A Model-Checking Approach to Analyse Temporal Failure Propagation with AltaRica

Alexandre Albore, Silvano Dal Zilio, Guillaume Infantes, Christel Seguin, Pierre Virelizier

A Model-Checking Approach to Analyse Temporal Failure Propagation with AltaRica

Alexandre Albore[1,2,3], Silvano Dal Zilio[2], Guillaume Infantes[3], Christel Seguin[3], and Pierre Virelizier[1]

[1] Institute of Research and Technology (IRT) Saint Exupéry, Toulouse, France
[2] LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France
[3] ONERA, 2 Avenue Edouard Belin, 31055 Toulouse, France

Abstract. The design of complex safety critical systems raises new technical challenges for the industry. As systems become more complex—and include more and more interacting functions—it becomes harder to evaluate the safety implications of local failures and their possible propagation through a whole system. That is all the more true when we add time to the problem, that is when we consider the impact of computation times and delays on the propagation of failures.
We describe an approach that extends models developed for Safety Analysis with timing information and provide tools to reason on the correctness of temporal safety conditions. Our approach is based on an extension of the AltaRica language where we can associate timing constraints with events and relies on a translation into a realtime model-checking toolset. We illustrate our method with an example that is representative of safety architectures found in critical systems.

1 Introduction

The increasing complexity of interactions between functions in modern industrial systems poses new technical challenges. In fact, developing complex systems often raise integration problems during the product final testing and verification phase. Besides, correcting these issues often generates a heavy rework and is a well-known cause for cost overruns and project delays. Therefore, finding solutions that contribute to anticipate and resolve integration problems as early as possible in the design process has become a prior concern for the industry.

New modelling techniques, such as MBSE or MBSA, propose to master the combinatorial complexity at early concept phases by using abstract high level representations of a system. These views constitute a promising ground to implement early validation techniques of the architectures. But, in order to be profitable and implemented by the industry, those validation techniques must remain lightweight and well integrated in the system design process. That is to say, the modelling workload must be limited, and the analysis results (even preliminary) must be available at the same time than designers evaluate the possible architecture choices.

In this paper, we describe an approach that allows us to extend models developed for safety analysis in order to reason about the correctness of temporal conditions. We intend to offer the capability to study a new range of system requirements that can be of main interest for functions such as failure detection, isolation and recovery. We advocate that timing properties are critical when assessing the safety of embedded and real-time systems. Indeed, temporal aspects—like network delays or computation times—can be the cause of missed failure detections or undesired reactions to (delayed) failure propagation. It is therefore necessary to be able to analyse the temporal properties of a model in order to build systems that will operate as intended in a real-world environment.

We define a model-based process to check simultaneously safety and temporal conditions on systems. Our approach is based on an extension of the AltaRica language [1] where timing constraints can be associated with events. This extension can then be translated into the intermediate language Fiacre [7], a formal specification language that can be used to represent both the behavioural and timing aspects of systems. This Fiacre model can be analysed with the realtime model-checker Tina [6]. The results of model-checking shed light on the dysfunctional behaviour of the original model, including how the cascading effects due to failure propagation delay reveal transitory failure modes.

Our contribution is as follows. We define a lightweight extension of AltaRica, meaning that timing constraints are declared separately from the behaviour of a system. Therefore it is easy to reuse a prior safety model and to define its temporal behaviour afterwards. We illustrate our method with an example inspired by safety architectures found in avionic systems. This example illustrate the impact of time when reasoning about failure propagation. We use this example to show that taking into accounts timing constraints—in particular propagation delays—can help finding new failure modes that cannot be detected in the untimed model currently in use. In the process, we define two safety properties: *loss detection*; and its temporal version, *loss detection convergence*, meaning that a system applies an appropriate and timely response to the occurrence of a fault before the failure is propagated and produces unwanted system behaviours. We show that these two properties, which are of interest in a much broader context, can be reduced to effective model-checking problems.

The paper is organised as follows. We start by defining the AltaRica language and the time model in Sect. 2. In Sect. 3, we introduce the Fiacre language by taking as example the encoding of an AltaRica node. This example gives an overview of how to encode AltaRica in Fiacre. We discuss the problem associated with time failure propagation in Sect. 4. Finally, before concluding with a discussion on related works, we give some experimental results in Sect. 5.

2 Model-Based Safety Analysis with AltaRica

Failure propagation models are defined by safety engineers and are usually obtained through manual assessment of the safety of the system. This is a complicated task since failures can depend on more than one element of the system;

be the result of the interaction between many faults; be the consequence of the missed detection of another fault (e.g. a fault inside an element tasked with detecting faults); etc. To cope with the complexity of the systems and the scenarios that need to be analysed, several model-based approaches have been proposed such as AltaRica, Figaro, etc. each with their associated tooling.

AltaRica is a high level modelling language dedicated to Safety Analysis. It has been defined to ease the modelling and analysis of failure propagation in systems. The goal is to identify the possible failure modes of the system and, for each mode, the chain of events that lead to an unwanted situation.

In AltaRica, a system is expressed in terms of variables constrained by formulas and transitions. Several versions of the language have been defined (see for instance [1,15]). In this work, we use the AltaRica 2.0 Dataflow language which is a fragment of other versions and which is sufficient to analyse the behaviour of computer based systems. (The approach discussed here could be applied on other AltaRica dialects.) The models used in this work have been edited and analysed using Cecilia OCAS, a graphical interactive simulator developed by Dassault Aviation [8].

An AltaRica model is made of interconnected *nodes*. A node can be essentially viewed as a mode automaton [2] extended with guards and actions on data variables. A node comprises three parts: a declaration of variables and events, the definition of transitions, and the definition of assertions. We illustrate these concepts with the example of a simple node called `Function`. We give the code (textual definition) of `Function` in Listing 1.1 and a schematic representation in Fig. 1. The node `Function` has one input, `I`, and one output, `O`.

```
domain FState = {NOMINAL, LOST, ERROR} ;
domain FailureType = {Err, Loss, Ok} ;

node Function
   flow   I : FailureType : in ; O : FailureType : out ;
   state  S : FState ;
   event  fail_loss, fail_err ;
   init   S := NOMINAL ;
   trans  S != LOST    ⊢ fail_loss → S := LOST ;
          S =  NOMINAL ⊢ fail_err  → S := ERROR ;
   assert O = case { S = NOMINAL : I, S = LOST : Loss, else Err } ;
edon
```

Listing 1.1. Example of AltaRica code for the node `Function`.

Nodes can have an internal state stored in a set of *state variables*, declared in a heading called `state`. In its nominal state (when `S = NOMINAL`), the `Function` node acts as a perfect relay: it copies on its output the value provided by its input (we have `O = I`); this is expressed in the `assert` block. On the opposite, its output is set to `Loss` or `Err` when `S` equals `LOST` or `ERROR`, respectively. The assert directive is used to express constraints on the values of the input and output variables of a node, also called *flow variables*, establishing a link between the node and its environment, i.e. the other interconnected nodes. It distinguishes between input (`in`) and output (`out`) variables. An assertion defines a rule to

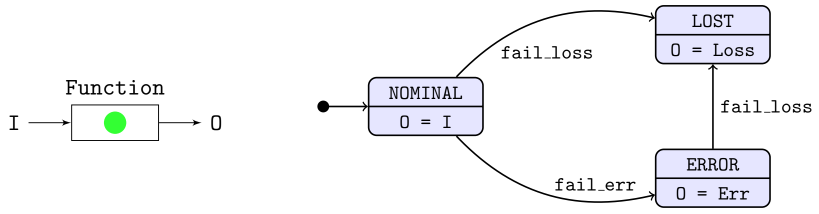

Fig. 1. Graphical representation of node `Function` (left) and its associated failure mode automaton (right).

update the value of output flows according to the state of the component and the value of input flows.

The state variables of a node can only change when an event is triggered. The code of `Function` declares two events: `fail_err`, that changes the state from `NOMINAL` to `ERROR`, and `fail_loss`, that can transition from any state to `LOST`. This behaviour is the one displayed on the mode automaton of Fig. 1. Transitions are listed in the `trans` block of the node. Each transition has an (event) name and a definition of the form `g` $\vdash$`evt`$\rightarrow$ `e`, where the guard `g` is a Boolean condition that can refer to state and flow variables. The event `evt` can be triggered when the guard is satisfied. In this case we apply the effect, `e`, that is an expression that modifies the values of state variables.

Events are useful to model the occurrence of failures or the reaction to conditions on the flow variables. We can assign a law of probability on the occurrence of the failure using the heading `extern`. For instance we could assert that event `fail_loss` follows an exponential distribution with the declaration: `extern law (<event fail_loss>)="exp 1e−4";` In the next section, we propose a way to enrich this syntax to express timing constraints on events instead of probability distributions. At the moment, it is not possible to use stochastic events in addition to time events.

In the general case, an AltaRica model is composed of several interconnected node instances, following a component-based approach. Global assertions relate the input flows of a component to the output flows of other components. For the sake of brevity, we do not describe component synchronisation here and we refer the reader to [16] for further details. More importantly, a hierarchical AltaRica model can always be "flattened", i.e. represented by a single node containing all the variables, events, assertions, and transitions from the composite system. We use this property in our interpretation of AltaRica in Fiacre.

Adding Timing Constraints to Events. There already is a limited mechanism for declaring timing constraints in AltaRica. It relies on the use of external law associated with a *Dirac distribution*. An event with Dirac(0) law denotes an instantaneous transition, that should be triggered with the highest priority. Likewise, an event with Dirac(d) (where d is a positive constant) models a transition that should be triggered with a delay of d units of time. In practice, Dirac laws are rather a way to encode priorities between events than an actual mean

```
node Pre
   flow  I : FailureType : in; O: FailureType : out;
   state Stored, Delayed : FailureType, S : BType;
   event pre_read, pre_wait;
   init  Stored := Ok, Delayed := Ok, S := Empty;
   trans
     (Stored != I) & (S = Empty) ⊢ pre_read → Stored := I, S = Full;
     (S = Full) ⊢ pre_wait → Delayed := Stored, S = Empty;
   assert  O = Delayed;
   extern law (pre_read) = "[0,0]"; law (pre_wait) = "[a,b]";
edon
```

Listing 1.2. Example of Time AltaRica code: the basic delay.

to express duration. Moreover, while Dirac laws are used during simulation, they are not taken into account by the other analysis tools. Finally, the use of Dirac laws is not expressive enough to capture non-deterministic transitions that can occur within time intervals of the form $[a, b]$, where $a \neq b$. These constraints are useful to reason about failure propagation delays with different best and worst case traversal time. For this reason, we propose to extend event properties with *temporal laws* of the form: `extern law (evt) = "[a,b]"`; It is also possible to use open and/or unbounded time intervals, such as `]a,∞[`.

With such a declaration, the transition `g ⊢evt→ e` can be triggered only if the guard `g` is satisfied for a duration (or *waiting time*) δ, with $\delta \in [a, b]$. A main difference with the original semantics of AltaRica is that the timing constraint of an event is not reinitialised unless its guard is set to false. Moreover, our semantics naturally entails a notion of *urgency*, meaning that it is not possible to miss a deadline: when δ equals b, then either `evt` is triggered or another transition should (instantaneously) change the value of the guard `g` to false.

We can illustrate the use of temporal laws with the following example of a new node, `Pre`; see Listing 1.2. This node encodes a buffer that delays the propagation of its input. When the input changes, event `pre_read` has to be triggered instantaneously. Then, only after a duration $\delta \in [a, b]$, the value stored by `Pre` (in the state variable `Stored`) is propagated to its output.

3 A Definition of Fiacre Using Examples

Fiacre [7] is a high-level, formal specification language designed to represent both the behavioural and timing aspects of reactive systems. Fiacre programs are stratified in two main notions: *processes*, which are well-suited for modelling structured activities (like for example simple state machines), and *components*, which describes a system as a composition of processes. In the following, we base our presentation of Fiacre on code examples used in our interpretation of Time AltaRica. We give a simple example of Fiacre specification in Listing 1.3. This code defines a process, `Function`, that simulates the behaviour of the AltaRica node given in Listing 1.1.

Fiacre is a strongly typed language, meaning that type annotations are exploited in order to avoid unchecked run-time errors. Our example defines two

```
type FState      is union NOMINAL | LOST | ERROR end
type FailureType is union Err | Loss | Ok end
type Flows       is record I:FailureType, O:FailureType end

function update(S : FState, env : Flows) : Flows is
   var  f : Flows := {I=env.I, O=env.O}
   begin
      f.O := (S = NOMINAL ? f.I : (S = LOST ? Loss : Err));
      return f
   end

process Function(&S : FState, &env : Flows) is
   states s0
   from s0 select
      on (S != LOST); S := LOST; env := update(S, env); loop
   [] on (S = WORKING); S := ERROR; env := update(S, env); loop
   end
```

Listing 1.3. Example of Fiacre code: type, functions and processes

enumeration types, `FState` and `FailureType`, that are the equivalent of the namesake AltaRica domains. We also define a record type, `Flows`, that models the environment of the node `Function`, that is an association from flow variables to values. Fiacre provides more complex data types, such as arrays, tagged union or FIFO queues. Fiacre also supports native *functions* that provide a simple way to compute on values. In our example, function `update` is used to compute the state of the environment after an event is triggered; that is to model the effect of assertions in AltaRica. It uses two ternary (conditional) operators to mimic the `case`-expression found in the `assert` heading of Listing 1.1.

A Fiacre *process* is defined by a set of parameters and control states, each associated with a set of *complex transitions* (introduced by the keyword `from`). Our example defines a process with two shared variables—symbol `&` denotes variables passed by reference—that can be updated concurrently by other processes. In our case, variable `S` models the (unique) state variable of node `Function`.

Complex transitions are expressions that declares how variables are updated and which transitions may fire. They are built from constructs available in imperative 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). In Listing 1.3, the `select` statement defines two possible transitions, separated by the symbol `[]`, that loop back to `s0`. Each transition maps exactly to one of the AltaRica events, `fail_loss` and `fail_err`, that we want to translate. Transitions are triggered non-deterministically and their effects are atomic (they have an "all or nothing" semantics). A transition can also be guarded by a Boolean condition, using the operator `on` or another conditional construct.

It is possible to associate a time constraint to a transition using the operator `wait`. Actually, the ability to express directly timing constraints in programs is a distinguishing feature of Fiacre. We illustrate this mechanism in the code below, that corresponds to the interpretation of the node `Pre` of Listing 1.2. Basically, a transition constrained by a (time) interval I can be triggered after a time δ, with $\delta \in I$, only if its guard stayed continuously valid during this time. It is this behaviour that inspired our choice of semantics for the temporal law.

A Fiacre *component* defines a parallel composition of components and/or processes using statements of the form `par` $\mathtt{P}_0 \parallel \cdots \parallel \mathtt{P}_n$ `end`. It can also be used to restrict the visibility of variables and ports and to define priorities between communication events. We give an example of Fiacre component in Listing 1.4.

A possible issue with the implementation of `Pre` is that at most one failure mode can be delayed at a time. Indeed, if the input `I` of `Pre` changes while the state is `Full`, then the updated value is not taken into account until after event `pre_wait` triggers. It is not possible to implement a version that can delay an unbounded number of events in a bounded time as it would require an unbounded amount of memory to store the intermediate values. More fundamentally, this would give rise to undecidable verification problems (see e.g. [11]). To fix this issue, we can define a family of operators, `Pre_k`, that can delay up-to k simultaneous different inputs. Our implementation relies on a component that uses three process instances: one instance of `front`, that reads messages from the input (variable `I`), and two instances of `delay`, that act as buffers for the values that need to be delayed. Process `front` uses the local ports `go1` and `go2` to dispatch values to the buffers. Component `Pre_2` is enough to model the use case defined in Sect. 4. Indeed, any element in the system may propagate at most two different status, one from `Ok` to `Err` and then from `Err` to `Loss`.

```
process Pre(&Stored, &Delayed : FailureType, S : BType, &env : Flows) is
   states s0
   from s0 select
      on (Stored != env.I and S = Empty); wait [0,0]; Stored := env.I; ...
   [] on (S = Full); wait [1,2]; Delayed := Stored; S := Empty; ...
   end

process delay[go : in FailureType](&O : FailureType) is
   states sEmpty, sFull
   var delayed : FailureType := Ok
   from sEmpty go?delayed; to sFull
   from sFull  wait [1,2]; O := delayed; to sEmpty

process front[p,q : out FailureType](&I : FailureType) is
   states s
   var stored : FailureType := Ok
   from s on (I != stored); stored := I; select p!I [] q!I end; loop

component Pre_2(&I, &O: FailureType) is
   port go1, go2 : FailureType in [0,0]
   priority go1 > go2
   par * in front[go1,go2](&I) || delay[go1](&O) || delay[go2](&O) end
```

Listing 1.4. An upgraded version of the delay operator `Pre`, with wait statement, components and synchronisation on ports.

4 Example of a Failure Detection and Isolation System

We study the example of a safety critical function that illustrates standard failure propagation problems. We use this example to show the adverse effects of temporal failure propagation even in the presence of Failure Detection and Isolation (FDI) capabilities. This example is inspired by the avionic functions that provide parameters for Primary Flight Display (PFD), which is located in the aircraft cockpit. The system of interest is the computer that acquires sensors measurements and computes the aircraft *calibrated airspeed* (CAS) parameter. Airspeed is crucial for pilots: it is taken into account to adjust aircraft engines thrust and it plays a main role in the prevention of over speed and stall.

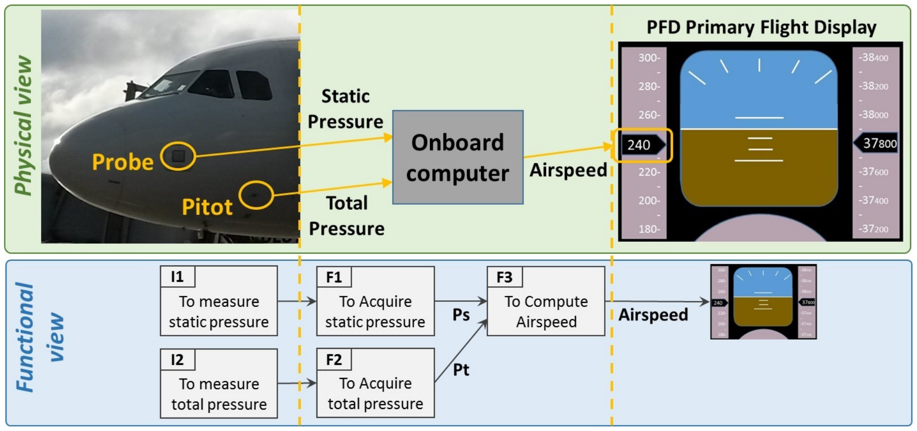

Fig. 2. Functional and physical views of the airspeed computation function.

CAS is not directly measured by a dedicated sensor, but is computed as a function of two auxiliary pressure measurements, the static pressure (Ps) and total pressure (Pt); that is CAS $= f(\text{Pt}, \text{Ps})$. These two measurements come from sensors located on the aircraft nose, a pressure probe and a pitot tube.

Our proposed functional view is given in Fig. 2. It consists in two external input functions `I1` and `I2` that measure static and total pressure; and three inner functions of the system, `F1` and `F2` for sensor measurements acquisition by the on-board computer and `F3` for airspeed computation. For simplification purposes, the PFD functions have not been modelled.

Next, we propose a first failure propagation view aiming at identifying the scenarios leading to an erroneous airspeed computation and display to the pilot (denoted `Err`). Such failure can only be detected if a failure detector is implemented, for instance by comparing the outputs of different functions. Undetected, it could mislead the pilot and, consequently, lead to an inappropriate engine thrust setting. We also want to identify the scenarios leading to the loss of the measure (denoted `Loss`). In such a case, the pilot can easily assess that the measure is missing or false and consequently rely upon another measure to

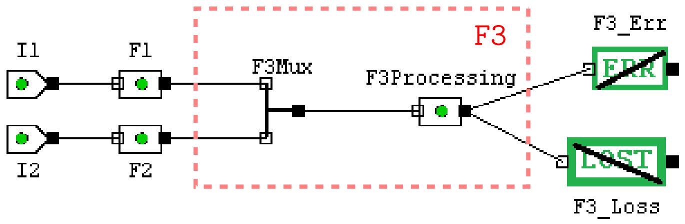

Fig. 3. A simple example of failure propagation.

control the aircraft (note that such redundancy is not modelled). For example, airspeed out of bound—indicating that an airliner has crossed the sonic barrier—is considered to be of kind `Loss`. It can be understood that scenarios leading to the loss of the airspeed are less critical than the ones leading erroneous values.

Safety model of the architecture without FDI. We provide an AltaRica model corresponding to the functional view of the CAS function in Fig. 3. This model, tailored to study failure propagation, is comprised of: two external functions, `I1` and `I2`, that have no input (so, in their nominal state, the output is set to `Ok`); two inner functions, `F1` and `F2`, which are instances of the node `Function` described in Sect. 2; and a function, `F3`, that is the composition of two basic elements: a multiplexer, `F3Mux`, representing the dependence of the output of `F3` from its two inputs, and a computing element `F3Processing` that represents the computation of the airspeed. `F3Processing` is also an instance of node `Function`.

In case of single failure scenario, `F3Mux` propagates the failure coming either from one input or the other. In case of multiple failures, when different failures propagate, one being `Loss` and the other being `Err`,—and without appropriate FDI—the system outcome is uncertain. Solving this uncertainty would require a detailed behavioural model of the on-board computer and a model for all the possible failure modes, which is rarely feasible with a sufficient level of confidence, except for time-tested technology. Given this uncertainty, it is usual to retain the state with the most critical effect, that is to say: the output of `F3` is `Err`.

Our goal is to prevent the computation of an erroneous airspeed while one of `F3` input signals is lost. The rationale is that the system should be able to passivate automatically the airspeed when it detects that one of its input signals is not reliable. This behaviour can be expressed with the following property:

Safety Property 1 (Loss Detection and Instantaneous Propagation). A function is *loss detection safe* if, when in nominal mode, it propagates a `Loss` whenever one of its input nodes propagates a `Loss`.

We can show that our example of Fig. 3 does not meet this property using the *Sequence Generation* tool available in Cecilia OCAS. To this end, we compute the minimal cuts for the target equation $((\texttt{F1.O.Loss} \vee \texttt{F2.O.Loss}) \wedge \neg\texttt{F3.O.Loss})$,

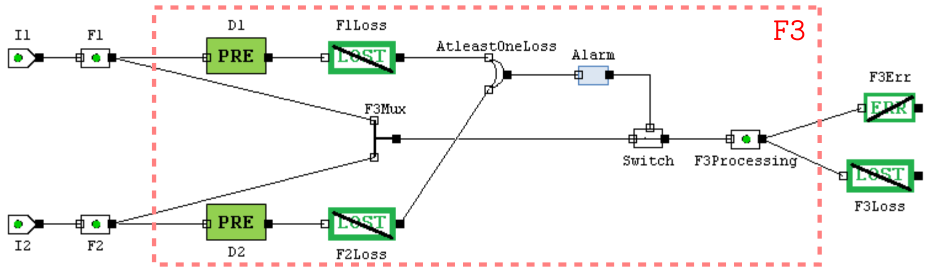

Fig. 4. Model of a FDI function with a switch and an alarm.

meaning the scenario where `F3` does not propagates `Loss` when one of `F1` or `F2` does. Hence function `F3` is loss detection safe if and only if the set is empty.

In our example, once we eliminate the cases where `F3` is not nominal (that is when `F3Processing` is in an error state), we find eight minimal cuts, all of order 2. In the following section, we correct the behaviour of `F3` by considering a new architecture based on detectors and a switch to isolate the output of `F3` when faulty.

Safety model of the architecture with FDI. The updated implementation of `F3` (see Fig. 4) uses two perfect detectors, `F1Loss` and `F2Loss`, that can detect a loss failure event on the inputs of the function. The (Boolean) outputs of these detectors are linked to an OR gate (`AtLeastOneLoss`) which triggers an `Alarm` when at least one of the detectors outputs true. The alarm commands a `Switch`; the output of `Switch` is the same as `F3Mux`, unless `Alarm` is activated, in which case it propagates a `Loss` failure. The alarm can fail in two modes, either continuously signaling a `Loss` or never being activated. The schema in Fig. 4 also includes two delays operators, `D1` and `D2`, that model delay propagation at the input of the detectors; we will not consider them in the following lines, but come back to these timing constraints at the end of the section.

The FDI function—with a switch and an alarm—is a stable scheme for failure propagation: when in nominal mode, it detects all the failures of the system and it is able to disambiguate the case where its inputs contains both `Err` and `Loss`. Once again, this can be confirmed using the Sequence Generation tool. If we repeat the same analysis than before—and if we abstract away the delay nodes—we find 56 minimal cuts, all involving a failure of either `Alarm` or `F3Processing`, i.e. a non-nominal mode. This means that, in an untimed model, our new implementation of `F3` satisfies the loss detection property, as desired. Even so, it is easy to find a timed scenario where the safety property is violated.

Assume now that `F1` and `F2` propagate respectively the status `Loss` and `Err`, at the same date. In such a case and considering possible latencies, while `Err` reaches `F3Mux` instantaneously, the output of `F1` might reach `F1Loss` at successive date. This leads to a transient state where the alarm is not activated whereas the output of `F3Mux` is set to `Err`. This brings us back to the same dreaded scenario than in our initial model.

This example suggests that we need a more powerful method to compute the set of cuts in the presence of temporal constraints. On the other hand, we may also advocate that our safety property is too limiting in this context, where perfect synchronicity of events is rare. Actually, it can be proven that the output of `F3` will eventually converge to a loss detection and isolation mode (assuming that `F3` stays nominal and that its inputs stay stable). To reflect this situation, we propose an improved safety property that takes into account temporal properties of the system:

Safety Property 2 (Loss Detection Convergent). A function is *loss detection convergent* if (when in nominal mode) there exists a duration Δ such that it continuously outputs a `Loss` after the date $\delta_0 + \Delta$ if at least one of its input nodes continuously propagates a `Loss` starting from δ_0 onward. The smallest possible value for Δ is called the *convergence latency* of the function.

Hence, if the latency needed to detect the loss failure can be bound, and if the bound is sufficiently small safety-wise, we can still deem our system as safe. In the example in Fig. 2, this property can indicate for how long an erroneous airspeed is shown on the PFD to the pilot, before the failure is isolated.

In the next section, we use our approach to generate a list of "timed cuts" (as model-checking counterexamples) that would have exposed the aforedescribed problems. We also use model-checking to compute the convergence latency for the node `F3`. In this simple example, we can show that the latency is equal to the maximal propagation delay at the input of the detectors. The value of the latency could be much harder to compute in a more sophisticated scenario, where delays can be chained and/or depends on the internal state of a component.

5 Compilation of AltaRica and Experimental evaluation

We have implemented the transformation outlined in Sect. 3; the result is a compiler that automatically generates Fiacre code from an AltaRica model. The compilation process relies on the fact that it is possible to "flatten" a composition of interconnected nodes into an intermediate representation, called a *Guarded Transition System* (GTS) [3]. A GTS is very similar to a (single) AltaRica node and can therefore be encoded in a similar way. Our tool is built using the codebase of the model-checker EPOCH [17], which provides the functionalities for the syntactic analysis and the linking of AltaRica code. After compilation, the Fiacre code can be checked using Tina [6]. The core of the Tina toolset is an exploration engine that can be exploited by dedicated model-checking and transition analyser tools. Tina offers several abstract state space constructions that preserve specific classes of properties like absence of deadlocks, reachability of markings, or linear and branching time temporal properties. These state space abstractions are vital when dealing with timed systems that generally have an infinite state space (due to the use of a dense time model). In our experiments, most of the requirements can be reduced to reachability properties, so we can use on-the-fly model-checking techniques.

We interpret a GTS by a Fiacre process whose parameters consist of all its state and flow variables. Each transition $\mathtt{g} \vdash \mathtt{evt} \rightarrow \mathtt{e}$ in the GTS is (bijectively) encoded by a transition that matches the guard `g` and updates the variables to reflect the effect of `e` plus the assertions. Each transition can be labelled with time and priorities constraints to take into account the `extern` declarations of the node. This translation is straightforward since all the operators available in AltaRica have a direct equivalent in Fiacre. Hence every state/transition in the GTS corresponds to a unique state/transition in Fiacre. This means that the state (reachability) graph of a GTS and its associated Fiacre model are isomorphic. This is a very strong and useful property for formal verification, since we can very easily transfer verification artefacts (such as counterexamples) from one model back to the other.

The close proximity between AltaRica and Fiacre is not really surprising. First of all, both languages have similar roots in process algebra theory and share very similar synchronisation mechanisms. More deeply, they share formal models that are very close: AltaRica semantics is based on the product of "communicating automata", whereas the semantics of Fiacre can be expressed using (a time extension of) one-safe Petri nets. The main difference is that AltaRica provide support for defining probabilities on events, whereas Fiacre is targeted towards the definition of timing aspects. This proximity in both syntax and semantics is an advantage for the validation of our tool, because it means that our translation should preserve the semantics of AltaRica on models that do not use extern laws to define probabilities and time. We have used this property to validate our translation by comparing the behaviours of the models obtained using Cecilia OCAS simulation tool and their translation. For instance, in the case of the CAS system of Sect. 4, we can compute the set of cuts corresponding to Safety Property 1 (loss detection) by checking an invariant of the form $((\mathtt{F1.O} = \mathtt{Loss}) \vee (\mathtt{F2.O} = \mathtt{Loss}) \Rightarrow (\mathtt{F3.O} = \mathtt{Loss}))$. In both cases—with and without FDI—we are able to compute the exact same set of cuts than Cecilia OCAS. This is done using the model-checker for modal mu-calculus provided with Tina, which can list all the counterexamples for a (reachability) formula as a graph. More importantly, we can use our approach to compute the timed counterexample described at the end of Sect. 4. All these computations can be done in less than a second on our test machine.

We have used our toolchain to generate the reachable state space of several AltaRica models [1]: RUDDER describes a control system for the rudder of an A340 aircraft [5]; ELEC refers to three simplified electrical generation and power distribution systems for a hypothetical twin jet aircraft; the HYDRAU model describes a hydraulic system similar to the one of the A320 aircraft [9]. The results are reported in Table 1. In each case we indicate the time needed to generate the whole state space (in seconds) and the number of states and transitions explored. We also give the number of state variables as reported by Cecilia OCAS. All tests were run on an Intel 2.50GHz CPU with 8GB of RAM

[1] All the benchmarks tested in this paper are available at https://w3.onera.fr/ifa-esa/content/model-checking-temporal-failure-propagation-altarica

Table 1. State space size and generation time for several use cases.

Model	**time** (s)	**# states**	**# trans.**	**# state vars**
RUDDER	0.85	$3.3\,10^4$	$2.5\,10^5$	15
ELEC 01	0.40	512	$2.3\,10^3$	9
ELEC 02	0.40	512	$2.3\,10^3$	9
ELEC 03	101	$4.2\,10^6$	$4.6\,10^7$	22
HYDRAU	1800	—	—	59
CAS	0.40	729	$2.9\,10^3$	6
CAS with `Pre`	46	$9.7\,10^5$	$4.3\,10^6$	10

running Linux. In the case of model HYDRAU we stopped the exploration after 30 minutes and more than 9.10^9 generated states. The state space is large in this benchmark because it models the physical a-causal propagation of a leak, so a leak can impact both upward and backward components and trigger a reconfiguration, multiplying the number of reachable states. In all cases, the time needed to generate the Fiacre code is negligible, in the order of 10 ms.

Our models also include two versions of the complete CAS system (including the detectors, the alarm and the switch); both with and without the delay functions `D1` and `D2`. The "CAS with `Pre`" model is our only example that contains timing constraints. In this case, we give the size of the state class graph generated by Tina, that is an abstract version of the state space that preserves LTL properties. We can use Tina to check temporal properties on this example. More precisely, we can check that `F3` has the *loss detection convergence* property. To this end, a solution is to add a Time Observer to check the maximal duration between two events: first, a `obs_start` event is triggered when the output of `F1` or `F2` changes to `Loss`; then an `obs_end` event is triggered when the output of `F3` changes to `Loss`. The observer has also a third transition (`obs_err`) that acts as a timeout and is associated with a time interval I and is enabled concurrently with `obs_end`. Hence, Time Observer ends up in the state yield by `obs_err` when the output of `F3` deviates from its expected value for more than d units of time, with $d \in I$. We have used this observer to check that the *convergence latency* of the CAS system equals 3, when we assume that the delays are in the time interval $[1, 3]$. The result is that `obs_err` is fireable for any value of d in the interval $[0, 3]$, while `obs_err` is not fireable if $I =]3, \infty[$. These two safety properties can be checked on the system (plus the observer) in less than 0.6 s.

6 Conclusion and Related Work

Our work is concerned with the modelling and analysis of failures propagation in the presence of time constraints. We concentrate on a particular safety property, called *loss detection convergence*, meaning that the system applies an appropriate and timely response (e.g. isolation) to the occurrence of a fault before the failure is propagated and produces unwanted system behaviours. Similar problems were addressed in [18], where the authors describe a process to model

Failure Detection Isolation and Reconfiguration architecture (for use on-board satellites) that requires to take into account failure propagation time, failure detection time, and failure recovery time. However, these needs are not matched by an effective way to check for the safety of the systems. Our approach provides a solution to model these timing constraints within AltaRica. We also provide an automatic transformation from Time AltaRica models in one of the input formats of Tina. We show that two interesting problems—computing "timed cuts" and bounding the convergence latency of a node—can be reduced to a decidable model-checking problem.

Several works have combined model-checking and AltaRica, the archetypal example being the MEC tool [14] that was developed at the same time than the language. More recently, Bozzano et al [12] have defined a transformation from AltaRica Dataflow to the symbolic model-checker NuSMV. While this tool does not support complex timing constraints, it offers some support for Dirac laws (and implicit priorities) by encoding an ad-hoc scheduler. The use of symbolic model-checking techniques is interesting in the case of models with a strong combinatorial blow up, like for instance model HYDRAU of Sect. 5. Nonetheless, even though Tina also includes BDD-based tools, no approaches allow to combine the advantages of both realtime and symbolic model-checking techniques.

Realtime techniques are central to our approach. We define an extension of AltaRica where timing constraints can be declared using temporal laws of the form `law (evt) = "[a,b]"`, with a semantics inspired by Time Petri nets. As a result, we can apply on AltaRica several state space abstractions techniques that have been developed for "timed models", such as the use of DBM and state classes [6]. In a different way, Cassez et al. [13] have proposed an extension of AltaRica with explicit "clock variables", inspired by Timed Automata, where clocks are real-valued flow variables that can be used inside the guards of events. Their work is mainly focused on the verification of behavioural properties and centres on the encoding of urgency and priorities between events, two notions that are naturally offered in Fiacre. Also, our extension is less invasive. If we ignore the `extern` declaration then we obtain valid AltaRica code. More research is still needed to further the comparison between these two approaches in the context of safety assessments.

Aside from these works on AltaRica, recent works centred on combining failure propagation analysis and timing constraints, such as [10]. This work defines an automatic method for synthesising *Timed Failure Propagation Graphs* (TFPG), that is an extension of the notion of cut-sets including information on the date of events. TFPG provide a condensed representation that is easier to use than sets of timed cuts.Therefore, it would be interesting to use this format in our case.

For future work, we plan to adapt our translation to a new version of the AltaRica language—called AltaRica 3.0, or OpenAltaRica [15]—that imposes less restrictions on the computation of flow variables. We also want to apply our approach to more complex industrial use cases, involving reconfiguration time besides failure detection and isolation; or even systems with humans and reaction time in the loop.

References

1. Arnold A., Griffault A., Point G., and Rauzy A. The AltaRica formalism for describing concurrent systems. *Fundamenta Informaticæ*, 40(2, 3):109–124, 2000.
2. Rauzy A. Mode automata and their compilation into fault trees. In *Reliability Engineering and System Safety*, 2002.
3. Rauzy A. Guarded Transition Systems: a new states/events formalism for reliability studies. *Journal of Risk and Reliability*, 222(4), 2008.
4. Arnold A., Point G., Griffault A., and Rauzy A. The AltaRica formalism for describing concurrent systems. *Fundamenta Informaticae*, 40(2, 3):109–124, 1999.
5. Bernard R., Aubert J.-J., Bieber P., Merlini C., and Metge S:. Experiments in model based safety analysis: Flight controls. *IFAC Proceedings Volumes*, 40(6):43–48, 2007.
6. Berthomieu B.,Ribet P.O., and Vernadat F. The tool Tina – construction of abstract state spaces for Petri Nets and time petri nets. *International Journal of Production Research*, 42(14), 2004.
7. Berthomieu B., Bodeveix J.-P., Farail P., Filali M., Garavel H., Gaufillet P., Lang F., and Vernadat F.. Fiacre: an intermediate language for model verification in the topcased environment. In *ERTS 2008*, 2008.
8. Bieber P., Bougnol C., Castel C., Heckmann J.-P., Kehren C., Metge S., and Seguin C. Safety assessment with AltaRica. In *Building the Information Society*, pages 505–510. Springer, 2004.
9. Bieber P., Castel C., and Seguin C. Combination of fault tree analysis and model-checking for safety assessment of complex system. In *European Dependable Computing Conference*, pages 19–31. Springer, 2002.
10. Bittner B., Bozzano M., and Cimatti A. Automated synthesis of timed failure propagation graphs. In *Proc. Int. Joint Conf. of Artificial Intelligence (IJCAI-16)*, pages 972–978, 2016.
11. Bouyer P., Dufourd C:, Fleury E:, and Petit A. Updatable Timed Automata. *Theoretical Computer Science*, 321(2-3):291–345, 2004.
12. Bozzano M., Cimatti A., Lisagor O., Mattarei C., Mover S., Roveri M., and Tonetta S. Symbolic model-checking and safety assessment of AltaRica models. *Electronic Communications of the EASST*, 46, 2012.
13. Cassez F., Pagetti C., and Roux O. A timed extension for AltaRica. *Fundamenta Informaticæ*, 62(3-4):291–332, 2004.
14. Griffault A. and Vincent A. The MEC 5 model-checker. In *International Conference on Computer Aided Verification*, pages 488–491. Springer, 2004.
15. Prosvirnova T., Batteux M., Brameret P.-A., Cherfi A, Friedlhuber T., Roussel J.-M., and Rauzy A. The AltaRica 3.0 project for Model-Based Safety Assessment. *IFAC Proceedings Volumes*, 46(22):127–132, 2013.
16. Rauzy A. Altarica Dataflow language specification version 2.3. Technical report, Ecole Centrale de Paris, June 2013.
17. Teichteil-Königbuch F., Infantes G:, and Seguin C. EPOCH probabilistic Model-Checking. In *Model Based Safety Assessment workshop*, Toulouse, France, 2011.
18. Thomas D. and Blanquart J.-P. Model-based RAMS & FDIR co-engineering at Astrium satellites. In *ESA Special Publication*, volume 720, page 33, 2013.