Model-Checking Real-Time Properties of an Auto Flight Control System Function

Pierre-Alain Bourdil, Bernard Berthomieu, Éric Jenn

Model-Checking Real-Time Properties of an AutoFlight Control System Function

Pierre-Alain Bourdil * † ‡, Bernard Berthomieu * ‡, Eric Jenn †
* CNRS, LAAS, 7 avenue du Colonel Roche, F-31400 Toulouse, France
† Thales Avionics, 105 av du Général Eisenhower, F-31100 Toulouse, France
‡ Univ de Toulouse, LAAS, F-31400 Toulouse, France
Email: {Pierre-Alain.Bourdil|Bernard.Berthomieu}@laas.fr, Eric.Jenn@fr.thalesgroup.com

***Abstract*—We relate an experiment in modeling and verification of an avionic function. The problem addressed is the correctness of a temporal condition enabling the detection of a range of faults in the implementation of the function. Using the Fiacre/Tina verification toolset, we produced a formal model abstracting the function, and confirmed by model-checking that the condition determined analytically is indeed correct. The modelling issues ensuring tractability of the model are discussed.**

I. Introduction

We relate an experience in modeling and verification of a function part of an AutoFlight Control System.

The system has safety and availability requirements, similar to [1]; we focus on safety issues. A distinctive feature of the system is that it is conceptually asynchronous; events may occur at unspecified times. But, for tractability and engineering reasons, it is designed as a logically synchronous system. This is typical of many IMA architectures, for instance, in which processing resources execute asynchronously relative to each other [2]. Consequently, non-functional constraints arise, capturing the temporal properties and constraints under which the system must operate [3].

A practical solution to solve these problems is to maintain events as long as necessary to cover asynchrony. The main objective of our work is to formally compute and check how long an event must be maintained. These non-trivial tasks are currently done analytically by experienced engineers. In our experiment, this is done by model-checking using the Fiacre/Tina tool suite, which is well suited here since the formalism underlying Fiacre/Tina models is timed and asynchronous. It is however necessary, by a suitable abstraction, to prevent combinatorial explosion.

Our modeling and verification of the function enable early design flaw detection. Moreover, our design leads to tractable state spaces on which the candidate durations for events persistence can all be be checked by exhaustive analysis.

The use case is introduced in the next section. Section III overviews the Fiacre/Tina verification environment. Section IV discusses modeling issues; our verification results are summarized in Section V.

II. Use Case Presentation

A. AutoFlight Control System (AFCS)

The function discussed in the paper is part of a generic AutoFlight Control System (AFCS). The AFCS provides four main capabilities to support the operational use of an aircraft : the Autopilot (AP), the Auto throttle (AT), the Flight Director (FD), and finally the Flight Guidance (FG). We focus on the latter.

The Flight Guidance (FG) allows the pilot to operate the aircraft according to high-level scenarios, called "modes". Example of modes are "capture a target altitude", "maintain a target vertical speed", "achieve a target thrust value", etc. The logic of interactions are beyond the scope of this paper; we focus on the altitude target acquisition function in mode "reach an higher altitude".

The architecture is meant to deal with safety issues related to a range of faults [1], thus there is a Command channel and a Monitoring channel (COM/MON). However we do not consider availability aspects here (redundancy, fail-safe operation or hot spares resources). We verify the correctness of the temporal condition enabling the detection of faults.

B. Architecture overview

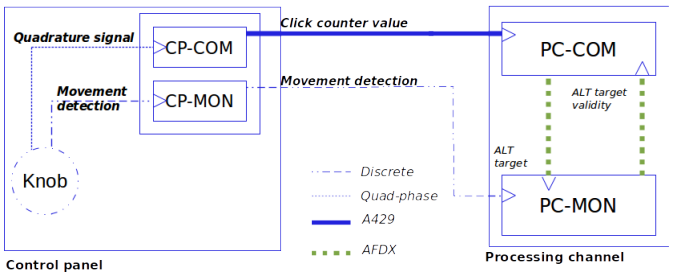

Fig. 1: Architecture of the Altitude Target Acquisition

a) The Control Panel: It is constituted (Fig. 1) of a knob, a command component (CP_COM) and a monitoring component (CP_MON). CP_COM and CP_MON share a common clock and starts synchronously. The knob has 256 positions, or "clicks", each with tactile feedback. To select an altitude target the pilot rotates the knob, in either direction, which in turns emits two signals: 1. A phase-quadrature encoded signal (+1 per click when turning clockwise, −1 when turning counterclockwise) and

2. An indication of movement (high if movement, low if no movement). CP_COM sends periodically the count of clicks modulo 256 to PC_COM through an A429 communication bus [4]. Each time the indication of movement is high, CP_MON outputs a "high" signal through a discrete wire. It maintains this signal high for $cProl$ ms after the indication of movement has fallen (see Figure 3).

b) The Processing Channel: It is constituted of two components: a command component PC_COM and a monitoring component PC_MON both implemented using periodic tasks with different phases, periods and durations. Each execution of such a task, called a cycle, starts with atomic reading of its inputs. Outputs are written atomically at the end of the cycle. PC_COM and PC_MON communicate asynchronously using the "sampling port" mechanism of AFDX [5] (i.e. messages are not buffered).

At each cycle PC_COM computes a target altitude based on the clicks count received from CP_COM and sends this target to PC_MON for validation purposes.

PC_MON sets a boolean local variable, alt_change, each time the current altitude read from its AFDX input is different from that of the preceding cycle. If the current altitude equals that of the preceding cycle, alt_change is set to false. PC_MON also sets a boolean local variable, $mvt\_prolong$, each time the signal on its discrete wire input is high. PC_MON sets $mvt\_prolong$ to false $pProl$ ms after the last cycle in which the signal was high.

c) Verification goal: We consider only one fault: PC_COM computes a new altitude target while the knob has not moved. Due to asynchrony, PC_MON may miss a knob movement, or may not see the movement at the same time as PC_COM. PC_MON, the control panel, and the communication networks are assumed reliable. Our verification goal is to prove that the model $\mathcal{M}$ of the system, instantiated for the times $\Gamma = (cProl, pProl)$, satisfies the linear time temporal property $DETECT$ below. If satisfied, then using that pair Γ in the actual system ensures the detection of faults from PC_COM.

$$\Box(alt\_change \Rightarrow mvt\_prolong) \qquad (DETECT)$$

III. Fiacre and Tina

A. Modeling with the Fiacre language

Fiacre [6] was designed as a joint effort by several research teams in the context of project TOPCASED [7], with the purpose of serving as an intermediate language between user notations like UML, SysML or AADL and verification tools like Tina [8] or CADP [9], providing a behavioral semantics and verification capabilities to the formers. The language has been since developed in several other projects and put to work in several verification toolchains, notably [10], [11].

Fiacre is a language in the vein of Promela or BIP. Fiacre programs are structured into *processes*, modeling sequential activities, and *components*, describing a system as a composition of processes or other components. Fiacre supports the two most common coordination paradigms: by shared variable and by asynchronous message-passing.

We illustrate the main concepts of Fiacre through a partial modeling of the knob, which is a phase quadrature encoder, and the processing and filtering of its outputs by the Control Panel.

Fiacre *processes* are defined from a set of parameters and control states, each associated with a set of symbolic *transitions* (following keyword `from`). The initial state is the source state of the first transition. The transitions declare how variables are updated, which events may occur, and when. They are built from standard deterministic programming language constructs, non-deterministic constructs (such as external choice, operator `select`), communication statements, temporal constraints (construction `wait`) and jumps to a state (keyword `to` or `loop`). For example, Fig. 2 shows a Fiacre model of the encoder movements as a process *KNOB* with two states. The transition from state *move* expresses a synchronization on port *down*, to be matched with that occurring in the transition from state *waitdown* in process *CP_MON*. From state *sustain* in *CP_MON*, two transitions are possible, non-deterministically. The first may only happen if no synchronization on *up* occurred for duration *cProl*.

```
type Clicks is union Change | NoChange end

process KNOB [up,down:sync] (&ck:Clicks) is
states idle,move
from idle
  up; ck:=Change; to move
from move
  down; ck:=NoChange; to idle

process CP_MON [up,down:sync] (&cMvt:bool) is
states idle,sustain,waitdown
from idle
  up; cMvt:=true; to waitdown
from waitdown
  down; to sustain
from sustain
  select
     wait [cProl,cProl]; cMvt:=false; to idle
  [] up; cMvt:=true; to waitdown
  end

component CONTROL_PANEL (&cMvt: bool) is
var ck:Click := NoChange
port up,down:sync
par * in
    KNOB [up,down] (&ck)
|| CP_MON [up,down] (&cMvt)
|| CP_COM (&ck)
end
```

Fig. 2: Fiacre model of the Control Panel

Components are built as parallel compositions of processes and/or other components (by operator `par` P_0 `||` ... `||` P_n `end`). Compositions specify both the process or component instances and their interactions. Shared variables and communication ports are within components. Communication ports may be associated with time constraints, applying to all interactions through that ports and with priorities. The ability to express timing constraints in programs is a distinguishing feature of Fiacre.

Finally, Fiacre comes with a rich language for expressing linear time properties. For improved readability, the properties can make use of the specification patterns as in [12] and of a realtime extension of them discussed in [13].

B. Behavioral verification with Tina

Tina(TIme Petri Net Analyzer) [8] is a software environment to edit and analyze enriched Time Petri Nets. The core of the Toolbox is an exploration engine generating state space abstractions; these abstractions are then fed to model-checking or equivalence checking tools. The front-end converts models into an internal representation — Time Transition Systems (TTS) — an extension of Time Petri Nets with data and priorities. A compiler, *frac*, converts Fiacre description into TTS descriptions, therefore enabling model-checking of Fiacre specifications by Tina.

A TTS example specification is shown in Fig. 3. It corresponds to the interpretation of the Fiacre process CP_MON from the control panel modeled in Sect. III-A. A TTS can be viewed as a Time Petri Net where transitions are decorated with guards (`pre`) and actions (`act`) on data variables. Compared to Time Petri nets, a transition in a TTS is enabled if both: (1) its preconditions are fulfilled, in terms of tokens in its input places; and (2) the predicate `pre` is true for that particular transition. When a transition fires, the store is updated atomically by executing the corresponding action `act`.

The TTS in Fig. 3 models the prolongation of the movement signal by CP_MON. Transition t_0 is synchronized through the label *up* with a transition from the model of the knob (not shown here). When the knob starts moving, t_0 fires and sets $cMvt$. This variable models the prolonged signal. When the knob stops moving, t_1 fires. From state `sustain`, and if no knob movement occurs within interval $[cProl, cProl]$ then t_2 fires and $cMvt$ is reset, else t_3 fires. The signal is thus prolonged for $cProl$.

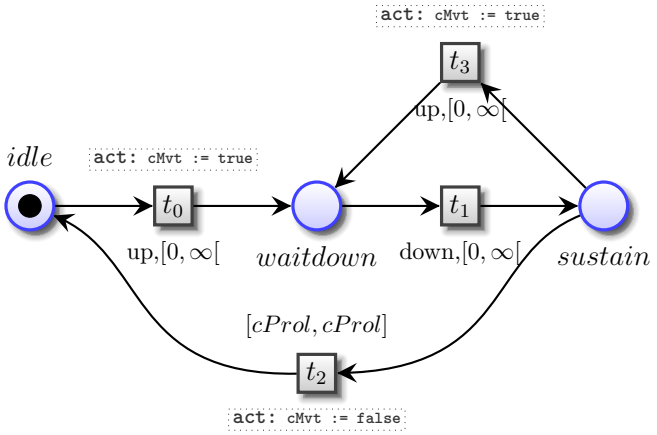

Fig. 3: CP_MON behavior

Time is assumed dense, so TTS typically have infinite state spaces; model-checking them require finite abstractions. Tina offers several such state space abstractions, based on so-called state classes, preserving specific families of properties like absence of deadlocks, state reachability, or formulas of some temporal logics. The properties are checked on the abstraction computed. One of the tools provided by Tina for this is *selt*, a model-checker for State/Event-LTL, a linear time temporal logic supporting both state and transition properties. For the properties proved false, a timed counter example is generated that can be replayed in the TTS simulator.

IV. Modeling the altitude target acquisition function

This section discusses the modeling of the use case. Due to the lack of space, we do not detail the modeling of synchronous components nor communication mechanisms. Instead we present modeling techniques that, put together, yield a tractable state space, allowing to check the effects of choice of constants Γ on property $DETECT$.

We observe that $DETECT$ is insensitive to specific altitude target values. The model may reflect this by considering change of altitude only. Recall from Section II that altitude increments are sent periodically from CP_COM as an integer modulo 256. We abstract these distinct altitude values as a boolean data type $\{Change, NoChange\}$. This classical abstraction is quite effective in decreasing the number of discrete states of the system.

Next, from this data abstraction we derive a temporal abstraction. First we abstract the knob's direction of rotation. Second, we abstract the quadrature-phase signal into a signal with two levels (high or low).

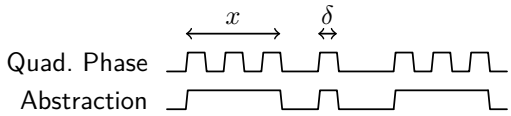

Fig. 4: Knob movement abstraction

In Figure 4, the first chronogram shows a possible behavior of the quadrature phase signal, neglecting the direction of this signal, where x represents a sequence of 3 knob clicks; δ is the duration of a click. Recall that it is always possible for the pilot to move the knob. The problem is that the PC_COM period is much longer than δ and that the PC_COM task may be executed infinitely often: this difference of temporal scales is a major source of combinatorial explosion. Intuitively, many knob moves may occur between two PC_COM executions.

Our abstraction is illustrated in Figure 4. Instead of modelling the knob clicks (top chronogram), we just model knob moves (bottom chronogram), and without time constraints between successive moves. This abstraction comes at the cost of a Zeno behavior however: infinitely many moves are allowed to occur in a finite amount of time. Our safety property $DETECT$ is insensitive to such Zeno behaviors, however. This abstraction simplifies the time constraints involved when computing the system state classes with Tina and consequently greatly decreases their number.

From the globally asynchronous nature of the system arises two classical asynchrony problems: clock deviation and asynchronous initialization of components. 1) Time Petri Net based models cannot natively represent clock deviations. We make the assumption that the system's up-time is sufficiently small to not introduce erroneous behaviors. This is a commonly accepted engineering assumption, and clocks are monitored in flight to ensure the assumption holds. 2) Asynchronous initialization leads to phase shift between components. Obviously phase shifts

are a major cause of combinatorial explosion. We adopt a divide-and-conquer strategy for this problem. Instead of one model encompassing all possible initialization orders and phase shift combinations, we consider one model per initialization order expressing all possible phase shifts given that order.

Finally, since $DETECT$ is a safety property, it can be checked in Tina using a construction preserving only discrete states (rather than states and traces), which typically yields considerably smaller behavior abstractions. Intuitively, this construction attempt to agglomerate sets of states related by entailment of their timing constraints. It proved to be very efficient in out experiments.

V. Experimental results

As briefly discussed in the previous section we have parameterized our model, ending with a template model which can be instantiated according to some parameters: $pProl$, $cProl$, initialization orders, component periods and durations, networks latencies, etc.

The concrete parameter values are extracted from the specifications of the system. $pProl$ and $cProl$ ranges from 1 to 6 cycles of PC_MON; 4 possible initialisation orders are considered. Thus we had to generate 144 models. For each model we computed its set of discrete states using the Tina construction discussed above. Their sizes are 2 millions states on average, occupying 2 Mb on disk. The overall verification for the 144 models run in less than 2 hours on a typical workstation.

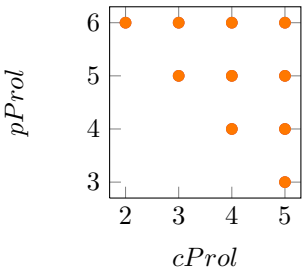

Fig. 5: Configurations for which $DETECT$ is satisfied.

Figure 5 pictures the results of our experiment. Each dot represents a value of $(cProl, pProl)$ for which the model satisfies $DETECT$, whatever the initialization order is. These are expressed in periods of PC_MON. The value pairs investigated include those found analytically by experienced engineers.

We then explored pairs $(cProl, pProl)$ for which the property is not satisfied, in order to obtain counter-examples. These constitute a crucial information for the engineers as testing methods could not necessarily uncover these scenarios. For this, we used the default state class construction of Tina, preserving discrete states and traces. The state spaces obtained average in this case 20 millions states in size, occupying about 60 Mb on disk. Using the Tina tools suite, we could then obtain the shortest timed counter-examples. These average 200 states in length, and so can still be analyzed by hand. It must be noticed that these counter-examples cover real errors found late in the system life cycle during the validation phase.

VI. Conclusion

We summarized in this paper an experiment on modeling a real time system and verification of non-functional properties using Fiacre/Tina. From this model, extracted from the actual specifications of the system, we exhibited error scenarios corresponding to real errors found late in the system's development lifecycle, the validation phase. Moreover, we could use this model to check a crucial non-functional condition ensuring the detection of faults. It took two man-month of work to conduct the whole experiment; most of this time being devoted to the extraction of relevant information from a text-based industrial specification. A necessary improvement of our work would be to include availability issues such as Failsafe COM/MON since crucial non-functional properties arise from this architecture pattern. Nevertheless our work demonstrate that formal verification is readily applicable to industrial problems using out-of-the-box available tools.

References

[1] P. Traverse, I. Lacaze, and J. Souyris, "Airbus fly-by-wire: A total approach to dependability," in *Building the Information Society*. Springer, 2004, pp. 191–212.

[2] S. Miller, D. Cofer, L. Sha, J. Meseguer, and A. Al-Nayeem, "Implementing logical synchrony in integrated modular avionics," in *Digital Avionics Systems Conference, IEEE/AIAA 28th*, oct. 2009, pp. 1.A.3–1 –1.A.3–12.

[3] A. Gamatié, C. Brunette, R. Delamare, T. Gautier, and J.-P. Talpin, "A modeling paradigm for integrated modular avionics design," in *Soft. Engineering and Advanced Applications, 32nd EUROMICRO Conference on*. IEEE, 2006, pp. 134–143.

[4] Aeronautical Radio Inc., *ARINC specification 429-ALL: Mark 33 Digital Information Transfer System Parts 1,2,3*, 2001.

[5] ——, *ARINC 664, Aircraft Data Network, Part 1: Systems Concepts and Overview*, 2002.

[6] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, F. Vernadat *et al.*, "Fiacre: an intermediate language for model verification in the TOPCASED environment," in *Embedded and Real-Time Software, Toulouse*, 2008.

[7] F. Vernadat, C. Percebois, P. Farail, R. Vingerhoeds, A. Rossignol, J.-P. Talpin, and D. Chemouil, "The TOPCASED Project - A Toolkit in OPen-source for Critical Applications and SystEm Development," in *Data Systems In Aerospace (DASIA), Berlin, Germany*. ESA Publications, may 2006.

[8] B. Berthomieu, P.-O. Ribet, and F. Vernadat, "The tool TINA–construction of abstract state spaces for Petri nets and time Petri nets," *International Journal of Production Research*, vol. 42, no. 14, pp. 2741–2756, 2004.

[9] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "Cadp 2010: a toolbox for the construction and analysis of distributed processes," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2011, pp. 372–387.

[10] B. Berthomieu, J.-P. Bodeveix, S. Dal Zilio, P. Dissaux, M. Filali, P. Gaufillet, S. Heim, F. Vernadat *et al.*, "Formal verification of AADL models with Fiacre and Tina," *Embedded Real-Time Software and Systems, Toulouse*, pp. 1–9, 2010.

[11] S. Rangra and E. Gaudin, "SDL to Fiacre translation," *Embedded Real-Time Software and Systems, Toulouse*, 2014.

[12] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *International Conference on Software Engineering*, 1999, pp. 411–420.

[13] N. Abid, S. Dal Zilio, and D. Botlan, "Real-time specification patterns and tools," in *Formal Methods for Industrial Critical Systems, Springer LNCS 7437*, 2012, pp. 1–15.