Synthesis of Real-Time Observers from Past-Time Linear Temporal Logic and Timed Specification

Charles Lesire, Stéphanie Roussel, David Doose, Christophe Grand

Synthesis of Real-Time Observers from Past-Time Linear Temporal Logic and Timed Specification

Charles Lesire, Stéphanie Roussel, David Doose, Christophe Grand

***Abstract*— Fault-tolerant architectures are mandatory to ensure the robustness of autonomous robots performing missions in complex and uncertain environments. The first step of a fault-tolerant mechanism is the detection of a faulty behavior of the system. It is then important to provide tools to help robot developers specify relevant observers. It is moreover crucial to guarantee a correct implementation of the observers, i.e. that the observers do not miss data and do not trigger unsuitable recovery actions in case of false detection. In this paper, we propose a specification language for observers that uses Past-Time LTL to express complex formulas on data produced by software components, and timed constraints on the evaluations of these formulas. We moreover provide an implementation of this specification that guarantees a real-time evaluation of the observers. We briefly describe the observers we have specified for a patrolling mission, and we evaluate the performance of our approach compared to state of the art on a benchmark in which we detect errors on a laser range sensor.**

I. INTRODUCTION

Autonomous robots are there. Their capabilities to evolve in complex environments have been considerably improved in the last years. However, no autonomous robot is really used today, neither in public area nor in critical industrial applications. In the later, there is especially a need to prove that the behavior of the robot is safe whatever the foreseeable situations. These situations may be internal failures or environmental disturbances (e.g., dynamic obstacles, sensing conditions). To ensure such a robutsness faced with such disturbances, fault-tolerant control (FTC) architectures [1] have been developed. FTC first requires to detect the fault (Fault Detection), then to identify or isolate the faulty part of the architecture (Fault Isolation), and finally apply an action to tolerate the fault (Fault Recovery). In this paper, we focus on Fault Detection (FD). The use of *observers* is identified in [2] as the most common FD mechanism used in robotics. To monitor the mere dynamics of the system, observers are based on classical system state estimation, like Kalman filters [3]. To monitor plan execution, execution controllers like PLANEX [4] or PRS [5] compare the actual state of the system to expected action outcomes.

While these mechanisms use system and action models to detect failures, there is also a need to monitor specific situations. In addition to model-based observers, [6] has moreover identified two other FD mechanisms used in robotic systems: *timing checks*, that monitor component liveness through timers, and *reasonableness checks*, that monitor data values with respect to reasonnable bounds. While there is very few works focusing on these approaches, they are common ways of designing situations in which a reaction is neccessary. For instance, some works that implement safety rules (i.e., rules to apply in case of failures to enforce the system safety, [7], [8], [9]) use as a common example of monitoring the robot velocity with expressions similar to Listing 1.

```
1 if linear_speed > max_speed then stop
```

Listing 1. max speed safety rule

In this paper, we are interested in providing tools or patterns to design *observers*, that can be used as the condition part of such safety rules. Most of the previous works let the developer implement such observers in a programming language [6], [8], and then neither rely on a formal modelling, nor guarantee a correct behavior of observers.

DeRoS [9], a Domain Specific Language to define safety rules, allows to specify observers using a combination of value testing (value within intervals), propositional logic and timing constraints, as depicted in Listing 2.

```
1 entity drive_system {
2   max_speed_exceeded : linear_speed > max_speed
        for 2 sec;
3 }
```

Listing 2. DeRoS specification of the max speed rule (from [9])

This `entity` generates a ROS node that monitors the rule periodically. The monitor frequency must be empirically defined, and is set to 30Hz in [9]. As a consequence, DeRoS does not provide any guarantee that the execution rate of the monitor is consistent with the timing specification of the safety rules. This may lead to false detections; for instance, the ROS node may miss a data with `linear_speed < max_speed` within the two seconds.

While DeRoS is a good step towards providing a formal language to design observers, including specifying timing constraints, we claim that: (1) there is a need for more complex operators to design observers, (2) real-time execution of observers must be tightly related to the specified timing constraints. In this paper, we propose to specify observers using a combination of Past-Time LTL for low-level observers, and timed specification for high-level observers. This specification leads to the synthesis of observers attached to the emission of data, ensuring a real-time behavior of these observers.

Authors are with ONERA/DTIS, Université de Toulouse, F-31055 Toulouse, France <firstname>.<lastname>@onera.fr

The paper is organized as follows. Section II presents the background of software architectures for robotics, introduces some terminology, and presents Past-Time LTL. Section III presents the specification language for observers, as well as the timed semantics. The specification of observers for a ground robot performing a patrolling mission is described in Sec. IV. In Sec. V, we finally evaluate the performance of the proposed real-time observers compared to the state of the art on monitoring the failure of a laser sensor.

II. BACKGROUND

A. Software Architectures for Robotics

Component-based architectures have become a common paradigm for developping robot software [10], [11]. Middlewares and toolchains for developping software architectures distinguish (i) *components* (term used in Orocos [12], MAUVE [13], SmartMDSD [14], Genom [15], or *node* in ROS [16]) that support the execution of functions, and (ii) communication between components through data *ports* (term used in Orocos [12], MAUVE [13], Genom [15], or *communication objects* in SmartMDSD [14], or *topics* in ROS [16]). In this paper, we then adopt the terms *component* to talk about an executable entity of the system, and *port* to talk about data streams between components.

B. Past-Time Linear Temporal Logic

Past-Time Linear Temporal Logics (PastLTL) is a finite trace past temporal logic, first introduced in [17] and used in [18] to monitor Java program executions. Let $\mathtt{P}$ be a set of propositional variables, $\mathtt{p}$ an element of $\mathtt{P}$, and *op* a standard propositional operator in $\{\wedge, \vee, \rightarrow\}$. Then the formulas of PastLTL are defined as follows:

$$\begin{aligned} \varphi ::= \quad & true \mid false \mid \mathtt{p} \mid \neg\varphi \mid \varphi \; op \; \varphi \mid \\ & Y(\varphi) \mid O(\varphi) \mid H(\varphi) \mid \varphi \, \mathcal{S} \, \varphi \end{aligned} \tag{1}$$

$Y(\varphi)$ should be read "yesterday φ", $O(\varphi)$ "once φ", $H(\varphi)$ "historically φ", and $\varphi_1 \, \mathcal{S} \, \varphi_2$ "φ_1 since φ_2".

The semantics of this logic is based on the concept of *trace*. A trace t is a sequence of states $(s_0, \ldots, s_k)$. A state s_i is a valuation of variables of $\mathtt{P}$ at the ith execution step. For a trace $t = (s_0, \ldots, s_k)$ and an integer $i \in [0\,..\,k]$, $t[i]$ denotes the sequence $(s_0, \ldots, s_i)$. The semantics of PastLTL, defined in [18], is given by[1]:

$$t \vDash true \tag{2}$$
$$t \nvDash false \tag{3}$$
$$t \vDash \neg\varphi \text{ iff } t \nvDash \varphi \tag{4}$$
$$t \vDash \mathtt{p} \text{ iff } s_k(\mathtt{p}), \text{i.e. iff } \mathtt{p} \text{ is true in } t\text{'s final state} \tag{5}$$
$$t \vDash \varphi_1 \wedge \varphi_2 \text{ iff } (t \vDash \varphi_1) \text{ and } (t \vDash \varphi_2) \tag{6}$$
$$t \vDash \varphi_1 \vee \varphi_2 \text{ iff } (t \vDash \varphi_1) \text{ or } (t \vDash \varphi_2) \tag{7}$$
$$t \vDash \varphi_1 \rightarrow \varphi_2 \text{ iff } (t \vDash \varphi_1) \text{ implies } (t \vDash \varphi_2) \tag{8}$$
$$t \vDash Y(\varphi) \text{ iff } t[k-1] \vDash \varphi \text{ if } k > 1, t \vDash \varphi \text{ otherwise} \tag{9}$$
$$t \vDash O(\varphi) \text{ iff } \exists i \in [0\,..\,k], t[i] \vDash \varphi \tag{10}$$
$$t \vDash H(\varphi) \text{ iff } \forall i \in [0\,..\,k], t[i] \vDash \varphi \tag{11}$$
$$\begin{aligned} t \vDash \varphi_1 \, \mathcal{S} \, \varphi_2 \text{ iff } & \exists i \in [0\,..\,k], t[i] \vDash \varphi_2 \quad \wedge \\ & \forall j \in [i{+}1\,..\,k], t[j] \vDash \varphi_1 \end{aligned} \tag{12}$$

[1]Note that other definitions can be found in the literature for those operators.

The use of PastLTL is particularly relevant for fault detection in robotic architectures, compared to more classical logics such as LTL. First, during a run, we only know about the past and cannot predict the future in the general case. Thus, having modal operators for future is useless. Secondly, PastLTL is based on finite trace and has a recursive nature, making it possible to evaluate formulas in a very effective way. For instance, for a trace t of size $k > 0$, we have the following property: $t \vDash H(\varphi)$ iff $t \vDash \varphi$ and $t[k-1] \vDash H(\varphi)$. This property indicates that the value of the formula $H(\varphi)$ can be computed by just looking one step backwards. Similar properties hold for each temporal operator (see [18]), meaning that only the values of modal formulas at previous time step need to be known to compute their current value. This makes PastLTL much more efficient than LTL. Finally, PastLTL offers an expressivity that makes it easy to use by non-expert engineers.

C. Extension of PastLTL

In order to offer more expressivety for the temporal properties to evaluate, we introduce new operators to the PastLTL logic. For a PastLTL formula φ, we extend PastLTL with the following formula:

- for $n \in \mathbb{N}, O_n(\varphi)$; meaning that φ has been true at least once during the last n states;
- for $n \in \mathbb{N}, H_n(\varphi)$; meaning that φ has always been true during the last n states.

Let $t = (s_0, \ldots, s_k)$ be a trace with length k, and φ a PastLTL formula, the semantics of these new operators is defined the following way:

$$t \vDash O_n(\varphi) \text{ iff } \exists i \in [k{-}n+1\,..\,k], s_i \vDash \varphi \tag{13}$$
$$t \vDash H_n(\varphi) \text{ iff } \forall i \in [k{-}n+1\,..\,k], s_i \vDash \varphi \tag{14}$$

These operators are mandatory to specify formulas on the behavior of robots. Formulas over all the values from the beginning of the mission, which would be the case using standard H or O operators, are rarely meeningfull. It is indeed more common to detect faults by looking for values over a limited horizon.

III. SPECIFICATION OF REAL-TIME OBSERVERS

We split the specification of real-time observers into two layers. The first layer is closely related to data published by components, and allows to express temporal properties on these data. To do so, we use the PastLTL logic defined in the previous section. The second layer is dedicated to high level properties that combine several first-layer observers into a timed specification.

A. Temporal Observers on Data Ports

Temporal observers (atomic observers) are designed to evaluate formulas on values written on data ports. Each observer is attached to a given component and can contain several formula targetting ports of this component. These formulas are based on PastLTL and do not contain any timing constraint as they are linked to the component's execution

```
1  atomicObs ::= atomic-observer idObs on idComp {
       listFormula }
2  idFormula ::= idForm : formula ;
3  listFormula ::= idFormula | idFormula listFormula
4  formula ::= idPred ( listAtom )
5       | formula opProp formula
6       | op1 formula | op2 ( formula , int )
7  opProp ::= and | or | implies | iff | S
8  op1 ::= not | H | O
9  op2 ::= H | O
10 atom ::= portId | int | real
11 listAtom ::= atom | atom , listAtom
```

Listing 3. Grammar of atomic observers

period. More formally, an atomic observer is defined using the grammar presented in Listing 3.

The basic element of each formula is a *predicate* (line 4), i.e. a function implemented within the observer. Predicate arguments can either be the data written on component ports, or numbers (line 10). Formula can then be composed (lines 7 to 9) using classical operators (`and`, `or`, `implies`, `iff`, `not`), standard PastLTL operators (`H`, `O`, `S`) or extensions H_n and O_n (lines 6, 9).

Figure 4 shows the specification of an atomic observer *gpsObs* on component *gps*. Formula *signalDeg* expresses that the frame accuracy of the gps has been bad for the last 10 cycles of component *gps*. Predicate *goodAccuracy* is defined by the user, and checks that the field *accuracy* contained in the data written to port *frame* of component *gps* is under a threshold (5 meters in our implementation).

```
1 atomic-observer gpsObs on gps {
2   signalDeg : H(not goodAccuracy(gps.frame),10);
3 }
```

Listing 4. *gspObs* atomic observer specification

B. Timed Observers

Timed observers are designed to combine several atomic observers, and evaluate them with respect to timed constraints. The grammar of timed observers is given in Listing 5.

```
1 observer ::= observer idObs on listIdAtomic {
      listFormula }
2 listIdAtomic ::= idAtomic | idAtomic ,
      listIdAtomic
3 oneFormula ::= (id : formula ;)
4 listFormula ::= oneFormula | oneFormula
      listFormula
5 formula ::= tForm | not
      tForm | tForm opProp tForm
6 tForm ::= bForm | opTimed bForm within time
7 bForm ::= atom | not atom | bForm opProp bForm
8 opProp ::= and | or | implies | iff
9 opTimed ::= all | one
```

Listing 5. Grammar of timed observers

Timed constraints can be specified using the second part of *tForm* (line 6), as either formula `one` φ `within` δ, meaning that φ must have been true at least once within the last δ seconds, or `all` φ `within` δ, meaning that φ must have been true during the last δ seconds. φ can be defined using classical operators (line 7). Timed constraints can also be composed using classical operators (line 5). Atoms of the formulas correspond to the formulas of the atomic observers.

Examples of specification of timed observers are given in the next section that presents an application.

C. Observer Implementation

The evaluation of observers formula must be done in real-time, i.e., must guarantee that no information is lost from the components that are being observed. To do so, the observers must be implemented as close as possible to the port in which data is written. To ensure such a real-time evaluation of observers, we made a first implementation using the real-time middleware MAUVE [13]. In this middleware, we can attach some code to port writting, so that this code is executed each time a data is written on the port.

The evaluation of observers is then done at the port level, and is decomposed in three steps, described in the following paragraphs.

1) Evaluation of atomic formulas: In this step, we evaluate the formulas of each atomic observer each time a data is written on the ports of the observed component. This evaluation of formula φ at time t is represented with two variables: the current value ν of the formula (whether it is true or false), and the time τ of the last change of this value.

$$eval(\varphi, t) = (\nu, \tau) \tag{15}$$

$$\nu = t \vDash \varphi \tag{16}$$

$$\tau = \tau' \quad if \quad \nu = \nu' \quad else \quad t \tag{17}$$

with τ', ν' the previous evaluation of the formula.

2) Evaluation of timed observers: The second step consists in evaluating the timed observers. This evaluation is performed each time one of the atomic formulas is updated, and uses the evaluation of the atomics, i.e. their variables ν and τ. The evaluation of parts based on standard operators is straightforward. We detail only the evaluation specific to timed constraints.

The semantics of the `all` operator is given by eq. (18).

$$t \vDash \texttt{all}\ \varphi\ \texttt{within}\ \delta \quad \text{iff} \quad \nu \wedge (t - \tau \geq \delta) \tag{18}$$

where $(\nu, \tau) = eval(\varphi, t)$ is the evaluation of the atomic sub-formula φ at time t. This evaluation is true if the evaluation of φ is true at time t, and the last change happened more than δ ago.

The semantics of the `one` operator is given by eq. (19).

$$t \vDash \texttt{one}\ \varphi\ \texttt{within}\ \delta \quad \text{iff} \quad \nu \vee (t - \tau \leq \delta) \tag{19}$$

where $(\nu, \tau) = eval(\varphi, t)$ is the evaluation of the atomic sub-formula φ at time t. This evaluation is true if the evaluation of φ is true at time t, or the last change happened less than δ ago.

3) Memorisation of formula edges: The previous step evaluates the formulas of every observer, immediately when a data is written to the ports of the observed components. In the context of fault detection, the evaluation of these formulas must be used in order to implement reactions to failures. While fault recovery is not part of this paper, we consider that an external (i.e., in another component) entity monitors the observers to trigger recovery actions (an example is given in next section). This *monitor* executes at its own rate.

From the observer point of view, we want to guarantee that this monitor will not miss a fault detection, for instance if the monitor reads the observer evaluation just after its value became false. To that purpose, this last step memorizes the time of the last rising and falling edges of each observer formula.

$$eval(\phi, t) = (\tau_{rise}, \tau_{fall}) \tag{20}$$

$$\tau_{rise} = \begin{cases} t & \text{if } (t \vDash \phi) \ \wedge \ (t' \nvDash \phi) \\ \tau'_{rise} & \text{otherwise} \end{cases} \tag{21}$$

$$\tau_{fall} = \begin{cases} t & \text{if } (t \nvDash \phi) \ \wedge \ (t' \vDash \phi) \\ \tau'_{fall} & \text{otherwise} \end{cases} \tag{22}$$

where $(\tau'_{rise}, \tau'_{fall}) = eval(\phi, t')$ is the previous evaluation.

4) Illustration: Figure 1 shows an illustration of the evaluation of a timed observer specified as `all` φ `within` δ. The upper timeline shows the evolution of an evaluation of the atomic formula φ along time t. The timeline in the middle shows the value taken by formula `all`. This formula becomes true only when φ has been true for δ, and stays true while φ is true. The lower timeline shows the final evaluation of the formula, that is represented by the times of the last rising and falling edges of the formula.

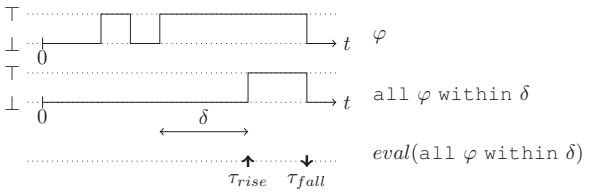

Fig. 1. Example timeline for the evaluation of `all` φ `within` δ.

IV. APPLICATION

A. Overview

The specification of real-time observers has been applied to the architecture of a ground robot that performs an autonomous mission (Fig. 2).

This mission consists in the survey of a critical site, by performing patrols in an area. The map of the area is known in advance, and the robot plans patrolling itineraries on this map. The robot is equiped with a GPS sensor to localize itself. It is also equiped with a laser range sensor, used to build a local map around the robot. This local map is used by a guidance algorithm to compute the trajectory towards the next point while avoiding obstacles.

Fig. 2. Robot for the patrolling mission

B. Observers

In this section, we present some timed observers we have specified to handle some failures occuring during the patrolling mission. The first type of observers are applied to sensors, and test whether some sensors provide erroneous data. We present only the observers on the laser sensor, but the similar observers are applied to other sensors of the architecture (GPS, IMU, ...). Then we present observers adressing the behavior of the robot, and that observe the guidance algorithm.

1) Laser sensor observers: Atomic observers on the *laser* component check the range values collected from the sensor and written in the *scan* port (see Listing 6).

```
1 atomic-observer laserObs on laser {
2   bad_scan : tooLowValues(laser.scan, 1);
3   fixed_scan : sameValuesThanPrev(laser.scan);
4 }
```

Listing 6. Specification of laser atomic observer

The *bad_scan* formula (line 2) tests whether the entire scan has values below 1 meter. The *fixed_scan* tests whether the scan is similar to the previous scan. Both tests are actually implemented in predicates.

The *laserError* timed observer defines two formulas on these atomic observers (see Listing 7). The *laser_data_error* (line 5) says that all scans within the last 2 seconds are bad (i.e., with values below 1 meter).

The second formula combines information from the laser sensor and the actual velocity of the robot. The *velocityObs* atomic observer defines formula *robot_moves* (line 2), that observes if the robot velocity has been non null for the last 100 values. Formula *laser_conn_error* (line 6) then observes if the robot has *moved* at least once within the last 10 seconds, while all the scans are identical. In that case, we suspect that the connection with the laser sensor is broken.

The last part of Listing 7 defines the monitoring, i.e. the rate at which we want to react to observers, and the actions to trigger if a formula is true. As mentionned in the previous section, monitoring is out-of-scope of this paper, and is just shown to illustrate how the real-time observers

```
1  atomic-observer velocityObs on robot {
2    robot_moves : H(not isNull(robot.velocity),
         100);
3  }
4  observer laserError on laserObs, velocityObs {
5    laser_data_error : all bad_scan within 2 sec;
6    laser_conn_error : (one robot_moves within 10
         sec) and (all fixed_scan within 10 sec);
7  }
8  monitor laserError at rate 1 {
9    if laser_data_error then stop;
10   if laser_conn_error then stop;
11 }
```

Listing 7. Specification of laser error timed observers

are used in a complete fault management architecture.

2) Guidance errors: Here we want to observe situations in which the guidance algorithm is trying to avoid an obstacle, but the guidance algorithm fails in finding a correct trajectory. Typically, we want to observe if the robot is just shaking left and right, failing in finding an exit from a complex obstacle. The observer on the *guidance* component (Listing 8) is decomposed into an atomic observer that evaluates if the guidance algorithm is currently avoiding an obstacle (formula *avoiding*, line 2), and if the robot seems to be static (line 3).

```
1  atomic-observer guidanceObs on guidance {
2    avoiding : isAvoiding(guidance.state);
3    move_less_than_1 : hasMovedLessThan(guidance.
         poses, 1);
4  }
5  observer stoppedWhileAvoiding on guidanceObs {
6    stoppedWhileAvoiding := all (move_less_than_1
         and avoiding) within 10 sec;
7  }
```

Listing 8. Specification of the guidance observers

The *stoppedWhileAvoiding* timed observer then combines these two formulas and observes if the robot was almost stopped while avoiding an obstacle during the last 10 seconds (line 6).

V. BENCHMARKING

A. Benchmarking protocol

In this section, we compare a real-time observer based on a mixed PastLTL and timed specification, and an observer following a behavior similar to what can be specified using DeRoS [9] or ROSRV [8]. In this experiment, the objective is to detect an error in the laser range sensor, by analysing the range values of the scan. The real-time observer is *laser_data_error*, given in Listings 6 and 7. The equivalent specification using DeRoS would be similar to that shown in Listing 9.

The DeRoS observer has been implemented in a single component that reads data in the laser. The evaluation of the formula has been implemented following the pseudo-code

```
1  entity laser {
2    laser_failure : (scan < 1) for 2 sec;
3  }
```

Listing 9. DeRoS specification of the laser error observer

given in [9, Fig. 6]. The period of the DeRoS component is one of the benchmark parameters, and is denoted T_D.

The objective of this experiment is to compare the detection behavior of the two architectures, in term of detection delay (time between the real failure and the detection by the observers), and false detection rate. In this aim, we modified the laser component in order to (1) generate randomly some data such that $scan < 1$, and (2) put the laser in a failure mode so that it will systematically procude values such that $scan < 1$. In the experiment, we made vary: (1) the probability to generate a scan with $scan < 1$, (2) the time at which the laser is definitely failing, (3) the period T_L of the laser component, (4) the period T_D of the DeRoS-like observer, (5) the period T_M of the monitor that reads the timed observer value.

B. Evaluation when the DeRoS frequency varies

In a first setting, we fixed the laser period T_L to 100 ms (10Hz), the monitor period T_M to 1 second (as specified in the monitor of Listing 7), and made vary the probability to produce bad scans from 1% to 50%. The DeRoS period T_D is varying from 1 second (similar to our monitor) to $1/30$ (the default period used in [9]). We ran a hundred of executions for each value of T_D. We then measured the false detections and the detection delay of the DeRoS observer and of our observer (noted PLTL-Monitor in the following figures).

Figure 3 shows the detection delays according to the DeRoS frequency. First, we can notice that the delay at 30Hz is similar to the results given in [9, Tab. 1], then validating the implementation of this observer. When DeRoS runs faster than the laser (frequency greater than 10Hz), we can notice that the delay is very short. However, when the DeRoS frequency is low, the detection delay inscreases, up to more than 1.7 seconds whereas the observer period is 1 second. In comparison, our monitor observes the fault with no more than 1 second of delay, and its mean delay is about 0.4 seconds whereas its period is 1 second.

Figure 4 shows the false detections of the observer. These false detections come from the fact the observer missed some data. The timed observer implementation ensures that the evaluation of observer formulas are executed exactly when the data is written. As a consequence, it is impossible to miss data, leading to no false detection, as noticed during the executions. The DeRoS observer has made some false detection, especially at low frequencies, i.e. when it is more likely to miss data.

C. Evaluation when the laser frequency varies

In this second setting, we fixed the DeRoS frequency and the monitor frequency to 30Hz. The laser frequency varies from 10Hz to 100Hz. We made a hundred of runs

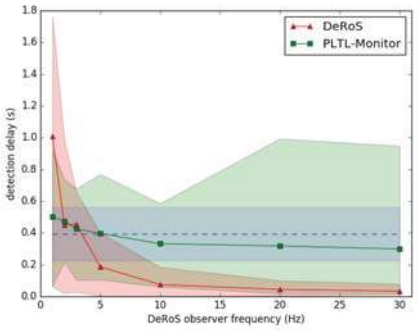

Fig. 3. Evolution of the detection delay w.r.t. DeRoS frequency. The blue dashed line shows the mean value of the PLTL-Monitor delay over all the executions. The blue envelop displays the standard deviation around this value. Green and red envelops respectivelty display the minimal and maximal delays for PLTL-Monitor and DeRoS.

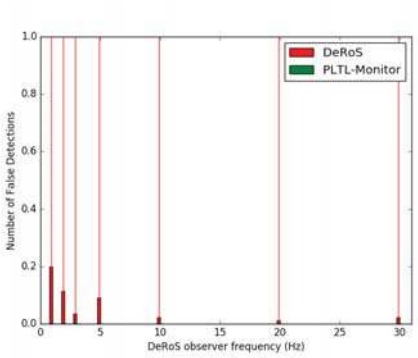

Fig. 4. Evolution of the number of false detections w.r.t. DeRoS frequency. The vertical bar represents the average number of false detection per run. The vertical line represents the maximal number of false detection per run.

for each laser frequency. Figure 5 shows the delay of the two observers, and Fig. 6 the false detections.

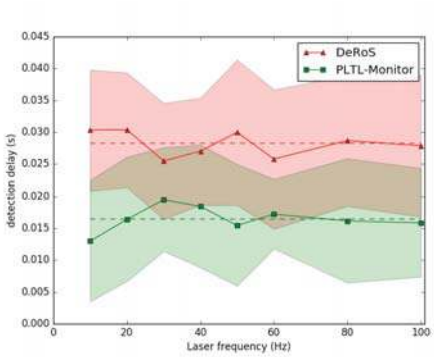

Fig. 5. Evolution of the detection delay w.r.t. Laser frequency. The dashed lines show the mean values of the delays. The envelops display the standard deviation.

As soon as the laser runs faster than the observers, the DeRoS observer detects false alarms, whereas it is never the case for the monitor. Detection delays of DeRoS are correct, but sometimes greater than the DeRoS period. Our monitor has better detection delays and never overtakes its period.

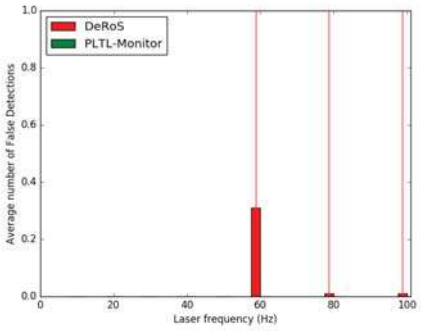

Fig. 6. Evolution of the average number of false detections w.r.t. Laser frequency. The vertical bar represents the average number of false detection per run. The vertical line represents the maximal number of false detection per run.

VI. CONCLUSION

In this paper, we have proposed a specification language for observers. *Atomic observers* observe the data written on component ports, and can be expressed using PastLTL. PastLTL allows particularly to define complex behaviors by reasoning over the last n values written on the port. As far as we know, there is no other specification language for fault detection that uses PastLTL in robotic architectures. Atomic observers are then combined into *timed observers*. We have proposed timed formulas with the `all ... within` and `one ... within` constructs, that allow to test if a sub-formula was true everytime (or once) within a given duration. We have provided a sound semantics for these operators, and a real-time implementation that evaluates the observers on each data writting, hence ensuring that no data is missed. The evaluation of the observers is then provided (for a monitoring component) as the times of the last rising and falling edges of the observer formula. We have shown a part of the specification of observers for a robot performing a patrolling mission. We have also compared the behavior of our observer with respect to a state-of-the-art observer for detecting laser errors. The conclusion of this benchmark is that we guarantee that no data is missed, and then we cannot produce false detections. Note that this property is guaranteed *by construction*: observers are systematically evaluated each time a new value is produced by any component. The implementation of the observers is real-time, and as a consequence, the reaction delay to faults is only dependent on the rate of the monitoring component. Compared to the DeRoS observer, which behavior is very dependent on the relative rates of DeRoS and the monitored components, it means that we can guarantee a good reaction time with appropriate rates, without having to run the monitor faster than the components we observe.

ACKNOWLEDGMENT

This work has been partly funded by the ONERA research program for drone safety PHYDIAS granted by the French Civil Aviation Authority (DGAC).

REFERENCES

[1] E. Khalastchi and M. Kalech, "On Fault Detection and Diagnosis in Robotic Systems," *ACM Computing Surveys*, vol. 51, no. 1, 2018.

[2] O. Pettersson, "Execution monitoring in robotics: A survey," *Robotics and Autonomous Systems*, vol. 53, no. 2, pp. 73 – 88, 2005.

[3] R. Washington, "On-board real-time state and fault identification for rovers," in *International Conference on Robotics and Automation (ICRA)*, San Francisco, CA, USA, 2000.

[4] M. Fichtner, A. Grossmann, and M. Thielscher, "Intelligent Execution Monitoring in Dynamic Environments," *Fundamenta Informaticae*, vol. 57, no. 2-4, pp. 371–392, 2003.

[5] F. Ingrand, R. Chatila, R. Alami, and F. Robert, "PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots," in *International Conference on Robotics and Automation (ICRA)*, Minneapolis, MN, USA, 1996.

[6] D. Crestani, K. Godary-Dejean, and L. Lapierre, "Enhancing fault tolerance of autonomous mobile robots," *Robotics and Autonomous Systems*, vol. 68, pp. 140 – 155, 2015.

[7] M. Machin, J. Guiochet, H. Waeselynck, J.-P. Blanquart, M. Roy, and L. Masson, "SMOF - A Safety MOnitoring Framework for Autonomous Systems," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 48, no. 5, pp. 702–715, 2018.

[8] J. Huang, C. Erdogan, Y. Zhang, B. Moore, Q. Luo, A. Sundaresan, and G. Rosu, "ROSRV: Runtime Verification for Robots," in *International Conference on Runtime Verification (RV)*, Toronto, Canada, 2014.

[9] S. Adam, M. Larsen, K. Jensen, and U. P. Schultz, "Rule-based Dynamic Safety Monitoring for Mobile Robots," *Journal of Software Engineering for Robotics (JOSER)*, vol. 1, no. 7, pp. 120–141, 2016.

[10] D. Brugali and P. Scandurra, "Component-based robotic engineering (part I)," *IEEE Robotics and Automation Magazine*, vol. 16, no. 4, pp. 84–96, 2009.

[11] D. Brugali and A. Shakhimardanov, "Component-based robotic engineering (part II)," *IEEE Robotics and Automation Magazine*, vol. 17, no. 1, pp. 100–112, 2010.

[12] P. Soetens and H. Bruyninckx, "Realtime Hybrid Task-Based Control for Robots and Machine Tools," in *International Conference on Robotics and Automation (ICRA)*, Barcelona, Spain, 2005.

[13] D. Doose, C. Grand, and C. Lesire, "MAUVE Runtime: a component-based middleware to reconfigure software architectures in real-time," *Journal on Software Engineering for Robotics (JOSER)*, vol. 8, no. 1, pp. 128–140, 2017.

[14] D. Stampfer, A. Lotz, M. Lutz, and C. Schlegel, "The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software," *Journal of Software Engineering for Robotics (JOSER)*, vol. 7, no. 1, pp. 3–19, 2016.

[15] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. F. Ingrand, "GenoM3: Building middleware-independent robotic components," in *International Conference on Robotics and Automation (ICRA)*, Anchorage, AK, USA, 2010.

[16] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, Kobe, Japan, 2009.

[17] E. Emerson, "Temporal and Modal Logic," in *Handbook of Theoretical Computer Science, Volume B:Formal Models and Semantics*, J. van Leeuwen, Ed. Elsevier, 1990, pp. 995–1072.

[18] K. Havelund and G. Rosu, "Synthesizing monitors for safety properties," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Grenoble, France, 2002.