Model-Checking Real-Time Properties of an Aircraft Landing Gear System Using Fiacre*

B. Berthomieu[1,2], S. Dal Zilio[1,2], and Ł. Fronc[1,2]

[1] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[2] Univ de Toulouse, LAAS, F-31400 Toulouse, France

Abstract. We describe our experience with modeling the landing gear system of an aircraft using the formal specification language Fiacre. Our model takes into account the behavior and timing properties of both the physical parts and the control software of this system. We use this formal model to check safety and real-time properties on the system but also to find a safe bound on the maximal time needed for all gears to be down and locked (assuming the absence of failures). Our approach ultimately relies on the model-checking tool Tina, that provides state-space generation and model-checking algorithms for an extension of Time Petri Nets with data and priorities.

1 Introduction

We describe our experience with modeling the landing gear system of an aircraft using the formal specification language Fiacre [1]. This case study has been submitted as a problem to be solved by the participants of the Case Study Track at the 4th International ABZ Conference. Our answer to this challenge is based on the use of a model-checking tool for an extension of Time Petri Nets with data and priorities. All the requirements were checked using a dense (continuous) time model, without resorting to discrete time verification methods. The Fiacre models used in this study are available online at `http://projects.laas.fr/fiacre/examples/landinggear.html`.

The purpose of the control system is to manage and monitor the hydraulic and mechanical parts operating the movement of the gears—and their associated doors—on a modern aircraft: activation of the electrical and hydraulic power; opening of the locks and doors; extension or retraction of the gears; ... A full description of the system is given in [2].

The control (digital) part of the system is fairly complex, since there are several subsystems involved—each associated with their own set of timing constraints—and many safety requirement to be satisfied. Some of these requirements are quite straightforward, like for instance that "gears should not be extended if the doors are closed", but other requirements depend on the architecture of the system. For instance that "the controller should not attempt to power the doors without first stimulating the general electro-valves" or that

* This work was partly supported by the ITEA2 Project OpenETCS

"stimulation of the electro-valves should be separated by at least 200ms". Another source of complexity stems from the multiple redundancies put in place as a contingency in case of mechanical failure. Actually, one of the main tasks of the control system is to identify the occurence of failures in order to warn the pilot of any anomalous behavior. This is a major safety requirement, since the pilot should be warned as soon as possible that he needs to engage his emergency extension system.

Our formal model takes into account the behavior and timing properties of the mechanical and control parts of the system, both in its normal and failure mode of operation. We study several versions of the model, each of growing complexity, by strengthening our assumptions on the system. The different versions are used to check safety and real-time properties on the system but also to find a safe bound on the maximal time needed for all gears to be down and locked (assuming the absence of failures). Therefore we experiment here with another interesting application of model-checking, that is as a tool for architecture exploration (dimensioning).

This case study is interesting for several reasons. First, it is well-suited for component-based modeling languages (since the description is highly modular) and it is a good example for real-time verification methods (since the specification has plenty of timing constraints). Also, a similar case study was used by Boniol et al. in 2006 [3], where they compared the use of several model-checking tools: a majority of tools based on the synchronous language Lustre, and one tool, Uppaal, based on timed automata. It is interesting to revisit these results that are nearly ten years old.

2 Fiacre and Tina

We describe the language and tools used to check the behavior of the system. Our approach is based on Fiacre (`http://www.laas.fr/fiacre/`), a specification language designed to represent compositionally both the behavioral and timing aspects of embedded and distributed systems for the purposes of formal verification or simulation. The language comes equipped with a set of dedicated tools, such as compilers from Fiacre to the input formats of model-checking toolboxes, like Tina [4].

2.1 The Fiacre Language

Fiacre is a modeling language for behavioral verification, with a textual notation, in the vein of Promela or BIP. It can be used for model-checking but is not tied to any particular toolset. The language supports two of the most common coordination paradigms: communication through shared variable (shared-memory) and synchronization through synchronous communication ports (message-passing). A formal definition of the language is given in [5].

Fiacre programs are stratified in two main notions: *processes*, which are well-suited for modeling structured activities, and *components*, which describe a system as a composition of processes, possibly in a hierarchical manner.

We give a simple example of a Fiacre specification in Fig. 1. It is the model of a computer mouse driver with double-click. A more complex example of Fiacre process, extracted from the case study, is given in Fig. 4. The behavior of the computer mouse is to emit the event `double` if it receives more than two `click` events in strictly less than one unit of time (u.t.). Note that the semantics of the language is based on a dense, "dimensionless", notion of time. This approach is consistent with several of the state space abstraction techniques used in our tools [6]. Indeed, the "geometric methods" based on the use of Difference Bound Matrices (DBM) are insensitive to the scaling of time (this is not true for methods based on a discrete time approach that may also be used in Tina).

```
process Push [click  : none,
              single : none,
              double : none,
              delay  : none] is

 states s0, s1, s2

 var dbl : bool := false

 from s0 click; to s1

 from s1
  select
     click; dbl := true; loop
  [] delay; to s2
  end

 from s2
  if dbl then double
  else single end;
  dbl := false; to s0
```

```
component Mouse [click : none,
                 once  : none,
                 twice : none] is

 port delay : none in [1,1]

 priority delay > click

 par
    Push [click, once, twice, delay]
 end

//------------------------------------

component Main is

 port click, once, twice, thrice : none

 par
    once → Mouse [click, once, twice]
 || once → Mouse [once, twice, thrice]
 end
```

Fig. 1. A double-click example in Fiacre

Processes: a process is defined by a set of parameters and control states, each associated with a set of *complex transitions* (introduced by the keyword `from`). The initial state of a process is the state corresponding to the first `from` declaration. Complex transitions are expressions that declare how variables are updated and which transitions may fire.

Expressions are built from deterministic constructs available in classical programming languages (assignments, conditionals, sequential composition, ...); non-deterministic constructs (such as external choice, with the `select` operator); communication on ports; and jump to a state (with the `to` or `loop` operators). For example, in Fig. 1, we declare a process named `Push` with four communication ports (`click` to `delay`) and one local boolean variable, `dbl`. Ports may send and receive typed data. The port type `none` means that no data is exchanged; ports of type `none` simply act as synchronization events. Regarding complex transitions, the expression for `s1`, for instance, declares two possible behaviors when in state `s1`: first, on a `click` event, set `dbl` to true and stay in state `s1`; second, on a `delay` event, change to state `s2`.

Data variables are not restricted to simple boolean values. The language provides rich datatypes, such as natural numbers, arrays, queues, records, ... For instance, in the model of the landing gear system (see Sect. 3), we use records and arrays of booleans to represent the signals from the replicated sensor probes. The language is strongly typed, meaning that type annotations are exploited in order to guarantee the absence of unchecked run-time errors.

Components: a component is built from the parallel composition of processes and/or other components, expressed with the operator `par` $\mathtt{P}_0$ `||` ... `||` $\mathtt{P}_n$ `end`. Components are the unit for process instantiation and for declaring ports and shared variables. The syntax of components allows to associate timing constraints with communications and to define priority between communication events. The ability to express directly timing constraints in programs is a distinguishing feature of Fiacre. For example, in the declaration of component `Mouse` (see Fig. 1), the `port` statement declares a local event `delay` with a punctual timing constraint $[1, 1]$. As a consequence, a transition from state `s1` to `s2` in the mouse cannot be delayed more than one unit of time. A behavior similar to the synchronization on a local, time-constrained port like `delay` (basically a time-out) can be obtained using the expression `wait` $[1, 1]$. Additionally, the `priority` statement asserts that a transition on event `click` cannot occur if a transition on `delay` is also possible.

2.2 Behavioral Verification with Tina

Tina [4], the TIme Petri Net Analyzer, provides a software environment to edit and analyze Time Petri Nets and their extensions. It is particularly well suited to the verification of systems subject to real time constraints, such as the landing gear system studied in this paper. The core of the Tina toolset is an exploration engine used to generate state space abstractions that are fed to dedicated model checking and transition system analyzer tools. The front-ends to the exploration engine convert models into an internal representation — the abstract Time Transition Systems (TTS) — that is an extension of Time Petri Nets (TPN) handling data and priorities [7]. We can use the *frac* compiler to convert Fiacre description into TTS and therefore to model-check Fiacre specifications.

We give the graphical representation of a TTS in Fig. 2. This example corresponds to the interpretation of the Fiacre process `Push` from the computer mouse example of Sect. 2.1. A TTS can be viewed as a Time Petri Net where transitions are decorated with guards and actions on data variables; the `pre` and `act` expressions inside dotted rectangles. Data is managed within the `act` and `pre` expressions and refer to a fixed set of variables that form the *store* of the TTS. In comparison with a TPN, a transition in a TTS is enabled if there is both: (1) enough tokens in the places of its pre-condition; and (2) the predicate `pre` is true. When a transition fires, the store is updated atomically by executing the corresponding action `act`. For example, when the token reaches the place s_2 in the TTS of Fig. 2, we use the value of `dbl` to test whether we should signal a double click or not. We can also see in this example the use of read arcs and priorities between transitions (dashed arrow between transitions).

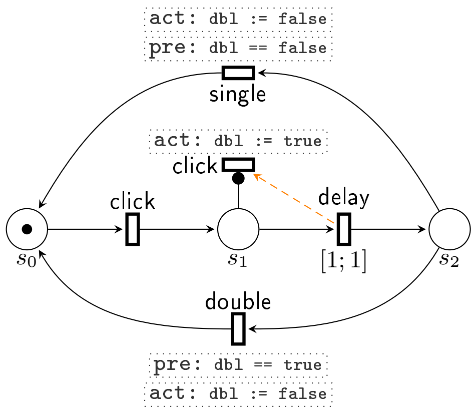

Fig. 2. Interpretation of the process `Push` in TTS

Time Transition Systems is the low level formalism used for model-checking. State space abstractions are vital when dealing with timed systems, such as TTS, that have in general infinite state spaces (because we work with a dense time model). Tina offers several abstract state space constructions that preserve specific classes of properties like absence of deadlocks, reachability of markings, linear time temporal properties, or bisimilarity.

In the case of the landing gear, most of the requirements can be reduced to safety properties, that is, checking that some bad state cannot occur. In this case, we do not need to generate the whole state class graph of the system and we can use "more aggressive" abstractions. Tina implements two main state-space abstraction methods, a default method that preserves the set of states and traces of the system, and a method that preserves the states but not the traces. While this abstraction gives an over-approximation of the set of execution traces of the system, it is often much more efficient than the default exploration mode. This second method can be used in Tina with the command line options `-M` or `-E`. The state-space abstraction corresponding to `-M` usually has a better space complexity than `-E`, but the latter is necessary when using models that have priorities between transitions.

For more complex properties, Tina provides several back-ends to convert its output into physical representations readable by external model checkers. In the context of this study, we need to check LTL properties in the case of failure mode requirements. Broadly speaking, we need to check that, after the failure of a mechanical part (the system is in a `fail` state), every event that triggers the part (say `evt`) will eventually lead to the anomaly being detected (the probe `normal_mode` is set to false). Since the system stays in a fail state when it reaches it, this property could be defined as follows in LTL:

```
[]((fail /\ evt) => <>(not normal_mode)) .
```

We can use *selt*, the model-checker distributed with the Tina toolbox, to check this kind of properties on a Fiacre model. It is a model-checker for an enriched version of State/Event-LTL, a linear time temporal logic supporting both state and transition properties. For the properties found false, we can compute a timed counter example and replay it in a TTS simulator.

3 Model of the Landing Gear System

We take benefit from the compositional and hierarchical nature of Fiacre to model the landing gear system. Each component described in the informal specification [2] is encoded using a Fiacre component and we use the instantiation mechanism to efficiently model the redundancies and symmetries of the system.

The digital and mechanical parts are all modeled using separate components. Only the pilot interface remains implicit as a set of shared boolean variables that can be triggered by the component modeling the system's environment. We also assume that two separate stimuli from the environment cannot occur in less than 100ms. This value of 100ms is taken from the timing constraints information provided by the landing gear specification document (Sect. 4.2 of [2]), namely that "two contrary orders (closure / opening doors, extension / retraction gears) must be separated by at least 100ms". The document does not specify any timing constraints on the movement of the handle or, equivalently, on the reactivity of the pilot/environment. We chose to apply the same constraints of 100ms here to avoid unrealistic scenarios in which the handle could be moved infinitely often in a finite amount of time.

The whole model—when taking into account the maximal level of details—amounts to about 1200 lines of Fiacre. Most of it was programmed in the course of one week by a model-checking specialist that was novice with Fiacre. When compiled into a Time Transition System (see Sect. 2.2) we obtain a net with about 100 places and 150 transitions. These numbers give a rough idea of the complexity of the "coordination" aspect of the system. Concerning the functional complexity of the model, we have about 60 variables in the resulting TTS, but many of these variables are correlated (at least in normal mode, because of the redundancies). This is close to the 54 discrete sensor values declared in the specification and the 5 electrical outputs (called *electrical orders* in the specification [2]) emitted by each computing module.

We describe the structure of the Fiacre specification starting from the data types used in the model. The main data types are almost word for word those given in the informal specification of the system.

3.1 Data types

Different parts of the system interact using electrical orders, hydraulic pressure or sensors. Our model represents this information as boolean values. For example we observe the presence or absence of hydraulic pressure but not its transition phase (growing up / going down). However the time needed by this transition phase is always taken in account and adequately modeled in different parts.

Because the sensors are triplicated, we use a record (a structured data type) formed of a boolean value and a natural number counting active sensors. This makes the interpretation of the sensors much easier and allows the representation of sensor failure.

To simplify the model, we also use arrays of sensors for closed/open door sensors, extended/retracted gear sensors and gear shock absorbers sensors. This allows to reduce the number of variables handled by different processes and to reduce the code size of our model without modifying the generated state space.

3.2 Digital part

No timing constraints are given on the speed of the digital part of the system. (Actually, the description of the system is quite heavily oriented toward a synchronous architecture rather than, say, a time-triggered one.) Since the speed of digital signals is incommensurate with the speed of mechanical parts, we have chosen a null response time for every interaction with the digital part. Thus the digital component computes new outputs instantaneously each time a sensor value changes. However, electro-valve order delays are considered (we adopt the same timing constraints than in the use case specification, see Sect. 4.2 of [2]):

- the simulation of the general electro-valve and the maneuvering electro-valves must be separated by 200ms;
- orders to stop the general electro-valve and the maneuvering electro-valves must be separated by 1s;
- two opposite electro-valve orders must be separated by 100ms.

The digital part is modeled using two instances of the same computing module component and an electrical "OR" process making the composition of computing modules orders. To keep the model simple, each computing module is divided in four processes: the computing process responsible for detecting failures and ordering electro-valves; a process handling general electro-valve timing constraints; and two processes handling contrary orders and their timing constraints. This architecture has been faithfully mimicked in our model even if it is redundant in the normal operation mode given the 100ms delay between stimuli and because both computing processes should behave in the same manner. We illustrate the structure of a computing module component in Fiacre in Fig. 3. (The whole model uses two copies of this component.)

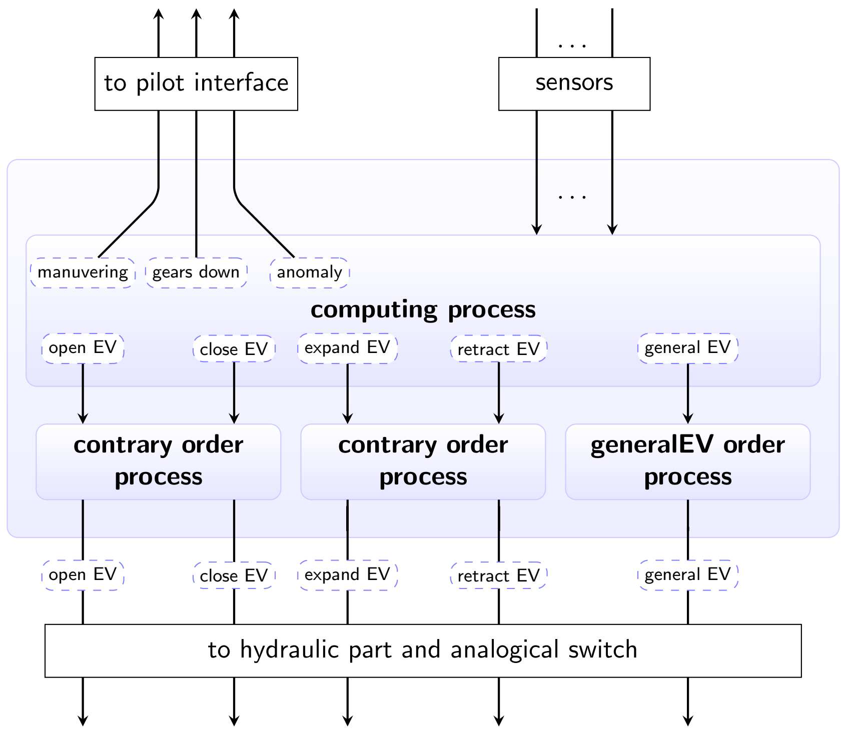

Fig. 3. Computing module implementation.

3.3 Hydraulic Part

The hydraulic part is modeled using a component handling doors and gears circuits. The component is composed of two electro-valves and three cylinders; each part in the hydraulic architecture (valve, cylinder, . . .) is modeled using a Fiacre process. The timing constraints used in the Fiacre processes are the one given by the specification (see e.g. Sect. 3.2 of [3]). For instance, an electro-valve changes its state from open to close in 1 second and from close to open in 3,6 seconds. The process for the cylinders is parametric and configured based on specification times. As for electro-valves, each cylinder motion can be reversed at any time. We consider the whole extension or retraction time in each case and take into account the 20% time variation mentioned in the specification (Sect. 3.3 of [2]).

The main simplification with regards to the specification is that we do not discretize the behavior of the valve and always consider the worst possible execution time. In the *experimental results* section of this paper (see Sect. 4), we also give some results on a “discrete” version of the model where we record the progress of the cylinders between a closing and opening requests and follow the physical behavior defined by the specification (see e.g. Fig. 8 of [2]). The discrete model use a sampling time of 100ms between every event. The size of the state space for this discrete model is quite big when compared to our abstracted model.

We used model-checking to compute the worst-case gear retraction time in our system. This time obtained with the discrete version of the model is the same than with our abstract version; actually we obtain a value that is marginally higher with the discrete model due to an accumulation of errors originating from the "time quantum".

3.4 Analogical Switch

The analogical switch is responsible for interfacing digital orders with the general electro-valve and protecting it from erratic orders. It is enabled each time the handle is moved. We model the closing and the opening of the switch by waiting a certain fixed amount of time (taken from the specification), that is, we do not discretize the state of the switch and always use the worst-case time when changing state.

We list the Fiacre process corresponding to the analogical switch process in Fig. 4. The process `AnalogicalSwitch` is parametrized with variables shared between processes which are used to update sensor states or pass electrical orders. We consider that these operations are immediate and thus are seen as shared boolean variables in our model. The values of these probes are used as guards on the transitions of the process (using the operator `on`). The different states of the switch are *open*, *closing*, *closed*, *opening*, and can be directly mapped to states given in the informal specification of the system. As can be seen on the transition from the closing to the closed state (line 14 of the code), the switch has to wait at most $800ms$ for changing state, as stated by the expression `wait` $[0, 800]$.

3.5 Handling Failures

The physical parts in the system have multiple ways to fail. In our model, we only consider cylinder failures by allowing gear and door cylinders to get stuck in their current position indefinitely. We also assume that a part cannot leave a failure state once it has entered it (no transient failure). We consider only one possible type of failure at a time since adding all the possible cases—and all their combinations—could lead to an intractable model.

To address failure mode requirements, we have added failure handling mechanisms in each computing module, allowing to detect failures and to notify the pilot. In the current model, the only notification mechanism is to set the shared variable `normal_mode` to false. This is done by watching sensor states with adequate timeouts. We focused on failures induced by the requirements $R_{7\star}$ which are stronger than $R_{6\star}$, however requirements $R_{6\star}$ could be easily implemented (we use the notation $R_{6\star}$ to stand for requirements R_{61} to R_{64}). So, requirements $R_{6\star}$ and $R_{8\star}$ were not addressed but could be added with no effort. We made this choice to limit state space sizes.

```
process AnalogicalSwitch(&handle : sensor,
                         &general_EV : electrical_order,
                         &out_EV : electrical_order,
                         &analogical_switch : sensor) is
   states open, intermediate1, closed, intermediate2
   var last_handle : bool := HANDLE_DOWN

   from open
      wait [0,0];
      on (handle.value <> last_handle); // handle state has changed
      last_handle := handle.value;
      to intermediate1 // move to state intermediate1

   from intermediate1
      wait [0,800];  // wait 800ms... then deliver power
      out_EV := general_EV;
      analogical_switch.value := SWITCH_CLOSED;
      to closed // move to state closed

   from closed
      select
         wait [20000,20000]; // wait 20s but only if...
         on (handle.value = last_handle); // handle did not move
         // then cut power and start intermediate2
         analogical_switch.value := SWITCH_OPEN;
         out_EV := false;
         to intermediate2
      [] wait [0,0];
         on (handle.value = last_handle); // if handle did not move...
         on (out_EV <> general_EV); // but the input value had changed
         out_EV := general_EV; // update the output
         loop // stay in this state without but do not reset
      [] wait [0,0];
         on (handle.value <> last_handle);
         // otherwise if handle state has changed...
         // reset immediately this state
         last_handle := handle.value;
         out_EV := general_EV;
         to closed
      end

   from intermediate2
      select
         wait [0,1200]; // wait 1.2s if handle did not move
         on (handle.value = last_handle);
         analogical_switch.value := SWITCH_OPEN;
         to open // move to state open

      [] wait [0,0]; // otherwise if handle did move
         on (handle.value <> last_handle);
         last_handle := handle.value;
         to intermediate1 // move to state intermediate1
      end
```

Fig. 4. The AnalogicalSwitch Process in Fiacre (see Fig. 8 of [2]). Full model available at `http://projects.laas.fr/fiacre/examples/landinggear.html`.

3.6 Optimizations

Because model checking is highly sensitive to state space explosion, our model embeds a certain number of optimizations. The electrical orders, hydraulic pressure, and sensors are abstracted to boolean values, so we can control the number of operations involved when a value changes. For example, we will trigger a component from the digital part of the system (a computing module) only when the change in its input probes leads to a change in the values that it writes. This is useful because it helps reduce the number of transitions in our system. Also, one can remark that computing modules are fully symmetric. Therefore, in normal mode, we will always observe the same values twice; once for each copy of the module. To avoid this unnecessary source of interleaving, we have added priorities between copies of the same component.

Priorities have also been added between the components of the hydraulic system so as to fix an arbitrary order between operations of the electro-valves and cylinders. This optimization is correct because all these devices are independent; hence we limit the interleaving between independent actions but do not rule out any possible scenario.

4 Experimental Results

We follow a methodology similar to the one adopted by Boniol et al. in a previous experiment with model-checking of a landing gear control system [3]. We define several versions of our model that corresponds to different abstractions or optimizations on the system. We define three sets of assumptions and, by combining these parameters, consider different cases of growing complexity.

Parameter V We consider two configurations for the gear-door sets, a version with only one gear-door set (denoted V_1) and a complete version, with all three gear-door sets (V_3).

Parameter H We consider several versions for the environment that stimulates the pilot handle. The most general case where the only constraint on handle movements is a $100ms$ delay between two stimuli is denoted H_2. We also consider simpler scenarios where the pilot can move the handle at most k times. This assumption is denoted $H_1(k)$.

Parameter N/F We use the notation N for models that are restricted to the normal mode, where no failures can happen, and the notation F for models that include failures.

With these parameters defined, it is possible to refer to a version of the model with a triplet, for instance $(V_1, H_1(2), N)$. This is the simplest possible, meaningful case: only one gear-door set; two actions on the handle; and no failures. The most complex case is (V_3, H_2, F).

Because of the complexity of the system, we considered only cylinder failures. Since we only consider cylinder failures, we do not duplicate the computing component in the digital part, however we provide a version of our model allowing this duplication. For checking behavioral properties, we assume that, in the initial state of the system, gears are extended and doors are opened. We also assume that gear absorbers are always relaxed, i.e. we assume that the plane is flying.

4.1 Normal mode requirements

The properties corresponding to normal mode requirements (see [2]) can be expressed as reachability properties. Indeed checking requirements $R_{2\star}$, $R_{3\star}$, $R_{4\star}$, $R_{5\star}$ corresponds to looking for a state were some condition is not satisfied, and requirement $R_{1\star}$ can be expressed with an observer of the system (waiting $15s$) and a reachability condition. This allows for efficient verification using the faster state-space abstraction of Tina (option `-E`) that preserves reachable states without building the whole class graph.

All these properties are expected to be true on our model. This is the worst possible case when checking reachability since it means that we need to generate the whole set of reachable states of the system. We give below the computation times and the memory usage for generating the whole state graph. We also give the complexity using the number of "markings" and "classes" that have been generated in each case. A marking corresponds to a particular value for every variable and state for each process in the system. A class adds timing constraints on the possible transitions enabled from a marking (hence there are always more classes than markings.) Markings are enough to decide the requirements $R_{1\star}$ to $R_{5\star}$, but we need to compute a set of classes in order to compute an exact set of reachable markings.

Normal mode state space computation times and memory usage							
		H_2	$H_1(10)$	$H_1(11)$	$H_1(12)$	$H_1(13)$	$H_1(14)$
V_1	time	$41s$	$56s$	$71s$	$88s$	$105s$	$123s$
	memory	$24MB$	$47MB$	$54MB$	$62MB$	$69MB$	$76MB$
V_3	time	$262s$	$248s$	$331s$	$415s$	$507s$	$602s$
	memory	$127MB$	$202MB$	$241MB$	$282MB$	$323MB$	$364MB$

Normal mode markings and classes sizes							
		H_2	$H_1(10)$	$H_1(11)$	$H_1(12)$	$H_1(13)$	$H_1(14)$
V_1	markings	$16 \cdot 10^3$	$56 \cdot 10^3$	$63 \cdot 10^3$	$71 \cdot 10^3$	$79 \cdot 10^3$	$86 \cdot 10^3$
	classes	$153 \cdot 10^3$	$252 \cdot 10^3$	$303 \cdot 10^3$	$356 \cdot 10^3$	$411 \cdot 10^3$	$468 \cdot 10^3$
V_3	markings	$90 \cdot 10^3$	$242 \cdot 10^3$	$283 \cdot 10^3$	$325 \cdot 10^3$	$367 \cdot 10^3$	$409 \cdot 10^3$
	classes	$979 \cdot 10^3$	$1\,125 \cdot 10^3$	$1\,409 \cdot 10^3$	$1\,701 \cdot 10^3$	$2\,015 \cdot 10^3$	$2\,333 \cdot 10^3$

We can observe that the infinite behavior scenario (H_2) is easier to handle than bounded ones when the bound is at least 10 handle moves for V_1 and 11 handle moves for V_3. This is mainly due to the fact that bounding the number of interactions is performed by implementing a counter that may increase the number of reachable states.

For our next experiment, we study the requirement R_{11} and try to find the smallest time, say t_{min}, for the gears to be fully extended and locked in open position. This property can be reduced to a simple reachability property since there is a specific state, s_i, in the process modeling the pilot behavior that is reached when the pilot stay idle for a time t_{min}. Indeed, it is enough to check that there are no states where the pilot is in s_i and the gears are not fully open. The following table gives the computation time and memory usage for different value of t_{min}, for the configuration (N, H_2, V_3) (no failures, no assumptions on pilot behavior, and the complete gear-door sets). The best time for which the property is true is 8.5s. We can observe that the computation time is much smaller for values below this threshold since the property is false in this case (and the state space exploration can be stopped). So, the computation is quasi-immediate when the time bound is below the 8.5s threshold but the whole state space needs to be computed above it.

Checking requirement R_{11} on (N, H_2, V_3) for different time limits t_{min}				
t_{min}	$15s$	$9s$	$8.5s$	$8.4s$
result	valid	valid	valid	falsified
time	$268s$	$268s$	$268s$	$2s$
memory	$127MB$	$127MB$	$127MB$	$5MB$

We also considered a discretized version of our model where all intermediate movement states were computed, for example the cylinder extension ratio, and where we used the exact (hybrid) physical behavior given in the specification. This discretization was made using a sampling time of 100ms. Because of the number or possible combinations of cylinders, analogical switch and electro-valves, the number of states grow much faster than with our abstract (non discrete) version. Actually the discretized version was our first attempt, because we initially believed that it was giving more precise bounds. However, the 100ms sampling time was not enough to provide better results than the non discrete version. With the discrete version, the configurations $(N, H_1(5), V_1)$ and above were not computable in reasonable times (less than 8 hours).

Normal mode state space computation times and memory usage (discrete)				
		$H_1(2)$	$H_1(3)$	$H_1(4)$
V_1	time	$17s$	$804s$	$19\,887s$
	memory	$33MB$	$1\,132MB$	$8\,982MB$

Normal mode markings and classes sizes (discrete)				
		$H_1(2)$	$H_1(3)$	$H_1(4)$
V_1	markings	$158 \cdot 10^3$	$5\,097 \cdot 10^3$	$112\,094 \cdot 10^3$
	classes	$217 \cdot 10^3$	$8\,648 \cdot 10^3$	$202\,266 \cdot 10^3$

These experiments show the interest of having different kind of abstractions implemented in the same tool (like having different symbolic methods available). The most complex configuration we tried to analyze with the default options of Tina (that preserves linear time properties) is (N, H_2, V_3). We stopped the analysis after 36 hours of computation and more than 2 billion state classes. The same model can be analyzed with the time-abstracted semantics (option `-M`) in two hours (7355s), then with the same option and after removing duplication of the digital component in 422s. Our results also show the interest of priorities to reduce the state space size. For instance, after adding priorities between independent devices and removing duplication of the digital component, we can analyze the same system in 262s (option `-E`). To see the impact of different optimizations we considered a smaller case $(N, H_1(8), V_3)$ with different configurations and all without computing module duplication, the results are shown in the table below.

Impact of optimizations on markings and classes.			
$(N, H_1(8), V_3)$	`-E`	priorities only	no priorities
time	$119s$	$5\,237s$	$12\,383s$
memory	$126MB$	$2\,204MB$	$5\,467MB$
markings	$160 \cdot 10^3$	$160 \cdot 10^3$	$292 \cdot 10^3$
classes	$619 \cdot 10^3$	$54\,342 \cdot 10^3$	$108\,302 \cdot 10^3$

4.2 Failure mode requirements

As mentioned in section 3.5, we focused on requirements $R_{7\star}$: “If one of the three doors is not seen locked in the open position more than 7 seconds after stimulating the opening electro-valve, then the boolean output normal mode is set to false”. To check that we satisfy these requirements we need to consider LTL formula.

We can express the requirement R_{71} quite naturally using LTL: after a failure (`fail_c1`), if at least one door is closed (`not open_d1`) and we later try to stimulate the opening electro-valve (`<>open_EV`) then the boolean *normal_mode* is eventually set to false.

```
[]((fail_c1 /\ (not open_d1) /\ (<>open_EV)) => <>(not normal_mode)).
```

We can observe that the 7 seconds delay does not appear explicitly in the formula. Indeed, this delay is part of the behavior of the digital module. This formula is false when checked on the model. After looking at the counter-example provided by the model-checker, we find that the problematic scenario corresponds to a situation where the pilot continuously moves the handle, waiting less than 7 seconds between each movement. We can modify the property in order to rule out this scenario; i.e. ask that the pilot does not move the handle up. We solve this issue by adding an *idle* state to our pilot that can be reached after moving the handle. If this idle state is reached then the pilot will not move the handle again. With this new state added, the correct formula is

```
[]((pilot_idle /\ handle_down /\ fail_c1 /\ (not open_d1)
                         /\ (<>open_EV)) => <>(not normal_mode)).
```

We were not able to model-check the system with the configuration H_2. Even if the number of reachable states remains quite small in this case, the number of classes is too large to address it in reasonable time. We give below the results obtained with a “bounded” pilot ($H_1(k)$) and an incomplete or full gear-door set (configurations with V_1 or V_3).

Failure mode, time and memory usage results for bounded scenarios					
		$H_1(3)$	$H_1(4)$	$H_1(5)$	$H_1(6)$
V_1	time	$2s$	$7s$	$15s$	$32s$
	memory	$7MB$	$17MB$	$34MB$	$54MB$
V_3	time	$70s$	$304s$	$968s$	$2418s$
	memory	$169MB$	$611MB$	$1\,544MB$	$2\,925MB$

Failure mode, markings and classes counts for bounded scenarios					
		$H_1(3)$	$H_1(4)$	$H_1(5)$	$H_1(6)$
V_1	markings	$12 \cdot 10^3$	$30 \cdot 10^3$	$54 \cdot 10^3$	$83 \cdot 10^3$
	classes	$17 \cdot 10^3$	$49 \cdot 10^3$	$108 \cdot 10^3$	$200 \cdot 10^3$
V_3	markings	$317 \cdot 10^3$	$1\,153 \cdot 10^3$	$2\,822 \cdot 10^3$	$5\,073 \cdot 10^3$
	classes	$458 \cdot 10^3$	$1\,725 \cdot 10^3$	$4\,967 \cdot 10^3$	$10\,847 \cdot 10^3$

4.3 Comparaison with a previous, similar study

A similar case study was used by Boniol et al. in 2006 [3], where they compared the use of several model-checking tools: a majority of tools based on the synchronous language Lustre, and one tool, Uppaal, based on timed automata. It is interesting to revisit these results that are nearly ten years old. This comparison is not very significant though. Indeed, even if the specification used in our work derives from the use case of [3], it is not clear if they are totally equivalent. Also, we do not know what optimizations were used in the other models. In particular, our use of an abstract (non discrete) behavior for the analog switches and the cylinders may account for most of our good results.

In the study of [3], no tools were able to deal with the failure mode requirements. For the nominal case, the most complex problem configuration studied is equivalent to (N, H_2, V_3) with our notation. With this configuration, Lustre-SMV requires $414MB$ of memory and 16mn 40s to compute its result. We give the running time for information only, since it is not meaningful to compare computers that are 10 years apart. On the opposite, the memory consumption offers a more reliable point of comparison. Using Tina on our model for (N, H_2, V_3) we need only $127MB$ and 5mn.

Uppaal, that is based on timed automata, provides the formalism that is the closest to Time Petri Nets from all the tools considered in this study. At the time, Uppaal gave no results on the configuration with three gears. On a configuration with only one gear, equivalent to (N, H_2, V_1) with our notation, it takes $761MB$ and nearly 6 hours to return a result. Unfortunately we do not have access to the model and do not know the number of states that were generated, so we cannot use this information as a basis for our comparison. This can be compared to the $24MB$ of memory that are needed in our experiment (and 41s, with the same caveat than previously).

5 Conclusion

We have illustrated the use of Fiacre for checking the real-time properties of a fairly large and complex real-life case study. We have provided a formal model that is as faithful as possible to the informal, reference specification, at the risk of obtaining intractable model-checking problems. This model could be further optimized in order to obtain better computation times when checking a specific set of properties, for example by reducing the inherent level of redundancies when it does not modify the behavior of the system. Nonetheless, even without further optimizations, it is possible to check most of the requirements that are part of the specification.

Other solutions for checking larger, more complex configurations of our model are worth pursuing. A first possibility will be to take benefit from the symmetries of the system (for instance, the two rear gears are interchangeable). Another solution will be to simplify the transient transitions of the model, that is the internal, instantaneous transitions that are only used for modeling purpose and have no "physical meaning" in the system. This simplification can be compared to what we already obtain by adding priorities between independent devices, but would be more efficient and simpler to define at the model level. Unfortunately, our toolset does not provide this optimization. A first investigation (by reducing the state class graph afterward) show that, this way, we could reduce the memory usage by a factor of about 20.

References

1. Berthomieu, B., Bodeveix, J.P., Farail, P., Filali, M., Garavel, H., Gauffilet, P., Lang, F., Vernadat, F.: Fiacre: an intermediate language for model verification in the topcased environment. In: Embedded Real Time Software (ERTS). (2008)
2. Wiels, V., Boniol, F.: Landing gear system. Case study for the ABZ 2014 conference (2014)
3. Wiels, V., Boniol, F., Ledinot, E.: Experiences in using model checking to verify real time properties of a landing gear control system. SIA/Articles techniques (2006)
4. Berthomieu, B., Ribet, P.O., Vernadat, F.: The tool Tina – construction of abstract state spaces for Petri Nets and time petri nets. International Journal of Production Research **42** (2004)
5. Berthomieu, B., Bodeveix, J.P., Filali, M., Garavel, H., Lang, F., Peres, F., Saad, R., Stoecker, J., Vernadat, F.: The syntax and semantics of fiacre. Repport LAAS N 07264 (2007)
6. Berthomieu, B., Vernadat, F.: State Space Abstractions for Time Petri Nets. Handbook of Real-Time and Embedded Systems, Ed. Insup Lee, Joseph Y-T. Leung and Sang Son, CRC Press, Boca Raton, FL., U.S.A. (2007)
7. Abid, N., Dal Zilio, S., Le Botlan, D.: A formal framework to specify and verify real–time properties on critical systems. International Journal of Critical Computer-Based Systems **5** (2014) 4–30