *supervisors* modules (see Figure 1) perform high-level tasks. They are not relevant for the purpose of this article.

3 LANGUAGE DEVELOPMENT CHOICES

There is a demand for accurate and rigourous methods for the development and verification of embedded software. A mean to reach these objectives is to increase the abstraction level of programming languages.

We choose the Esterel language [1, 9, 3] for programming the reactive cyclic controller. Esterel is a reactive synchronous language suited for control dominated systems such as controllers, protocols, embedded systems in general. It is modular, has an imperative style with explicit parallelism and sequencing. Logical threads communicate by signal broadcasting. Its semantics is defined in term of Finite State Mealy Machines. We use the V5 version of Esterel [2] which is free software, and allows for controlling asynchronous external task harmoniously with the language semantics.

The advantages of synchronous languages in general are :

- expressivity and concision
- mathematically defined semantics
- determinism : the same input sequence always results in the same output sequence
- executability
- open to formal verification techniques.

We use Java for implementation of Esterel abstract types, and for prodedural parts of the reactive and deliberative tasks. The Esterel code is compiled to a single Java program, using the ocjava tool [11]. This Java code implements the reactive cyclic loop body.

4 Formal verification on synchronous language code

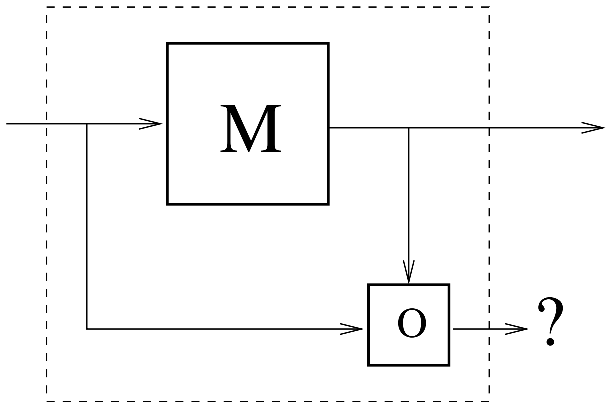

FIG. 3 – The observer technique. M is the program to be verified, o is the program which expresses the property to be verified on M.

Formal verification techniques [5] are able to prove in a mathematical way that given properties of a given program are satisfied, *without executing the program*, that is *for every possible sequence of inputs*.

The *observer principle* is often used in formal verification. The idea is (see Figure 3) to join the program for which some property is to be verified, with another program that checks continuously this property (or the negation of this property). Moreover, advantage is taken of the fact that the properties to be verified on the program are written *in the same language* as the program itself (Esterel in our case).

Many formal verification tools are available. We use Xeve [4, 14].

5 VERIFYING THE REACTIVE PART OF THE FLIGHT SOFTWARE : EXAMPLES

5.1 A reachability property

The first property we want to check is that it is possible to reach a state in which the `almCmd` signal is emitted. This reachability property is directly verified with the verification tool, actually without the need to write any observer (the observer is the output signal itself). The verication tool exhibits a sequence of inputs that reaches a state in which the signal is emitted, hence proving the property.

5.2 A safety property

We want now to check that, if an alarm is emitted (`almCmd`), then a hot spot has been detected by the hotspotter (`hspAcq`). In other words : there are no false alarms. Expressed formally, the property is `almCmd => hspAcq`. The corresponding observer is the following piece of code :

```
loop
   present [ almCmd and not hspAcq ] then
      emit VIOLATION_P1
   end ;
   pause
end loop
```

This observer is composed of a loop construct in which the presence of `almCmd and not hspAcq` is tested. The `pause` instruction just waits for the next reaction cycle. So, this observer code is actually the negation of the property to be verified : if `almCmd and not hspAcq` is present in the same reaction, which is not desired, then the signal `VIOLATION_P1` will be emitted.

The prover formally verifies that `VIOLATION_P1` can never be emitted, for any possible sequence of inputs. So the property is proved.

5.3 A complex bounded vivacity property

The property we want to check now is informally stated as : «any equipment that shows a failing behavior receives a `reset` signal within one second». A first attempt to express this property with an observer is the following.

```
loop
   await [ not powered or not working ] ;
   abort
      await second ;
      emit VIOLATION_P2
   when immediate reset
end loop
```

This observer code is again the negation of the property to be verified. It consists of a loop in which

- we wait first for a failure event (`not powered or not working`)
- if this failure occurs, we wait for the next `second`[1], and then we emit the signal `VIOLATION_P2`, unless a `reset` has arrived before the `second` or is present in the same reaction[2].

In fact, the observer is incorrect, and the verification tool is able to find a sequence of inputs such that the signal `VIOLATION_P2` is emitted ! Looking at this sequence, the designer understands that he/she made two mistakes :

- before checking the desired property, the equipment must have received a `powerOn` signal (otherwise it is of course not powered)
- after a reset, before re-checking the property, one must wait for the first occurrence of the two following events :
  - an elapsed delay — say 3 seconds — for the reset to have a chance to produce an effect on the equipment,
  - the equipment is powered and working again.

So, the desired property is far more complex than expected at first glance. Fortunately, the language allows to express this complex property in a quite understandable way. A correct formulation of the property is the following observer.

```
await powerOn;
abort
   await 3 second ;
when [ powered and working ] ;

loop
   await [ not powered or not working ] ;
   abort
      await second ;
      emit VIOLATION_P3
   when immediate reset ;
```

[1]The `second` signal is supposed to be emitted regularly each physical second by an external clock.

[2]This is the meaning of the `abort - when immediate` construction. If the `second` and the `reset` signals occur in the same reaction, then the body of the `abort` construction is aborted (and `VIOLATION_P2` is not emitted). This behavior is dictated by the `immediate` keyword.

```
   abort
      await 3 second ;
   when [ powered and working ] ;
end loop
```

This time, the prover formally verifies that `VIOLATION_P3` can never be emitted, for any possible sequence of inputs, which proves the property.

6 Conclusion

This article reports the effective use of a formal method and tool to verify some typical properties on the preliminary version of the FS for an autonomous observing satellite. It shows that expressing properties is not always an easy task, but that both the language and associated tools can help.

The limits of the approach are the following.

- The properties to be proven must be explicitly described in the language, and so are limited to those expressable in the language. However, this is not a tight constraint in practice.
- The properties are limited to "boolean" one ; it is not possible with this approach (language and tools) to take into account directly numerical values, nor a continuous time (as in timed automata).

Nevertheless, this verification approach proves its usefulness even for simple properties, because

- it is automatic and complete
- it enforces the specification itself.

References

[1] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. *"The Synchronous Languages Twelve Years Later"*. Proceedings of the IEEE, 91(1):64–83, 2003.

[2] G. Berry. The Esterel V5 Language Primer, version v5_91. École des Mines et INRIA, July 2000.

[3] G. Berry, Amar Bouali, X. Fornari, E. Ledinot, E. Nassor, and R. de Simone. *"Esterel : A formal method applied to avionic software development"*. Science of Computer Programming, 2000.

[4] A. Bouali. Xeve : an Esterel Verification Environment. Technical report, INRIA, 1997.

[5] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. Systems and Software Verification. Model-Checking Techniques and Tools. Springer, 2001.

[6] M.C. Charmeau, J. Pouly, E. Bensana, and M. Lemaître. *"Testing Spacecraft Autonomy with AGATA"*. In 9th INternational Symposium on Artificial Intelligence, Robotics and Automation in Space (ISAIRAS'2008), Los Angeles, California, Feb. 2008.

[7] S. Chien. *"Using Autonomy Flight Software to Improve Science Return on Earth Observing One"*. Journal of Aerospace Computing, Information, and Communication, 2:196–216, Apr. 2005.

[8] Earth Observing-1, Legacy Site. `http://eo1.gsfc.nasa.gov`.

[9] N. Halbwachs. *"Synchronous programming of reactive systems, a tutorial and commented bibliography"*. In Tenth International Conference on Computer-Aided Verification, CAV'98, Vancouver (B.C.), June 1998. LNCS 1427, Springer Verlag.

[10] Proba, Observing the Earth. `http://www.esa.int/proba`.

[11] A. Ressouche and M. Robert. `ocjava`, a Java code generator for Esterel programs. `www-sop.inria.fr/meije/esterel/ocjava.html`.

[12] G. Verfaillie and M.-C. Charmeau. *"A generic modular architecture for the control of an autonomous spacecraft"*. In IWPSS 2006 (International Workshop on Planning and Scheduling for Space), Baltimore, USA, MD, 22 - 25 October 2006. STSI (Space Telescope Science Institute).

[13] G. Verfaillie, M. Lemaître, and M.-C. Charmeau. *"A generic architectural framework for the closed-loop control of a system"*. In 2nd National Workshop on "Control Architectures of Robots: from models to execution on distributed control architectures" (CAR 2007), Paris, France, 31 May-1 June 2007. Université Pierre et Marie Curie.

[14] `http://www-sop.inria.fr/meije/verification/Xeve/`.