Early Error Detection for Fault Tolerance Strategies

Thomas Robert, Matthieu Roy, Jean-Charles Fabre

Early Error Detection for Fault Tolerance Strategies

Thomas Robert
Institut Telecom- Telecom ParisTech - LTCI
46, rue Barrault, F-75634 Paris CEDEX 13, France
Email: {firstname.lastname}@telecom-paristech.fr

Matthieu Roy, Jean-Charles Fabre
LAAS-CNRS; Université de Toulouse
Toulouse, France
Email: {firstname.lastname}@laas.fr

Abstract

In this paper we present an integration of early run-time monitors in real-time systems to improve their fault tolerance properties. Early Error Detection is a mechanism that provides a theoretically optimal run-time error detection service, based on a formal specification of an application, e.g., given by a timed automata. We show how our approach can improve classical fault tolerance strategies by investigating two use-cases, namely for a design pattern that provides several degraded modes of operation, and in on-board avionics safety critical systems.

Keywords: run-time monitoring; fault-tolerant; generation; early error detection

1. Introduction

The efficiency of error detection mechanisms is of primary concern for dependable system designers, since it impacts the efficiency of fault-tolerance mechanisms for error-detection and recovery strategies. When an error is detected, these fault-tolerance mechanisms modify the system state to guarantee that the service is still delivered, be it in nominal or in a degraded mode.

In conventional implementation of error-detection and recovery strategies, the approach is reactive in the sense that recovery mechanisms are triggered when the current state of the system is detected as erroneous. This approach has two impacts: i) the latency of error detection can be very high, and indeed the error is never detected before it actually happens, and consequently ii) the time left to perform a recovery action is reduced.

Improvements in error detection can take advantage of recent progresses in run-time verification, by integrating error detectors generated from specifications [12]. The inability to assess the gain in predictability brought by such detectors was the reason why they were not widely adopted. Their cost was also most of time prohibitive in terms memory and CPU usage. These aspects can now be mastered and modeled. The behavior of the new generation of formally generated error detectors can be formally inferred including timing aspects, [5, 10, 19, 17].

In this paper we propose an approach that is *proactive* in the sense that it tries to anticipate the detection of an erroneous state in the actual system. Our method performs the detection based on an abstract model of the computation, defined by *timed automata*.

In previous work [17], we showed that this method, the *Early Error Detection (EED)*, leads to detect in advance a problem that will appear later in the actual system execution. The model is embedded in the implementation and is animated by our monitoring system, based on events related to the system execution. Intuitively, when the continuation of the execution in the model can only reach a failure state, an error is raised. In this work we show how we can use this approach to anticipate effective problems and perform recovery actions *before* the actual error takes place. We will detail the principle of early error detection, the way it can be implemented in real-time operation systems, and finally how usual fault tolerance strategies can benefit from our monitoring system.

In Section 2, we present and analyze related works on this topics. Section 3 describes in more details the principle of Early Error Detection and its formal foundations. Section 4 focuses on the description of the input formalism to our detector generation framework: timed automata. It also explains how early error detection can be interpreted with respect to this model. Section 5 briefly presents the implementation of our framework in Xenomai. Section 6 is devoted to explain how this mechanism can be used in usual fault tolerance design patterns. Section 7 sums up the contribution and provides some insights on future works.

2. Problem statement & Related work

Several surveys provide insights on error detection through model driven engineering [7, 8, 12]. Models are often related to academic formalisms, e.g., temporal logics or timed automaton [6, 2].

In this field of research, run-time monitors are independent detection components integrated in a system to enhance its error detection capabilities. The monitor observes the system state and check on-line user-defined properties. Thus, it emits error signals as soon as one of the properties being assessed is violated. This architec-

ture is now well established and can be found in several key papers along the past decades [9, 15, 14, 12, 10].

In those architectures, two phenomena contribute to detection latency. First, there is the latency introduced by the communication mechanisms used between the system and the monitor to relay observation. Second, there is also an issue when specifications are not expressed as simple invariants, leading to a run-time evaluation overhead.

As soon as the monitor has to store information in order to correlate the system state at two distinct instants, then the algorithm used for the decision procedure may introduce some latency waiting for useless observations. In order to avoid such a situation, the notion of early error detection first appeared in [15] for monitoring dynamic deadlines on a fixed set of events (chronicles). In [17], we described an implementation of such a run-time monitor detecting on the fly when the system execution is no longer matching behaviors described by a variant of timed automata [2]. A similar approach has been suggested in [5]. Nevertheless, it was not implemented nor detailed from the algorithmic point of view.

In this paper, we propose to integrate our early monitoring system for real-time applications to usual handling of degraded modes of operation. We rely on our run-time monitor implemented for a real-time kernel, Xenomai [1], focussed on application tasks.

3. Early detection for trace monitors

We pointed out on the fact that there exists a class of monitors providing early error detection services. We recall in this section the ideas behind this notion of early error detection and their formalization in terms of traces.

3.1 Correct behavior and error symptoms

We briefly recall some terminology [4] to identify and describe the threats to dependability: failures, errors and faults.

These notions assume that the system interface, its boundaries, is known. The expected service of a system is embodied by its nominal behavior. A *failure* is defined by an unexpected behavior. It occurs when there is a gap between the system actual behavior and the expected one according to its specification. An error is an internal state of the system leading to failures. Finally, the fault is the cause of errors. The study of faults is out of the scope of this paper. Nevertheless, the notion of early error detection is directly related to the relationship between errors and failures. Because the duration between the fault activation and the failure may be quite large, there is a real interest to optimize the error detection latency. It would provide more time for recovery actions.

In order to identify erroneous states, a well spread method assumes the following claim: *any state that has not been declared correct by the system designer is a priori erroneous*. We use this assumption and derive an error detector from a description of the system correct states.

There are usually two ways to specify system correct states: through logical formulas or transition systems. Because many architectural design languages attach transition systems to their component structures, we considered such a formalism to specify correct system behaviors (UML and statecharts [22], AADL and its behavioral annex[11]).

In order to detail the notion of early error detection, we need to formalize the trace notation used to represent system executions.

3.2 From Bad prefixes to early detection.

We are relying on a formalized representation of a system execution: traces. Timed traces are the most common representation of real-time system executions.

Timed traces are sequences of alternating events (*start, stop*, ...) and quantitative durations (1, 5 time units). Such trace model has been extensively studied in [3]. Alternating durations and events ensures that at most one event can occur at any given instant. The concatenation symbol '.' is used to separate events and durations, e.g. $start.10.stop$. The durations considered are unit free : no physical duration is actually associated to these numbers. The physical duration unit, e.g. seconds or milliseconds, can be defined afterward.

In order to reason about the detection process, prefixes of traces have been identified as a key concept [5, 13]. For any traces v and u, if there exists a trace w such that $v.w = u$, then v is a prefix of u, and u a continuation of v. In practice, prefixes and continuations can be seen as the past and future of a trace. The set of prefixes of a given timed trace represent all the *past states* of u.

If a trace is not a prefix of some valid trace (let $\mathcal{L}_V$ be the set of valid traces), then this trace is called a bad prefix. A bad prefix, by definition, does not admit any continuation in $\mathcal{L}_V$. It means that the system cannot meet its specification, whatever is executed later in the system. Eventually, any bad prefix corresponds to an erroneous state, and leads to the system failure.

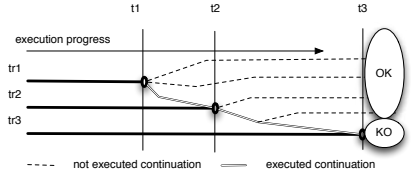

Figure 1. Good and bad prefixes

The prefixes of a given trace are totally ordered. We presented a detailed justification for considering the following claim in [17]: *the lower bound of the prefixes of a bad prefix, that are also bad prefix, is a bad prefix except in rare artificial cases*. In other words, there is always a trace that allows identifying the exact border between correct and erroneous traces at run-time. Thus, this lower bound is call the shortest error symptom. In practice, this

trace corresponds to the first instant from which the specification can no longer be satisfied. The Figure 1 represents three traces such that $tr1$ is the prefix of $tr2$, and $tr2$ is the prefix of $tr3$. On the figure, the shortest error symptom of $tr3$ is between $tr2$ and $tr3$.

3.3 Error signalling and Monitor Synchronization

As defined above, the monitor *can* signal an error as soon as a shortest error symptom is observed. One of the advantages of such a formal definition of the shortest error symptom is to provide the foundations to build a formal definition of error detection latency.

The shortest error symptoms identify the earliest instants at which the specification allows to distinguish correct from incorrect executions. If an error detector can signal errors as soon as the system execution matches a shortest error symptom, then it means that this detector has a "zero latency".

One of the core issues when dealing with run-time monitor implementations lies in their architecture. The problem comes from the communication methods used to connect the different elements of the monitor. The usual architecture of the run-time monitor is made of three layers shown in Figure 2: an observation layer, a decision layer, and an error dispatch layer or error handling layer.

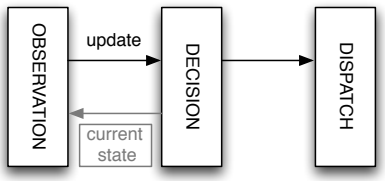

Figure 2. Abstract architecture of a monitor-based detector

Two main issues have to be handled : the location of the layers, the way they synchronize and communicate, and how observations are delivered to the decision layer. We considered the following setting :

- The layers are deployed on a same node. Our implementation of the monitor is not distributed.
- The layers synchronize themselves thanks to the scheduling services provided by the underlying runtime support (here, it is the Xenomai real-time operating system).
- The observations are handled according to a FIFO policy by the decision layer.

These assumptions guide the design of the monitor using Xenomai programming interface.

The usual synchronization scheme assume that the monitor is a purely reactive component waiting for new observations. A monitor enforcing early error detection tries to identify shortest error symptoms. One of the nice properties of shortest error symptoms is to catch both timing and behavioral errors (wrong event).

Consider the following set of correct traces : $\mathcal{L}_V = \{start.X.stop \mid X \in [0, 20[\}$. The trace $start.24.stop$ is a bad prefix and corresponds to a deadline miss. The shortest error symptom of $start.24.stop$ is $start.20$. The monitor will have to wake up as soon as the state represented by $start.20$ is reached. For these reason, the monitor cannot be implement as a fully reactive system. Timers can be used for this purpose.

The next section explains how to generate a fully operational monitor that provides early error detection for a set of valid traces specified by a timed automaton.

4. Shortest error symptoms Monitor

In this section, we first highlight the kind of errors and failure targeted through our approach. Then, we explain how timed automata can help to specify and check at runtime relevant misbehaviors. Finally, we show where shortest error symptoms are involved.

4.1 Targeted errors and their symptoms

There are usually two kinds of error detection services: generic or application dependent detection services. Using the specification of application valid behaviors, we are targeting the second class of services.

The kind of formalism used for specifying valid behaviors has a strong impact on the kind of detected errors.

4.1.1 Events and timing

Considering timed traces of events, the detector is targeting errors tempering with the control flow of applications. Many operating system have watchdog timers and deadlock detectors. Nevertheless, these mechanisms lack application specific knowledge: watchdog detectors suffer from high latency and poor error confinement, and deadlock detection services lack coverage. The aim here is to use application dependent knowledge to identify wrong timing or suspicious event involved in data race or deadlock conditions.

Three types of constraints can be defined on the elements of an execution trace:

- Compulsory ordering between events, induced by the natural sequence of events,
- Upper bound on the time between two events,
- Lower bound on the time between two events.

The aim of the error detector is to trigger an error signal as soon as it observes that the system will no longer be able to comply with its specification. Whereas the scheduling theory is mostly concerned with comparing

different types of upper bounds (i.e. deadline and execution times), EED takes advantage of leveraging information on both lower and upper bounds to identify possible inconsistencies in an erroneous execution. In few words, if we can trace a function execution progress and compare it to its deadline, then if the function is late on schedule, the monitor can decide to trigger an error signal before actually reaching the deadline.

We use formal model to generate the monitor. Thus, if the observations match a shortest error symptom it proves that the specification is not respected.

4.1.2 Example of early detection

Consider the following execution scenario for a task relying on a shared resource for its execution. The task has two modes, i.e. , two execution paths are possible. The shared resource represents a device from which data is read, and the task waits to obtain the read value to complete its execution. Reading the value entails a locking time on the calling task. This locking time depends on the execution path, since the resource is used differently in the two execution paths. In the first path, the call is synchronous and entails a locking time of 10ms. In the second path, the resource is not actually used in the sense that the task only gets the previous measure that is in the cache. In this case, the locking time is so low that it can safely be considered instantaneous. Another (global) timing constraint for both paths is that the completion of any of these two paths must occur within 20ms. In this example, the second path can be considered as a degraded mode of operation of the task, since it will use degraded (with respect to freshness) information to perform the same task, but with a lower WCET. Let us observe the description of the first path. The combination of the minimal locking time with the deadline allows us to claim that if the task is started, and accesses the resource t ms after its starting time, with $t > 10$ms, then the task will either violate the minimal locking time, or miss the deadline as depicted in Figure 3[1].

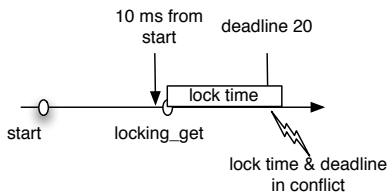

Figure 3. Conflict between timing constraints

An early termination of the reading on the shared resource as well as the deadline miss could be interpreted here as the observable consequences of either an error in the execution flow of the critical section, or the propagation of a late timing failure before the critical section.

When such a situation can be identified in advance early error detection will provide a mean to save time and avoid the execution of non-reversible actions that would for instance propagate the errors outside of the monitored application.

In most cases inconsistencies come from a clash between the knowledge of what should happen (i.e. deadlines for instance) what would happen in normal circumstances (i.e. the minimal locking time), and what is actually observed. In the next subsection we describe a formalism in which such information can be easily integrated in order to get the best of early detection.

4.2 Timed automaton

Timed automata are one of the leading formalisms used to formally specify real-time behaviors. As any other type of automaton, they can be seen as a handy way to define sets of traces (representing valid execution) or analyze state properties (liveness and safety). Through the modeling of the previous example, the descriptive power of this formalism will be illustrated based on the content of the seminal paper [2]

Finite state machines are straightforward representations of the state that can be reached during an execution of a system. Transitions represent possible state changes and are statically defined. In the example described above, there would be at least four states: one identifying the initial state from which the execution of a cycle starts; one identifying the intermediate state from which two transitions, representing the two possible execution paths (nominal and fast), can be fire; and two distinct states representing the critical section in each path.

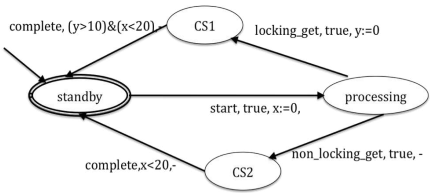

Figure 4. A Timed Automaton to specify valid executions

In addition to states and transition inherited from classical automata, timed automata introduce special timing variables, that can be seen as resettable clocks with uniform time elapse. Edge are labelled with triplets: an *event label* allowing to identify the transition by a name, a *guard* being the condition under which the transition can be fired, and the set of *clocks* that have to be reset when the transition is fired.

For our example of Figure 4, the edge between states standby and processing is a transition that can always

[1] In this case, the WCET is at least $t + 10$, so WCET $\geq t + 10 > 20$, whereas WCET < 20 by the deadline specification

be fired, and is fired when event start is observed, leading to reset the clock x. The transition from the state CS1, representing the execution of the critical section in the first execution path, and the standby state is identified by the event complete. This condition has a guard combining a condition on the clock that count the time elapse since the beginning of the current execution of the task, x, and the clock storing the time spent in the critical section, y. This transition can only be fired from states satisfying the guard, i.e. those with clock values such that $(x < 20) \wedge (y > 10)$.

Notice that if an event is observed in a state s whereas no corresponding transition can be fired from s, then it means that the event was unexpected, either because it was in a bad timing, or because it was forbidden. Guards allow defining conditions in which events are forbidden. The definition of final states, i.e., states that need to be reachable, provide the means to define the completion condition of an execution corresponding to requirements related to bounded liveness.

Definition 1 (Accepting condition on traces) *A trace is valid if and only if*

- *It can be executed on the automaton starting from the initial state (state identified by inbound arrow), with all clocks set to zero.*
- *The last state reached through the execution of the trace is a final state (i.e., a state identified by a double circle).*

In the remainder of this paper, the requirement that a final state is actually reachable from the current state will be called the *reachability* condition. Whereas there is a straightforward way to identify unexpected events, it is far more difficult to identify errors due to the violation of the reachability condition defined by the automaton. This is exactly where the EED property defines the objective to be met.

Intuitively, a monitoring system, based on a description made of timed automata, that complies with the EED property performs the steps described in Figure 5 (a more detailed description can be found in [17, 18]).

The major point in this skeleton is to be able to efficiently assess at run-time the reachability property, i.e. , we need to have an efficient way to answer the following question at every time instant: *"From the current state, is there a continuation that reaches a final state?"*. The following subsection presents our approach to reach this goal, by using a modified version of time abstractions.

4.3 Final states reachability conditions

There are basically two ways to check at runtime the reachability condition. The naive method is to browse the automaton again and again to check the reachability condition on final states. Although simple and easy to implement, this approach is not only very expensive, but, more importantly, the time needed at each step to check the reachability property is unpredictable.

1. track the current state of the system within the automata by synchronizing it to the observations
2. **event-based error detection:** when an event e occurs:
   (a) check whether e is forbidden, because no such transition exists,
   (b) if a corresponding transition exists, check that firing the transition will not put the system in a state from which the reachability of final states is violated.
3. **pure timing error:** handle the case in which the reachability condition is violated, due to time elapse and no transition firing, leading typically to deadline misses.

Figure 5. Major steps of an EED-compliant monitoring system

The approach, we advocate, is to precompute most of the reachability condition and encode it in the monitor, in order to have efficient and predictable reachability tests. Obviously, in this pre-computation step, special care has to be taken to minimize the amount of stored information, limiting it only to what is needed at run-time for the detection of violations of the automaton specification.

Intuitively, this means that the transformation will unfold (or compile) timing constraints and transform guards on transitions so that the evaluation of reachability condition can be performed when firing the transition, without having to browse the automaton. As clock constraints can also be defined on control states of the automaton, it is called an invariant. Thus, it is possible to define the condition under which control states are still alive through a conjunction of inequations defining the upper bound of each clock.

In our example, this results in:

- Adding to all nodes a deadline on x at 20, and adding it also on each transition.
- Updating the constraint on the transition on locking_get by requiring that x is lower than 10 when firing.

When comparing the new automaton with the original specification, it appears that the transformation has made explicit all conditions on clocks that used to be implicit[2]. In most cases a human made timed automaton transforma-

[2]Recall the example: the process could access the shared data at $x > 10$ because no guard prevent it locally —the transition can be fired—. Nevertheless, it will lead the system to the kind of conflict depicted in 3 later on.

tion is not possible, since it is a very burdensome way to specify behaviors.

Yet, such a transformation can be inferred automatically from a well established transformation: *timed abstraction*. The time abstraction splits up control states as soon as the enabled and valid transitions differ, according to the clock values. In our approach, we used the tool KRONOS [21] to perform this transformation. The aim was to allow local decision in the sense that steps 2 and 3 of Figure 5 can be done in a greedy way. Time abstraction adds clock upper bounds to control states, and integrates the reachability test and the guards in timing bounds put on each state. Hence, the challenge was to be able to interpret all the implicit conditions entailed on transition by the requirement of reachability of final states. This transformation has been proven to be always possible for deterministic timed automata even if it can sometimes lead to an exponential blow up in automaton complexity. This transformation is called time abstraction and the full description of this technique is beyond the scope of this paper.

In the next section, we will discuss how we performed this transformation, and how we used it for our monitoring system that we developed for the operating system Xenomai.

5. Generating EED from timed automaton

This section presents our proposal for generating real-time error detectors that implement the early error detection property for Xenomai applications. In the previous section, we have briefly presented how a timed automaton can be normalized so that it can be used to identify on-the-fly shortest error symptoms. The next step is to embed such a model in a detector observing the system activity.

5.1 Observation layer

This layer is integrated at compilation time in the application source code. Observations are collected through calls to a function triggering the verification in the decision layer. These function calls are inserted by hand within the source code through C macros. The macro has a single parameter that is used to specify which event is observed. The set of observed events is listed in an `header (.h)` file that has to be included within each file where observations are performed.

In practice, it means that observations can be done in pseudo-concurrence as the verifications can be called by threads reaching observation macros at the same time. The monitor state is thus a shared variable. Each call to the verification function first reads this state, and then updates it. This instrumentation strategy is simplistic but our concern was not to provide another instrumentation tool.

5.2 Decision layer

The decision layer is implemented by a function that has access to the data structure storing the normalized automaton description, and especially the location invariants and the transition table. We use the KRONOS tool set [21] to normalize the automaton. Because KRONOS only provides a raw text format, we translate it in a C data structure. Then, this data structure is loaded during the initialization phase of the application.

As said before, the state of a timed automaton is made of a location index and a clock vector. The automaton state is only updated when an observation is performed. This update can affect both the location index and the clock vector. Clocks are increased or reset depending on the transition fired. Invariants are enforced by setting a timer according to the time left in each location. For instance, if we consider the example of the previous section, then the location $processing$ has to comply with "$X < 20$". Thus, as soon as this location is entered (firing the transition "$start$"), then a timer is set to enforce the deadline on the clock X. This timer is set to expire at $20 - X$ time units later.

Each time an event is received, then the transition table of the currently occupied location is searched. The event is validated if and only if the condition is true: there exists a transition with its guard satisfied by the current clock vector, such that its label corresponds to the event being checked. Each time an event is validated, then the transition is fired and a new location is entered, then the timer has to be reset or updated.

The verification function has three possible behaviors :

- check an event and validate it : the function return normally
- check an event and declare it invalid : i) the verification function releases a recovery task concurrent to the calling task with a higher priority, ii) the verification function returns an error code like any C function.

The two methods proposed to signal the error are intended to allow either local or global error handling policies.

5.3 Synchronization and scheduling during verification

The monitor state is a shared variable used for read and write accesses during the verification of an event. Thus, it is necessary to ensure at least mutual exclusion on verifications. Moreover, the verification call enforces a strong synchronization between the application and the monitor. In order to offer maximal error confinement capabilities, the verification is performed synchronously.

For this reason, the verification has to be performed as fast as possible. Before actually starting the verification, the verification function protects itself from any preemption by locking the scheduler. Once the verification is finished the scheduler is released just before returning from the verification function.

This synchronization method can be considered as disappointing with respect to usual best practices in scheduling theory. Verification steps deactivate the scheduler. Thus, a lower priority task can trigger a verification step whereas a higher priority task should have been rescheduled in a close future.

We are dealing with error detection at the scale of a set of tasks. A same automaton describes synchronizations and timing constraints on tasks of the same criticality, a priori. Thus, lower priority tasks may be as relevant as higher priority tasks in order to detect errors. One could consider that the verification task is supposed to have at least a higher priority than any task monitored.

Moreover, the recovery actions triggered by the monitor will comply with application scheduling policy: either the recovery actions are executed by the task (at its own priority level) that called the detector, or by an independent recovery task when global recovery action is needed.

In both cases, the user can rely on the usual properties of the fixed priority scheduler of Xenomai to analyze and predict when the recovery task may finish.

6. Integration of EED into fault tolerant systems

The objective of this section is to illustrate the interest of EDD in fault-tolerant systems. The first example is based on conventional design patterns used to implement degraded modes of operation. A nominal mode of operation is developed together with a series of degraded modes of operation. The use of EED maximizes the remaining time for running a degraded mode of operation. The second example is extracted from on-board avionics safety critical systems [20]. The "command-monitor" design pattern used in the Airbus 320 to 380 can be improved using EED, i.e., improving the error detection coverage of the monitor in the temporal domain.

6.1. Implementing degraded modes of operation

Let us consider a real-time periodic application that implements a service defined by a timed specification, as described in previous sections.

6.1.1 Design

The application, in its nominal mode, implements the full specification Spec(Nominal) which is associated with a worst case execution time WCET(Nominal). We also suppose that there exists a sequence $(\mathsf{Deg}_i)_{i=1..n}$ of degraded implementations of the application, with restricted quality of service, that implement degraded specifications $(\mathsf{Spec}(\mathsf{Deg}_i))_{i=1..n}$. These degraded implementations are, by nature, associated with lower worst case execution times $(\mathsf{WCET}(\mathsf{Deg}_i))_{i=1..n}$:

$$\forall i < n : \begin{cases} \mathsf{WCET}(\mathsf{Deg}_{i+1})) & \leq & \mathsf{WCET}(\mathsf{Deg}_i)) \\ \mathsf{WCET}(\mathsf{Deg}_i)) & \leq & \mathsf{WCET}(\mathsf{Nominal})) \end{cases}$$

In safety critical systems, there exist at least two modes: the nominal mode and a safe stop mode. Degraded versions permit to provide more flexibility and availability by trying to reconfigure the application to avoid, when possible, stopping the application. A traditional reconfiguration mechanism may stop the nominal application when a failure occurs, and use one of the available degraded modes of operation. In a real-time system, it is crucial that the reconfiguration occurs as soon as possible, but available failure detectors will not be able to detect a deadline miss before the actual deadline expires. Using Early Error Detection, we are able to catch the failure sooner in the execution, i.e., before the deadline is actually missed, and use the remaining time budget to launch one of the possible degraded modes of operation, as shown in Figure 6.

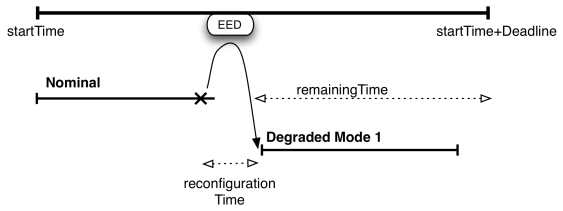

Figure 6. EED-based reconfiguration

In this example, the application runs in nominal mode with a given time budget (expressed by the variable Deadline). The detection engines monitors the application and signals any deviation to the nominal mode specification. When such a signal occurs, the application is stopped, the reconfiguration engine computes the remaining time budget (remainingTime), and restarts the application, using a degraded mode whose worst case execution time is compatible with the remaining budget remainingTime.

6.1.2 Algorithm

As mentioned above, the application is a periodic task that is scheduled by the system for a maximum budget time, i.e., with a static deadline. Let Spec(Nominal) be the nominal specification of the service, and $(\mathsf{Spec}(\mathsf{Deg}_i))_{i=1..n}$ be degraded specifications with reduced quality of service. We associate with each specification an EED monitor that checks online whether an implementation satisfies its specification. As soon as a deviation is detected, the monitoring engine signals it to the reconfiguration engine. In practice, each specification is implemented in a separate module, and is available to the reconfiguration engine that can dynamically chose which version is to be run, given a time budget. When an EED signal occurs, the reconfiguration engine first computes the remaining time budget, then looks for the degraded mode that has an expected worst case execution time closest to the remaining time, and then launches it. If no such mode exists, this means that it is not possible to reconfigure the application in a temporally satisfactory

manner, and a failure is signaled. The code executed when such a reconfiguration occurs is shown in Figure 7.

```
// Initial nominal mode
startTime = System.currentTime()
EED.startMonitoring(Spec(Nominal))
Exec (Nominal)

// Code executed when an EEDSignal occurs
when EEDSignal is received
execTime=System.currentTime()-startTime
remainingTime=budgetTime-execTime-reconfigurationTime
availVersions = {Degi : WCET(Degi)<remainingTime}
  // only look at degraded versions that are supposed
  // to finish before the actual deadline
if availVersions=Ø
then // in this case it is no longer possible to match
     // the deadline
     signal ApplicationFailure
else // choose the best degraded mode and launch it
next=Degj  such that WCET(Degj)=max(WCET(availVersions))
EED.startMonitoring(Spec(next))
Exec(next)
```

Figure 7. Nominal and reconfiguration code

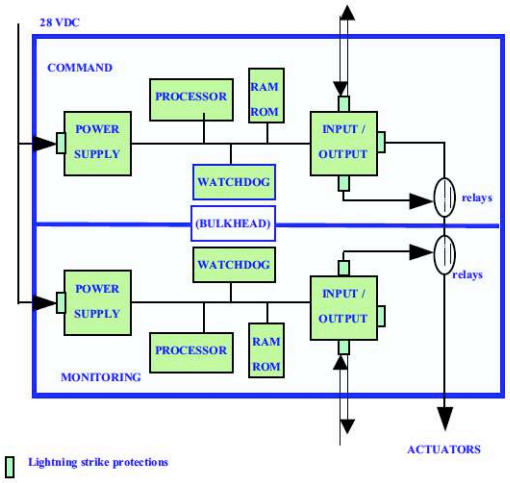

Figure 8. Computer global architecture

Notice that this approach assumes that degraded modes can be started at any time, and that nominal and degraded modes do not interfere with each other. This is the case, for example, when the periodic task is a calculus task, or a filter task, that do not write information in memory before it is finished.

6.2. Integration into avionics systems

The electrical flight control systems in modern aircrafts control the slats, flaps and spoilers. These systems have very stringent safety requirements (in the sense that the runaway of these control surfaces is generally classified as Catastrophic and must then be extremely improbable). The use of such system is nowadays common practice in civil aircraft, from the A320 to the big Jumbo A380 developed by Airbus [20], but also in concurrent aircrafts done by Boeing.

6.2.1 Technical approach

In Airbus systems, the implementation of safety critical functions relies on duplication and diversification. Such function is implemented with a PRIM ("active") and a SEC computer ("standby", hot spare). The PRIM computer is responsible for service delivery until it fails, i.e. becomes silent. The assumption coverage of the fail silent assumptions relies on a typical architecture, based on the "command-monitoring" design pattern. The command channel ensures the function allocated to the computer (for example, control of a moving surface). The monitoring channel ensures that the command channel operates correctly. Command and monitoring are running in parallel on two different processors. Failure detection is mainly achieved by comparing the outputs between the command and monitoring commands with a predetermined threshold.

The diversified redundancy at system level (PRIM/SEC) enable safety requirements to be reached for certification purposes. From a practical standpoint, the dissimilarity is such that the SEC computer runs a simpler version of the command software. In summary, at software level, the architecture of the system leads to use four software packages (for instance ELAC/-COM, ELAC/MON, SEC/COM, SEC/MON) when, functionally, one would suffice.

6.2.2 EED integration and interest

The EDD mechanisms could be integrated into both the command and/or the monitoring channel of each computers. The EED mechanisms would improve the detection in the temporal domain, but also to the behavioral domain depending on the granularity of the functional software modeling. This implies, at least, that a model of the command specification has been developed for its runtime verification. The integration into the COM is a direct application of the use of EDD for handling several degraded modes of operation. It means that the COM software must be developed with degraded versions, the bottom version being the reset of the processor hosting the COM. The execution of the EED monitoring is thus collocated with the COM and thus the EED monitoring can access the event and data required to animate the model. This approach enables the COM to improve its behavior with respect to transient physical faults that could impair the outputs and their delivery time. The integration of EDD has an impact on the development of the COM but no impact on the whole COM-MON architecture. However, the EED monitor being run on the same processor with the COM is also subject to error propagation. Conversely, the integration of EED into the MON provides better separation of concerns and physical isolation between the COM and the MON. However, this solution implies a slightly different architecture or, at least requires more complex interactions between the COM and the MON, which is not the case in the standard configuration. Indeed, this solution requires

a more detailed interaction between the "command" and the "monitoring" channels. All events required to animate the model at runtime must be extracted and forwarded to the monitoring "channel". At present the interaction between the "command" and the "monitoring" channels is very limited, just a watchdog mechanism. We would assume that the communication channel between COM and MON in this case is reliable. All messages sent are correctly received in FIFO order until one fails. In both cases, the use of EED would improve the standard behavior. On the PRIM computer, instead of waiting for COM-MON output discrepancy, the EED-based monitoring will suspend the "command" channel as soon as possible and stop sending the "I'm healthy signal". This implies a switch to the SEC computer that can provide output results early. This approach enables the SEC channel to take over the lead without waiting the detection of the PRIM failure on outputs.

7. Conclusion

Many run-time monitoring frameworks are intended to be used during the debug phase of system development. Few of them have been extended in order to support the integration of error recovery actions at run-time, and all such prototypes are developed in Java as remote objects.

The interest of our solution lies in the way the detector is integrated in a real-time system in order to provide timely reaction. Our implementation in Xenomai benefits from the fact that Xenomai provides several ports of usual RTOS APIs, thus enabling the integration of heterogenous off-the-shelf applications. In this context, the monitor will ensure that all these applications behave as expected especially in the case they use API ports instead of Xenomai native system calls.

The overhead introduced by our approach can be decreased at least by a factor 2. An implementation of this service as a kernel module of Xenomai would decrease highly the time spent switching between kernel and user space during the verification. From our measurements [18], the time spent in system calls is always greater than the time spent checking events and computing deadlines (case study drawn from UPPAAL case studies). Our prototype is available at the following address: [16].

Interestingly, we showed in this paper that our approach can be easily integrated with usual resilience patterns, namely degraded modes of operation and the critical system that runs in many Airbus airplanes.

The next step is to consider a specification language that fully integrates the architectural dimension of modern real-time applications, like AADL. Generating monitors from such models could take into account the scope of errors and specifications. It would be an opportunity to study optimization strategies to use early error detection when the gain in terms of detection latency is high.

References

[1] Xenomai homepage. `https://www.xenomai.org`.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] E. Asarin, P. Caspi, and O. Maler. Timed regular expressions. *Journal of the ACM*, 49(2):172–206, 2002.

[4] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput*, 1(1):11–33, 2004.

[5] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In S. Arun-Kumar and N. Garg, editors, *FSTTCS*, volume 4337 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 2006.

[6] P. Bellini, R. Mattolini, and P. Nesi. Temporal logics for real-time system specification. *ACM Comput. Surv.*, 32(1):12–42, 2000.

[7] P. Chevochot and I. Puaut. Experimental evaluation of the fail-silent behavior of a distributed real-time run-time support built from cots components. In *International Conference on Dependable Systems and Networks, 2001. DSN 2001.*, pages 304–313, July 2001.

[8] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Software Eng*, 30(12):859–872, 2004.

[9] M. Diaz, G. Juanole, and J.-P. Courtiat. Observer—A concept for formal on-line validation of distributed systems. *IEEE Transactions on Software Engineering*, 20(12):900–913, Dec. 1994.

[10] D. Drusinsky. On-line monitoring of metric temporal logic with time-series constraints using alternating finit automata. *J.UCS: Journal of Universal Computer Science*, 12(5):482–498, 2006.

[11] R. B. França, J.-P. Bodeveix, M. Filali, J.-F. Rolland, D. Chemouil, and D. Thomas. The AADL behaviour annex - experiments and roadmap. In *ICECCS*, pages 377–382. IEEE Computer Society, 2007.

[12] K. Havelund and A. Goldberg. Verify your runs. In B. Meyer and J. Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 374–383. Springer, 2005.

[13] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer-Verlag, Berlin, 2002.

[14] M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky. Monitoring, Checking, and Steering of Real-Time Systems. In K. Havelund and G. Roşu, editors, *Runtime Verification*, volume 70 of *Electronic Notes in Theoretical Computer Science*, pages 96–112. Elsevier Science, July 26 2002.

[15] A. Mok and G. Liu. Early Detection of Timing Constraint Violation at Runtime. In *The 18th IEEE Real-Time Systems Symposium (RTSS '97)*, pages 176–186, Washington - Brussels - Tokyo, Dec. 1997. IEEE.

[16] T. Robert. RTRV -real-time run-time verification framework.

[17] T. Robert, J.-C. Fabre, and M. Roy. On-line monitoring of real time applications for early error detection. In *PRDC*, pages 24–31. IEEE Computer Society, 2008.

[18] T. Robert, M. Roy, and J.-C. Fabre. Evaluation of a real-time monitoring framework. In SIA, editor, *Proceedings of the 4th European Congres on Real Time Software*, page 8, 2008.

[19] U. Sammapun, I. Lee, and O. Sokolsky. RT-maC: Runtime monitoring and checking of quantitative and probabilistic properties. In *RTCSA*, pages 147–153, 2005.

[20] P. Traverse, I. Lacaze, and J. Souyris. Airbus fly-by-wire - a total approach to dependability. In *IFIP Congress Topical Sessions*, pages 191–212, 2004.

[21] S. Tripakis and S. Yovine. Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, 18(1):25–68, 2001.

[22] M. von der Beeck. A Comparison of Statecharts Variants. In *ProCoS: Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148, London, UK, 1994. Springer-Verlag.