HAL
archives-ouvertes.fr

Formal Modelling and Safety Analysis of an Avionic Functional Architecture with Alloy

Julien Brunel, David Chemouil, Vincent Ibanez, Nicolas Meledo

▶ To cite this version:

Julien Brunel, David Chemouil, Vincent Ibanez, Nicolas Meledo. Formal Modelling and Safety Analysis of an Avionic Functional Architecture with Alloy. Embedded real-time software and systems (ERTS^2 2014), Feb 2014, TOULOUSE, France. hal-02272135

HAL Id: hal-02272135

https://hal-onera.archives-ouvertes.fr/hal-02272135

Submitted on 27 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Formal Modelling and Safety Analysis of an Avionic Functional Architecture with Alloy

Julien Brunel[1], David Chemouil[1], Vincent Ibanez[2], and Nicolas Meledo[2]

[1]Onera/DTIM , F-31055 Toulouse, France, firstname.lastname@onera.fr
[2]Thalès Avionics, F-33185 Le Haillan, France,
firstname.lastname@fr.thalesgroup.com

Abstract

We propose an approach based on Alloy to formally model and assess a system architecture with respect to system-level safety requirements. The system on which we instantiate our approach is a specific Required Navigation Performance system from a Thalès Avionics named Localizer Performance with Vertical guidance Approach (LPV). In this article, we describe how to define such a system architecture and how to verify safety objectives.

1 Introduction

This work has been produced in the context of the Artemis European project CESAR, ended in 2012. The aim of CESAR was to boost cost efficiency of embedded systems development and safety and certification processes in the different domains: automotive, aerospace, rail and automation. To achieve this goal, CESAR promoted model-based techniques and formal methods.

In this context, Thales Avionics and Onera collaborated on applying formal methods in order to assess avionic architectures. Indeed, in the avionic domain, the definition of the architecture is one of the main activities in the development phase. The architecture must be defined very early and must validate different constraints such as physical reuse, safety, security, etc. . .

To help avionics architects to define the final architecture, modelling facilities are currently deployed and linked to early validation capabilities. The aim of this work is to study the benefit of a formal method in order to assess models of avionic architectures with respect to some specific high-level safety requirements. We will use the formal language (and its associated tool) Alloy and apply our approach on a specific RNP approach called Localizer Performance with Vertical guidance Approach (LPV approach).

2 LPV case study

An approach with vertical guidance (APV) uses lateral and vertical guidance, and does not meet the requirements established for precision approach and landing operations. Depending on the type of APV procedure, lateral guidance can be provided from either stand-alone Global Navigation Satellite Systems (GNSS), or by RNAV derived positioning using multi-sensor positioning capability. Vertical guidance can be provided from GNSS augmentation system such as SBAS (or possibly Galileo in the future) or a barometric reference.

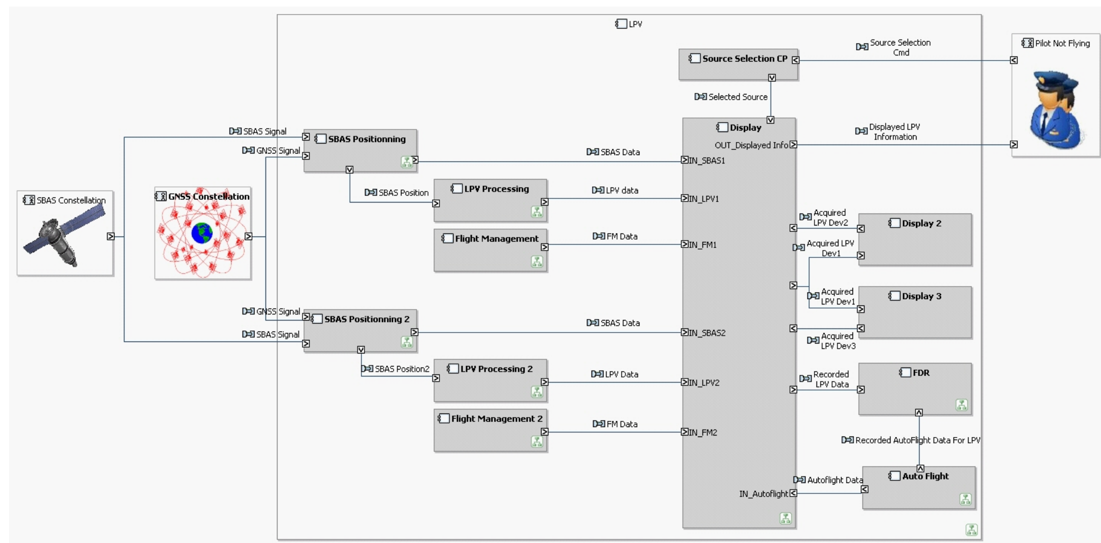

Figure 1: LPV architecture

APV procedures are based on two different concepts, which allow to reduce minimums compared to conventional Non-Precision Approaches. On the one hand, the APV-BARO (highly developed in the mainline segment) relies on navigation systems with high integrity baro-altimeter equipment with temperature compensation. On the other hand, the second concept called APV-SBAS relies on a satellite based augmentation system that improves the navigation system localization through the enhancement of GNSS localization precision and error correction.

LPV approach is characterized by a Final Approach Segment (FAS). FAS is the approach path which is defined laterally by the Flight Path Alignment Point (FPAP) and landing Threshold Point/Fictitious Threshold Point (LTP/FTP), and defined vertically by the Threshold Crossing Height(TCH) and Glide Path Angle (GPA).

In this article, we consider the functional architecture of the LPV system and assess the two following safety objectives:

- *Loss of LPV capability*. The architecture must have two independent ways to provide LPV data.
- *Misleading information integrity*. The architecture must control the value of the LPV data provided by each calculator and between each screen and find mitigation in case of erroneous data.

3 Evaluation of the case study with Alloy

Alloy [2] is a formal system-modelling language amenable to automatic analyses. Alloy is implemented as a cost-less free-software tool, the Alloy Analyzer, which is programmed in Java and hence runnable on the majority of platforms. With respect to the safety objectives we are going to assess, the AltaRica [1] language would have been another possible choice. However, we

decided to take benefit from the model-based aspect of Alloy and its expressiveness for the specification of the properties to check. Indeed, Alloy allows to define easily the metamodel of the avionic architectures we will analyze instead of encoding them in terms of AltaRica concepts. Moreover, the specification of the properties we want to check are expressed in (a relational extension of) first-order logic with many features adapted to model-based reasoning.

To represent the functional architecture of LPV (see Fig 1) in Alloy, we identified the major concepts at stake and defined an Alloy signature for each of them: Function, Port, IPort (input port), OPort (output port). Each Function has a set input of IPorts and a set output of OPort as attributes. An OPort of a function is related to a set of IPort of other functions through a flow and to a set of IPorts of the very function through the relation dependsOn in order to express which input ports can influence an output port (in term of failure propagation). Each function and each port hold a dysfunctional status (OK, Lost, or Err). Moreover, each port has a value. Notice that, in our model, the value will only be useful for certain ports, representing the pilot selection, the discrepancy between both LPV processing, and the reset of displays. Concerning the other ports, we only deal with their status and the value is just ignored (a finer modelling would have been possible here but it would have led to a worthless, more complex model). The following Alloy code corresponds to these concepts definitions (see also Fig. 2).

```
enum Status { OK, Err, Lost }
abstract sig Port {
        status : Status,
        value : Value
}
abstract sig Port {
        status : Status,
        value : Value
}
abstract sig IPort extends Port {}
abstract sig OPort extends Port {
        flow : set IPort,
        dependsOn : set IPort
}
abstract sig Function {
        input : set IPort,
        output : set OPort,
        status : Status
}
```

We then define instances of these concepts corresponding to the LPV functional architecture. The function instances take into account the selection of the source by the crew (SelectSource), two occurrences of SBAS positioning (ComputeSBAS1, ComputeSBAS2), two occurrences of LPV processing (ComputeLPV1, ComputeLPV2), three occurrences of displays (Acquirei, $i \in \{1..3\}$), three occurrences of display resetters (Crosschecki, $i \in \{1..3\}$) and of monitors in order to launch an alarm, (Monitori, $i \in \{1..3\}$). We also define the different ports of each function, and the way ports are related to each other via flows. For instance, the following Alloy code is an excerpt of the flow definition, expressing that the output port oSBAS1 is related to the input port iSBAS1 via a flow (idem for oSBAS2 and iSBAS2).

```
flow = oSBAS1→iSBAS1 +oSBAS2→iSBAS2 +...
```

We also define some global constraints the architecture must satisfy, such as the fact two ports related by a flow share the same status and the same value:

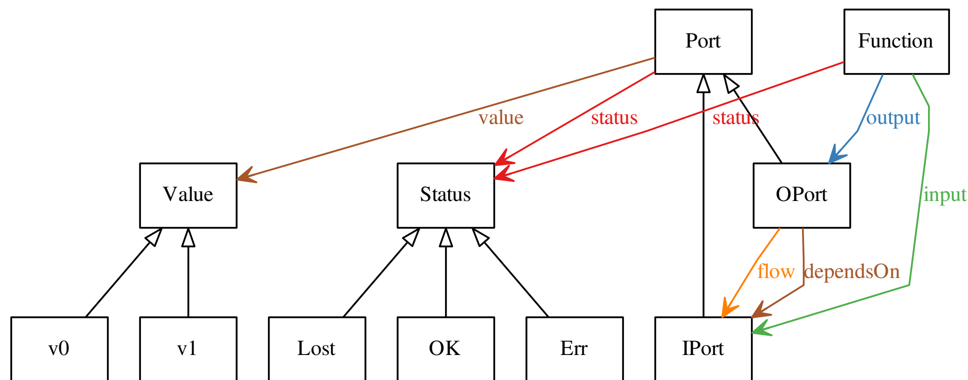

Figure 2: Case study metamodel (simplified)

all p1, p2 : Port | p1→p2 **in** flow **implies** p1.status = p2.status **and** p1.value = p2.value

We now define the relation between input and output ports inside each function in term of failure propagation. For instance, the following code expresses that the status of the output port oDeviation1 of function ComputeLPV1 is equal to the status of the input if the function ComputeLPV1 is OK, is equal to Lost of the function ComputeLPV1 is lost, and is erroneous (Err) otherwise.

```
let st = ComputeLPV1.status |
oDeviation1.status = {
                    st = OK implies iSBAS1.status
                    else st = Lost implies Lost
                    else Err
}
```

The following code defines the status of the output port oSelected1 of function Aquire1. If this function is OK, oSelected1 the status is either equal to the status of its first input or to the status of its second input, depending on the selection made by the pilot. If the function Aquire1 is lost, then the status of output oSelected1 is lost. Otherwise, it is erroneous.

```
let st = Acquire1.status |
let v = iSelection1.value |
oSelected1.status = {
                   st = OK and v = v0 implies iDeviation11.status
                   else st = OK and v = v1 implies iDeviation21.status
                   else st = Lost implies Lost
                   else Err
}
```

Similarly, we define, for each function, output ports status (and value in the case of pilot selection, discrepancy, and display reset) from input ports status (and value).

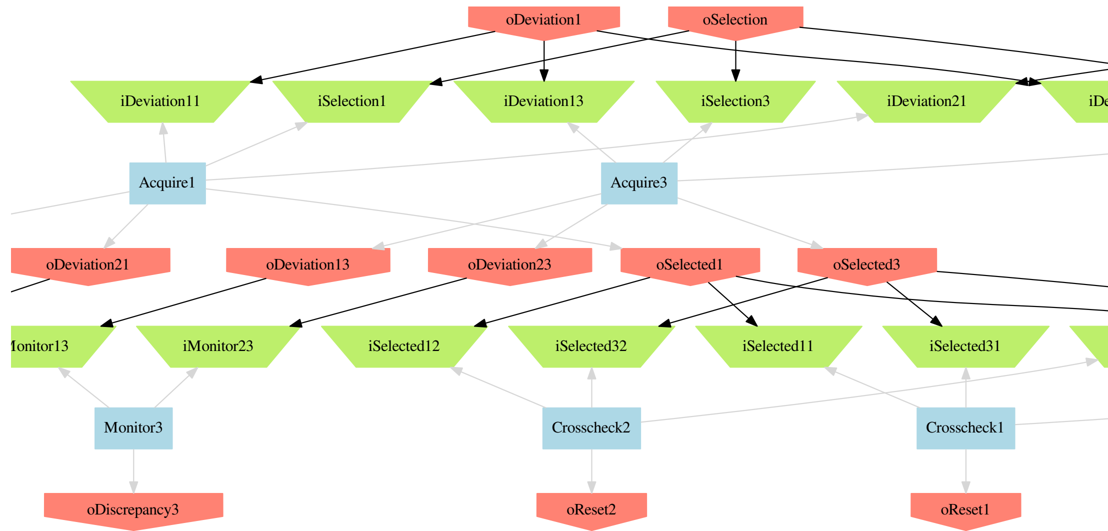

Figure 3: Visualization of LPV functional architecture in the Alloy Analyzer (excerpt)

Then, Alloy Analyzer proposes, with the command **run** to produce automatically an instance that satisfies all the definitions we expressed, and the global constraints. Of course, this is only possible if there is no inconsistency in the definitions. We can customize the way the instance is graphically displayed. Fig. 3 shows an excerpt of an instance produced by the Analyzer, where functions are represented by blue rectangles, output ports by red "inversed houses", and output ports by green inversed trapezoids. The status and values are not displayed.

We are now able to check formally various properties expressed in Alloy. A first kind of properties we can check consists of structural properties about the model. For instance, the fact that a port can only belong to one function, the fact that flows correspond to one-to-many communication, and the fact that ports that are related by the dependsOn relation belongs to the same function, is expressed by the following properties:

```
assert model_structure {
  input in Function one →IPort
  output in Function one →OPort
  flow in OPort one →IPort
  all op : OPort, ip : op.dependsOn | ip in (output.op).input
}
```

The command **check** model_structure verifies that, *up to a certain bound*, all possible instances of the model satisfy these properties. If it is not the case, it yields a counter-example instance.

Now we want to validate the safety objectives expressed in Sect. 2. Concerning the *Loss of LPV capability* constraint, we express the following property:

• *If one (and only one) LPV processing is lost, and if the pilot does a correct selection, then the data sent by the three displays are still correct.* This corresponds to the following Alloy code (for the loss of LPV1):

```
assert one_computer_lost {
  (all f: Function | (f ≠ComputeLPV1 implies f.status=OK)
                     and ComputeLPV1.status=Lost and oSelection.value=v1)
  implies oSelected1.status = OK
        and oSelected2.status = OK
        and oSelected3.status = OK
}
```

The command `check one_computer_lost` verifies that the model satisfies this property.

Concerning the *Misleading information integrity* constraint, we expressed two properties to be satisfied by the functional architecture:

• *If one LPV processing produces erroneous data, then an alarm (modeled by the variables oDiscrepancy) is launched on the three displays.* This corresponds to the following Alloy code (for the LPV1 in erroneous failure mode):

```
assert one_computer_erroneous {
  (all f: Function | (f ≠ComputeLPV1 implies f.status=OK)
                     and ComputeLPV1.status=Err)
  implies oDiscrepancy1.value=v1
        and oDiscrepancy2.value=v1
        and oDiscrepancy3.value=v1
}
```

• *If one display returns an erroneous data, it resets itself.* This corresponds to the following Alloy code (for display 1 (Acquire1) in erroneous failure mode:

```
assert one_display_erroneous {
  (all f: Function | (f ≠Acquire1 implies f.status=OK)
                     and Acquire1.status=Err)
  implies oReset1.value=v1
}
```

All the properties we have expressed in this section are validated by Alloy Analyzer.

A last aspect we verified concern the "functional chains", used by Thales to support safety analyses. A functional chain is sequence of functions and ports linked by flows that lead to a given output port. If we know all the functional chains leading to a given output port viewed as a target data, it gives information about all the functions whose failure could have an impact on the integrity of the target data. It turns out from discussions that these functional chains are specified by engineers by hand. We propose here a way to generate all the functional chains leading to a given output port (this set of functional chains defines what we call a functional tree). The following Alloy code defines what a functional tree is. It is defined by an *inductive* relation *next* and a given output port target satisfying the following constraint:

```
abstract sig FTree {
  next : Port→ Port,
  target : OPort
}
fact {
    all ft : FTree, p0, p1 : Port | p0→p1 in ft.next iff {
          (p1 = ft.target and p0 in p1.dependsOn) or
```

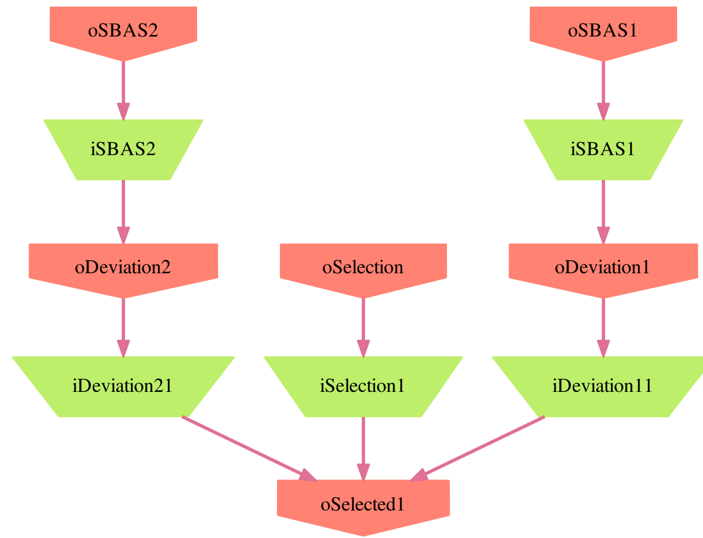

Figure 4: Example of a functional tree

```
            (some p2 : OPort | p1→p2 in ft.next and p1 in p0.flow) or
            (some p2 : IPort | p1→p2 in ft.next and p0 in p1.dependsOn)
    }
}
```

So that Alloy Analyzer generates a functional tree whose target is the output port oSelected1 (output of function Aquire from display 1), we have to specify a simple predicate expressing that there is a functional tree whose target is oSelected1, and then to ask for an instance satisfying this predicate, with the command run.

```
pred functional__tree__oSelected { some ft : FTree | ft.target = oSelected1 }
run functional__tree__oSelected
```

This command produces an instance displayed by Alloy analyzer as showed by Fig. 4

4 Conclusion

In this article, we modeled the LPV functional architecture in the Alloy language. We then expressed different properties from the safety objectives in Alloy language. We checked with Alloy analyzer that these properties are fulfilled by the model. We also proposed a definition that allows to generate all the functional chains leading to a given data, in order to support some safety analyses.

References

[1] A. Arnold, A. Griffault, G. Point, and A. Rauzy. The altarica formalism for describing concurrent systems. *Fundamenta Informaticae*, 2-3:109–124, 2000.

[2] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.