A hierarchical fault tolerant architecture for an autonomous robot

Anthony Favier, Antonin Messioux, Jérémie Guiochet, Jean-Charles Fabre, Charles Lesire

A hierarchical fault tolerant architecture for an autonomous robot

Anthony Favier, Antonin Messioux
LAAS-CNRS
INPT ENSEEIHT, Univ. of Toulouse
Toulouse, France
{anthony.favier, antonin.messioux} @etu.enseeiht.fr

Jérémie Guiochet, Jean-Charles Fabre
LAAS-CNRS
UPS, INPT, Univ. of Toulouse
Toulouse, France
{jeremie.guiochet, jean-charles.fabre}@laas.fr

Charles Lesire
ONERA/DTIS
Univ. of Toulouse
Toulouse, France
charles.lesire@onera.fr

Abstract—**This paper presents a generic approach to specify a fault tolerant robot controller, and its implementation and validation with ROS and Gazebo. The main idea is to implement a fault tolerance strategy using a fault tree and an ordered set of recovery modules. A fault injection campaign has been carried out with a mobile autonomous robot for airport inspection using simulation with Gazebo and ROS. This successful experiment implements a safety-first strategy.**

Index Terms—**Fault tolerance, Autonomous mobile robot, Simulation, Fault injection, ROS, Gazebo**

I. Introduction

Increasing complexity and autonomy of tasks performed by mobile robots, requires to deploy more and more techniques to analyse and guarantee confidence in such systems [1]. Among all dependability techniques, fault tolerance, defined as a technique to avoid service failures in the presence of faults [2], has been widely used in robots and autonomous systems. Basically, a fault tolerance mechanism (FTM) is composed of a detection module (DM) and a recovery module (RM). This definition is applicable to basic redundant architectures (e.g., redundant sensors), to more complex fault tolerant architectures (e.g. at the localization function level [3], or at the system level [4]), and it is also widely used in component-based robot controllers [5]. However, such approaches fail to address an issue which is the coordination between several fault tolerance mechanisms included in the robot controller at different layers. Indeed, when integrating FTM, the recovery actions may be triggered in a concurrent way, which can lead to unwanted states of the system. For instance, a FTM could be in charge of respawning some active control components, which are actually required by another FTM to stop the robot. To cope with such situations, our approach is to define a partial order between FTM based on a fault tree analysis, and another one for RM impact on safety. Based on these basic elements, we propose a basic safety-first strategy implemented in a FTM Manager. We apply this approach to a simulated mobile autonomous robot checking lights on airport runways in a ROS/Gazebo environment.

This paper first introduces the case study, the Osmosis project [6], in section III. Then we present an overview of the approach in section IV and its application in section V. Lessons learnt are provided in the conclusion.

II. Related work

Fault tolerance is at the very heart of robotic and autonomous systems community interest. Indeed, in the SPARC roadmoap [7], the proposed “dependability levels” actually do not address dependability as a whole, but defines levels of autonomy of the robot regarding fault tolerance (e.g. how the system is able to autonomously manage, even predict, and recover from faults). If most of the work in robotic fault tolerance was developed to cover hardware failures (e.g., for industrial manipulators in [8], [9]), the concept has been extended to the complete architecture of robot and autonomous systems, from the functional layer to the decisional layer. At the functional level, fault tolerance in robotics has been experimented for actuators, sensors or perception software errors. For instance, [5] propose to develop dedicated monitors for each software component for mobile robots, which is also done in [10]. In these papers, timing or reasonableness checks are performed for hardware and software modules with recovery mechanisms impacting the decisional level (for instance, reduce the autonomy level of the robot). In [11], the faults from environment and sensors are detected and recovered in the layer responsible for action sequencing and execution in autonomous robot. In case of error detection, the corresponding function is executed in a fall-back mode. Other functions are chosen to deliver the same task or the level of autonomy is reduced by switching to a tele-operated mode. In this case, the decisional layer is disconnected. At decisional level, some works on “execution monitoring” [12]–[14] focus on planner capacity to cover errors, like environmental hazards [15] or faults in the hardware layer [16]. There are actually few works really focusing on fault tolerance at the decisional level as in [17]–[19], and one popular approach is to use active safety monitors in independent layers as fault tolerance mechanisms [4], [20]–[22].

All these works are actually focusing on detection and recovery mechanisms dedicated to specific layers in autonomous robot architecture, but do not address the issue of synchronisation between fault tolerance mechanisms. Such approach can be found in distributed systems [23] and networks [24], but they are globally more focusing on synchronisation protocols rather than on the consistency between the recovery mecha-

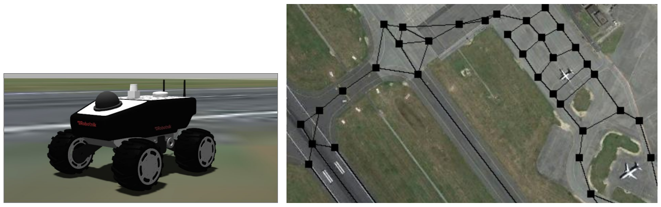

Fig. 1. The simulated Summit Xl robot and the airport with Gazebo

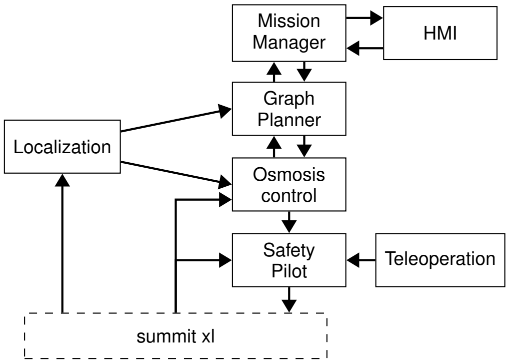

Fig. 2. Osmosis controller ROS-nodes architecture

nisms that can be triggered by the fault tolerance mechanisms. Another important point is that they usually do not deal with detection and recovery mechanisms that can be implemented at different levels of abstraction in the architecture. In an autonomous architecture, fault tolerance mechanisms have to deal with several levels of abstraction, from functional layer to the decisional layer. Hence, managing consistency between these mechanisms while dealing with different levels of abstraction is still an open issue in autonomous systems.

III. OSMOSIS Case study

The OSMOSIS experiment [6] is inspired from the SafeAM experiment as part of the CPSE Labs project [25]. In this experiment, we consider an autonomous mobile robot able to navigate on the airport like in Figure 1 in order to reach runways, and then proceed to light inspections by driving along the light row and grabbing light intensity data using a specific sensor. For this experiment, we use the simulator Gazebo, and the simulated robot Summit XL from Robotnik [26] presented in Figure 1 running under ROS.

The proposed control architecture is shown on Figure 2, where only nodes and data flow are presented. The complete code and its simulator is available at [6]. The mission is basically a description of which runway the robot has to check (e.g., runway A and B). Two basic control modes are applied : “taxi” when the robot has to reach the starting point of a runway, the robot is able to avoid obstacles using a potential field algorithm, and “light verification” when the robot is on a runway and checking the lights, in this case the robot just stop in case of an obstacle presence, and wait until the obstacle is removed. The mission manager is in charge of transmitting the objective point and the associated control mode (taxi or light-verif), the graph planner computes the trajectory using a graph of the airport, and send intermediary points to the Osmosis Controller which is charge of computing the speed command according to the current intermediary point to reach. The safety pilot is in charge of passing commands, unless an obstacle is too close (then the system switches to controlled stop, i.e. a 0 speed command is sent), or a preemptive request for teleoperation (with a joystick). The *summit xl* layer provides the simulated sensors (odometry, hokuyo laser, IMU, GPS) and actuators (in this case only a linear and angular speed control node). In the current version we use the absolute localization published by Gazebo instead of fusion between IMU, Odometry and GPS as it is done on the real platform.

IV. Approach overview

The basic steps of the deployment of our approach are presented below and detailed hereafter:

1) Analyse the fault propagation chain with a fault tree;
2) Identify fault tolerance mechanisms (FTM) mitigating the effect of the unwanted events in the fault tree;
3) Export the resulting FTM tree (FTMTree) and identify the recovery modules partial order (RMGraph);
4) Choose the recovery strategy that will be implemented in the FTM Manager.

Among all the risk analysis techniques, the fault tree analysis is still one of the most used in industry and research. It basically consists of a top-down analysis, starting from the unwanted event (also called “top event”, e.g., “unwanted stop of the robot”), and identifying the combination of unwanted events that may lead to this top event. This combination is usually modeled with OR and AND gates. Note that few concepts are required to start using it, whereas a high expertise is required to develop effective trees.

The second step is to identify all fault tolerance mechanisms that could mitigate the effect of the unwanted events that are identified in the fault tree. A fault tree analysis is usually done to find minimal cut sets in order to identify weaknesses of the design, or to estimate the probability of the top event. In our case, we do not investigate such a use of the fault tree analysis, nor a complete risk analysis, but only focus on how a fault tree could be a tool for developing fault tolerance mechanisms. As it will be the case in next section, not all unwanted events of a fault tree could be mitigated by a fault tolerance mechanism, but this is out of the scope of this paper. A fault tolerance mechanism (FTM) can be represented as a barrier, as it is done in the left part of bow-tie diagrams [27], to mitigate the propagation of the unwanted events. We chose to define a FTM by its two main modules: a detection module (DM) and a recovery module (RM). A FTM is thus defined as $FTMx = (DMy, RMz)$. A DM could be a timing or value error detection, with basic comparisons or more complex detection with active components such as watchdogs. For RM we stick to the three basic ones : forward, backward, and masking [2]. Forward recovery is usually a mission level action (i.e. emergency stop, or controlled stop), whereas backward recovery and masking are used at component level.

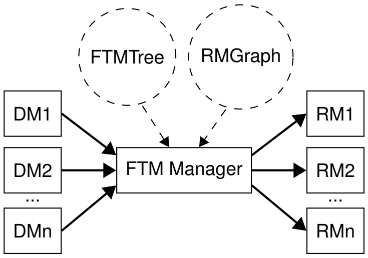

Fig. 3. FTM Manager inputs and outputs

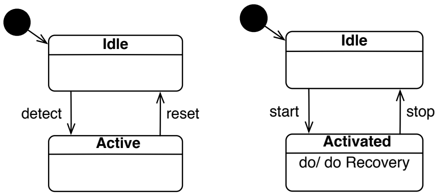

Fig. 4. DM and RM state machines

The FTM tree in the third step is actually a direct translation of the previous fault tree (where FTMs act as barriers), where the relation order is the distance in the cause-consequence chain to the top event. For instance, in the tree of Figure 6, $FTM1$ (details are given in the next section) is the root mechanism, whereas $FTM5$ and $FTM7$ are leaves of the tree.

For the RMGraph, the partial order relation between RM depends on the objective of the overall fault tolerance strategy. For instance, let us consider only two RMs, "Emergency Stop" (a complete shutdown of the power supply leading to block the wheels in our case study) and "Controlled stop" (sending a zero value for the speed command, but keep the control loop active). In a mission-first strategy, the "Emergency Stop" (which drastically impact the mission fulfilment) might be ordered differently if the strategy would be safety-first. In such a safety-first strategy, the relation order would be "is safer than", expressing the fact that the safety of the reached state after activation of the "Emergency stop", would be safer than "controlled stop". "Safer" would be estimated based on potential severity and likelihood of a harm induced by the robot in case of failure or adverse situation.

The last step is the specification of the strategy that should be implemented, particularly when two (or more) FTM are concurrently active. In a safety-first strategy, which has been implemented in our case study, we manage to execute the FTM according to the FTMTree and the RMGraph as it is presented in Figure 3. The role of the FTM Manager is to centralize all detections and prioritize recovery modules. Our implementation is based on state machines to handle such situations (see Figure 4). Basically, DMs have two states, $Idle$ and $Active$, and RMs have also two states $Idle$ and $Activated$. An FTM is said to be $Active$ when its DM is $Active$, and $Activated$ when its RM is $Activated$. The FTM Manager is responsible for controlling the situation where one FTM is $Activated$ and another FTM becomes $Active$.

TABLE I
DETECTION MODULES DESCRIPTION

DM1 Prohibited Area	A redundant device is able to detect if the robot reach a prohibited area
DM2 Command not updated	Freshness of command is assessed
DM3 Wrong command value	Out-of bounds error detection
DM4 Control node crash	A watchdog is associated to the main control node
DM5 Non-control nodes crash	A watchdog is associated to all nodes
DM6 Localization not updated	Freshness of localization is assessed
DM7 Localization node crash	A watchdog is associated to the localization node

TABLE II
RECOVERY MODULES DESCRIPTION

RM1 Emergency stop	A command to engage brakes at low level is simulated
RM2 Controlled stop	Linear and angular speed are assigned a 0 value.
RM3 Respawn Control nodes	Osmosis Control, Mission Manager, Graph Planner are respawn
RM4 Respawn Non-control nodes	Respawn non-control nodes (Teleoperation, Localization, HMI)
RM5 Switch to teleoperation	The safety pilot is activated to only execute commands from the joystick

Our objective is to deploy an architecture of the FTM Manager allowing the developers to specify several strategies with no modification of the implemented DMs and RMs. In the next section, we chose to implement a safety-first strategy, which always leads to activate only one RM, but as we are using concurrent objects programmed with state machines in the DM and RM abstract classes, the strategy may be easily changed to allow an activation of simultaneous recovery mechanisms, and each state machine could also be extended using inheritance.

V. APPLICATION TO OSMOSIS

The experiment has been carried out on only one fault tree, i.e. for one top event. We chose to study the top event "moving inside a prohibited area" among the identified hazards. A simplified fault tree is presented Figure 5. This fault tree was analyzed to identify FTMs that could mitigate the propagation of unwanted events that induce the previous top event.

The FTMs are presented Figure 6, where $FTMx = (DMy, RMz)$ is used to associate a detection module (DMy) to a recovery module (RMz). The complete list of DMs and RMs for this experiment are given in Table I and II. Not all unwanted events could be mitigated in this fault tree with FTMs, for this paper we do not address this issue, but on how existing fault tolerance mechanisms could coexist.

For this proof-of-concept paper, we chose to implement recovery mechanisms of several types. RM1, RM2 and RM7 are forward recovery (the system goes in a new state). In table II, "Control nodes" are those that contain mission data

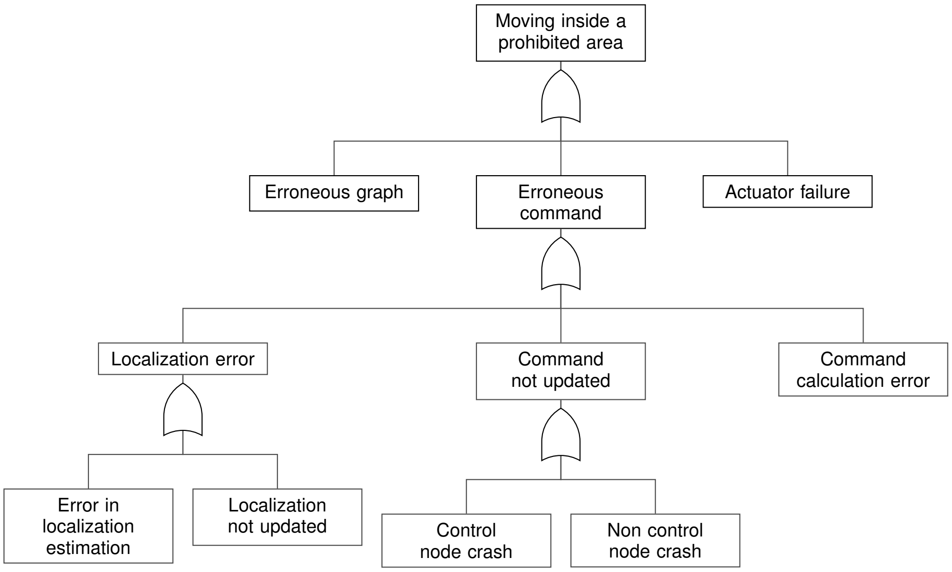

Fig. 5. Fault tree for the top event "moving inside a prohibited area"

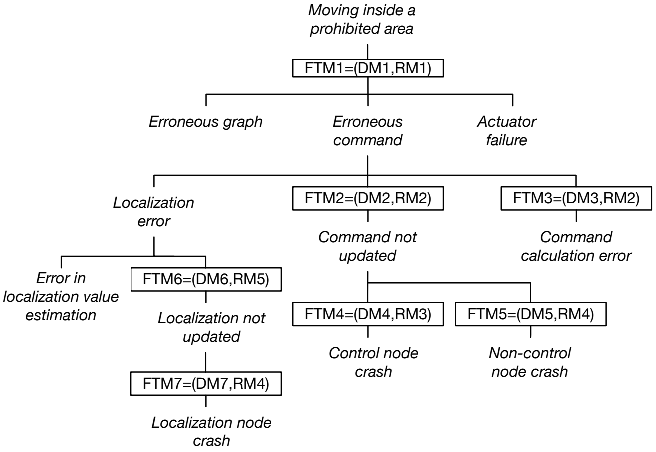

Fig. 6. FTM as barriers added to the fault tree of the top event "Moving inside a prohibited area"

(i.e. Mission Manager, Graph Planner, Osmosis Control in Figure 2). "Non-control nodes" are Localization, HMI and Teleoperation (because they are memoryless, they can easily be restarted). The Safety Pilot node is not considered as a functional node, because it is part of the recovery mechanism implementation. In short, we should consider that this node cannot crash otherwise fault tolerance is not guaranteed. RM4 is a simple restart of control nodes. RM3 could be qualified as a backward recovery technique since "Control nodes" must be restarted with previously checkpointed states.

A. Strategy algorithm

The proposed strategy algorithm is provided in Figure 7 and the complete code and UML diagrams of the implementation are available at [6].

We implement a safety-first algorithm that always select the highest FTM (in the FTMTree) in case of concurrent activation, but also able to manage concurrent non-ordered FTMs using the RMGraph. In this study, we chose to use the following partial order sets (poset) for the $FTMTree =< FTM, \geq >$ where $FTMx \geq FTMy$ means "FTMx is closer to the unwanted event in the cause-consequence chain than FTMy". In the same way, we define $RMGraph =< RM, \geq >$ where $RMx \geq RMy$ means "RMx is safer than RMy", i.e. the probability and severity of potential harm after RMx engaged, is less than for RMy (such a risk matrix is usually done by the experts). The resulting poset is represented with a Hasse diagram in Figure 8.

In the FTMTree we will use the $dominance$ function usually used in control flow, defined by: an element $FTMx$ dominates an element $FTMy$ if every path from the root node to $FTMy$ go through $FTMx$. For instance, in Figure 8(a), $FTM1 = dom(FTM1, FTM2, FTM4)$. If no dominant is found, the function returns 0. The function lowest common dominant (low_com_dom) corresponds to the closest parent node of a set of nodes in the tree. For instance, the lowest common dominant of FTM4 and FTM5 is FTM2, i.e. $low\_com\_dom(FTM4, FTM5) = FTM2$. This lowest common dominance of a subset A of the set P, could be formally defined by : $a = low\_com\_dom(A)$ iff $\nexists x \in P, (x = dom(A, x) \wedge a = dom(x))$.

The $dominance$ function is also used for the RMGraph, such that $dom(RM5, RM3, RM4) = RM5$ and $dom(RM3, RM4) = 0$ (there is no dominance between RM3 and RM4).

The proposed algorithm shown in Figure 7 is following basic steps in case of simultaneous active FTMs. Let FTM* be the set of active FTMs. If only one FTMx is active, then the corresponding RM is activated (noted RM(FTMx)

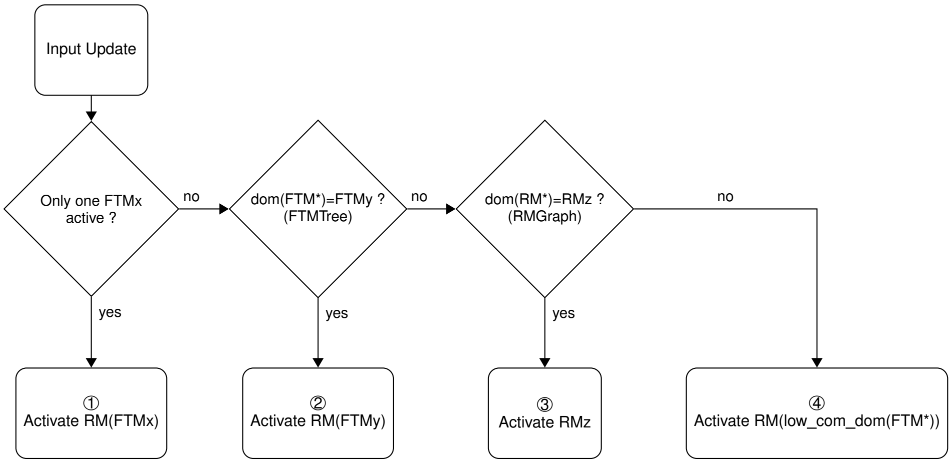

Fig. 7. Safety-first strategy algorithm

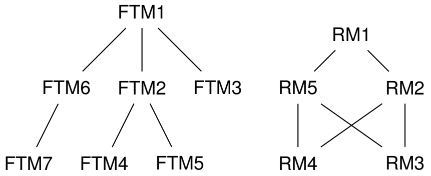

Fig. 8. (a) Fault Tolerant Mechanism Tree (FTMTree) (b) Recovery Module poset (RMGraph)

TABLE III
RESULTS FOR THE FAULT INJECTION CAMPAIGN

Active FTMs (DMs)	**Corresponding RMs**	**Algorithm action block**	**Executed RM**
FTM7	RM4	①	RM4
FTM1 and FTM6	RM1, RM5	②	RM1
FTM3 and FTM4	RM2, RM3	③	RM2
FTM2, FTM6 and FTM5	RM2, RM5, RM4	④	RM1

in block ① of Figure 7). In case of several active FTMs, if one FTM dominates the active FTMs (i.e., $dom()$ returns $FTMy$), then it is executed ($RM(FTMy)$ in ②). If there is no dominant FTM (option marked as "no"), the algorithm checks if there is a dominant active RM. If there is one, this leads to execute the dominant recovery mechanism in ③. Finally, if there are no dominant in the RMGraph ($dom()$ returns 0), then the algorithm activates the RM of the lowest common dominant FTM in the FTMTree ④. Table III presents some examples covering the four cases explained above. In case of simultaneous FTM, with several dominant RMs, we chose to not execute all these RMs, but to only execute the lowest common dominant. This choice has been made by considering that the impact of simultaneous activation of RMs might not guarantee to put the system in a safe state. It is of course a conservative approach which can be extended to other strategy.

This conservative strategy can be extended to a less conservative one, for mission-first strategy for instance (not presented in this paper). For the same reason as just above, to ensure only one RM is activated at a time, we stop every RMs dominated by a RMx just before activating it. Yet, we have to be careful when stopping RMs. We have to be sure the RM to stop has finished its action or finished putting the system into a degraded mode. Interrupting a RM makes us unable to guarantee in which state the system is, and thus its safety.

Figure 9 is a class diagram showing the implementation of the FTM Manager and corresponding components. Using inheritance for DM and RM, let us define generic state machines, which led to easily modify DM and RM behavior. We also provide a generic implementation for the FTMTree and RMGraph, which make them easily extendable to new ordered sets.

B. Verification

The early validation of the approach and the verification of the FTM implementation has been carried out through a set of experiments, including a fault injection campaign. All these experiments were conducted on the simulated Summit Xl robot with Gazebo. The complete opensource code is available online at [6].

The verification of the system implies a careful analysis of its behavior in the presence of faults. The faults considered here are those identified so far and belonging to the fault tree given in Figure 8. It is clear that the verification steps depends on the considered FTM; new faults, new FTM, imply new verification experiments. A robot like this is subject to the evolution of the environment and maybe weather conditions that may lead the robot designer to consider other faults and by the way new mechanisms. The evolution of the conditions may be difficult to anticipate at initial design time. This means that the heath status of the robot and surroundings conditions should be monitored and FTM updated accordingly. Adaptation of FTM is out of the scope of this paper, but this issue has already been investigated in companion work, such as in [28].

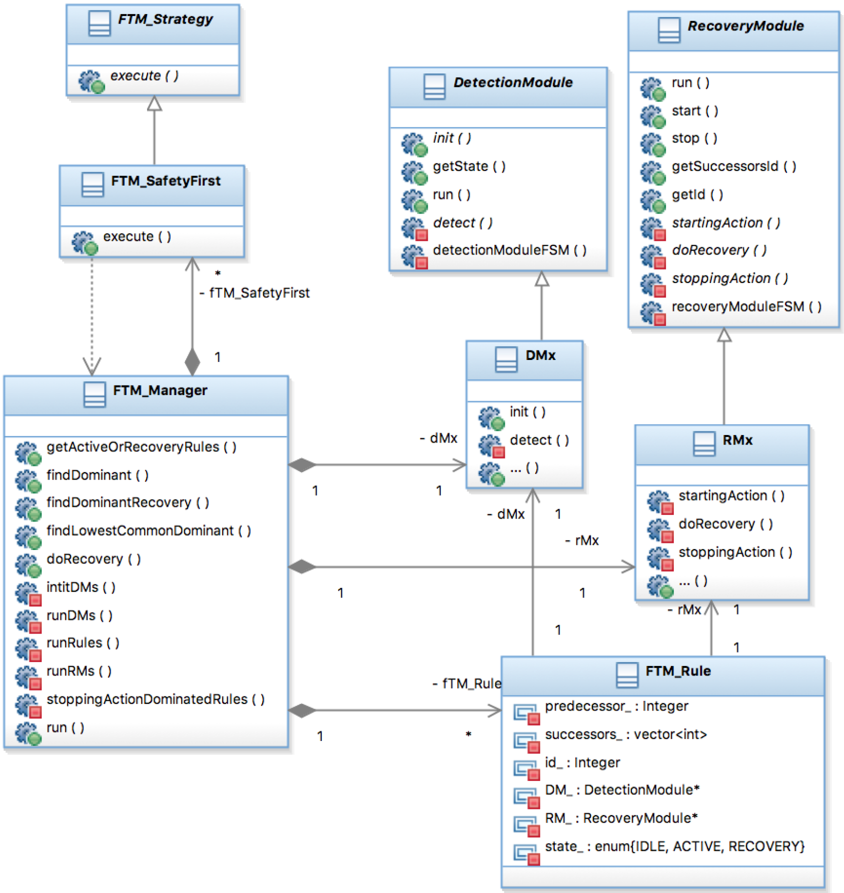

Fig. 9. Class diagram of the fault tolerance mechanism

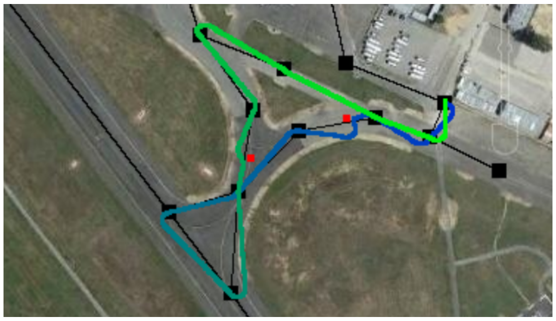

Fig. 10. Golden Run of Osmosis with two obstacles (Start:blue, End:green)

The objective here is not to quantify error detection and recovery coverage, but to analyse the behaviour of the system in the presence of unwanted events, namely faults. An experiment is organized in two steps: a golden run is done first to observe the nominal behavior of the robot with no faults (see Figure 10); then a number of runs are done with unwanted events injected for which FTMs should be normally activated. Two golden runs were considered with and without obstacles which led to two types of faulty runs, with and without obstacles as well.

First, regarding the effect of the implemented RMs one by one, the results obtained are given in Table IV and V. The first table is the test case of Osmosis without any FTM implemented.

A run is considered $successful$ if the mission requested has been fulfilled by the robot. That is to say, if the robot has gone through every points specified in the mission file in the right order and with the right mode : “taxi” or “light verification”. A run is considered $safe$ if the robot never went into a prohibited area and if it didn’t collide with any obstacles.

Table IV shows how fault injection affects the mission and safety without any FTM implemented. Each injected fault put the robot in an unsafe state which means it would be very dangerous to deploy the system like this. For example, if the robot’s localization is somehow not updated the command computed stays the same. Thus, the robot can go in a prohibited area or collide with an obstacle, which is obviously not safe. Also, in most of cases the mission is a failure because after fault injection the robot became non-operational as above. Detailed effects of each injected fault are described in Table IV.

In order to keep the robot in a safe state we added FTMs and the results can be observed in Table V. We can easily see that every run is now safe no matter which fault was injected. Once again, the implemented FTMs are safety-first oriented so their

TABLE IV
OSMOSIS BENCHMARK WITHOUT THE FTM

Injected Fault	Obstacles (Yes - No - Both)	Mission result	Safety result	Description
None	Both	Success	Safe	Golden Run
Command out of bounds (DM3)	No	Success	Unsafe	The erroneous command can move the robot into a prohibited area and the real system's hardware could have been damaged. But once the temporary fault is gone the robot finishes its mission
Command out of bounds (DM3)	Yes	Fail	Unsafe	If it happens during an avoidance manoeuvre a collision can occur. If the robot is too close to an obstacle for an avoidance manoeuvre it stops, so after a collision due to the injected fault there are high chances that the robot stops (mission failed).
Localization not updated (DM6)	Both	Fail	Unsafe	Without localization updates the last one received is kept. Thus, the command is the same : collision, prohibited area
Command not updated (DM2)	Both	Fail	Unsafe	The last command is kept so same consequences as above
Non-control node crash (DM5)	Both	Fail	Unsafe	HMI or Teleoperation node crash doesn't have much effect but a Localization node crash leads again to the same consequences as localization not updated
Control node crash(DM4)	Both	Fail	Unsafe	Effects differ according to which node actually crashed. If MissionManager or GraphPlanner crashes the controller will simply have no new orders to follow so robot will reach the point it was targetting then eventually stop. In the case of OsmosisControl no more command will be computed so it's equivalent to command not updated.

TABLE V
OSMOSIS EXPERIMENT WITH FAULT TOLERANCE MECHANISM

Injected Fault	Obstacles (Yes - No - Both)	Mission result	Safety result	Description
None	Both	Success	Safe	Golden Run
Command out of bounds (DM3)	Both	Fail	Safe	Robot stopped in a safe area and before any collision
Localization not updated (DM6)	Both	Fail	Safe	Robot stopped and switched to teleoperation before any collision
Command not updated (DM2)	Both	Fail	Safe	Robot stopped in a safe area and before any collision
Non-control node crash (DM5)	Both	Success	Safe	All missing non-control node are restarted instantly
Control node crash (DM4)	Both	Fail	Safe	Node restarted but mission progress is lost so the robot either reaches its current goal or stops

activation is all about ensuring the safety and not fulfill the mission. That's why, now, when a "command out of bounds" occurs the mission always fails because the robot is ordered to stop, but since it wasn't into a prohibited area before the fault it is safe. However, now when a non-control node crashes it is restarted, in particular the localization node. So after being restarted the robot's localization is again updated and the mission can be completed. Thus, the FTM made the run both safe and successful. Moreover, for control nodes, backward recovery should be done but it hasn't been implemented yet. These nodes should often save their state in order to be restarted in the same one and continue the mission. Currently, the nodes are restarted anyway despite losing the mission progression. Since they start in an idle state where the robot is stopped, the mission is failure but the run is safe. Also, to avoid collision when an information isn't updated we calculated the watchdogs values according to the robot maximum velocity and the obstacle detection distance. So the FTM is activated before a collision can occur. Therefore, FTMs significantly improved the safety of the system. The last step is to see how the system reacts with concurrent FTMs.

As a proof of concept, we did not focus on the results in terms of detection latency, or overhead induced by FTMs (which should actually be done in a real implementation with a dedicated layer for fault tolerance mechanisms), but we focus on the logical part of robot behavior in case of concurrent activated FTMs, to observe if the safety-first strategy is practically efficient. Test cases given in Table III enable the four action blocks of the algorithm to be executed, see Figure 7 (denoted noted as ①②③④). "Active FTM" are FTMs where their DM is active, and "Executed RM" is the resulting RM run by the FTM Manager. This table confirms the logical behaviour of the robot and thus the efficiency of our strategy.

VI. CONCLUSION

The proposed FTM framework relies mainly on two trees (FTMTree and RMGraph), and an FTM strategy using these trees.

We described in this paper a first implementation of this framework as a proof of concepts. The combination of detection mechanisms and recovery mechanisms in a flexible way offers the opportunity to adjust the objectives and extend the framework easily. In our case, ”safety first” was the objective.

The set of mechanisms defined to this aim were verified by fault injection, considering a representative set of faults that may impair the behavior of the robot and violate safety.

A first important result is the genericity of the implementation, where any extension of new FTM, DM or RM is easily done with few code modification. Moreover, our approach allows to integrate more complex FTM rules, using logical gates (e.g., $FTM = (DM1 \wedge DM3, RM2 \wedge RM4)$). All nodes (including DM, RM and FTM Manager) are based on generic state machines, which also makes it more easily extendable to complex behaviors. However, the experiments also reveals open issues requiring more investigation. The first one is the choice of the relation order that could be extended to more complex definitions. For instance, the "safer" relation could also be mixed with an assessment on the autonomy impact, or any other mission performance property. We also do not investigate how the FTM Manager communicates with the Mission Manager, in order to replan, repair a plan, or cancel the mission, which may have an impact on the FTM Strategy. Finally, some additional features must be added to the framework, such as a stable storage feature enabling checkpointing to increase recovery mechanisms performances. However, in this paper we did not study the overhead cost induced by the whole FTM (including RM, DM and FTManager algorithm), as we only focus first in a proof of concept of the FT mechanism, implemented in a ROS architecture. This issue is of paramount importance for a complete deployment, and we plan to use a dedicated layer to implement the FTM. A preliminary study was performed using ROS for the functional part, and the framework MAUVE [29] for the fault tolerant mechanism, and it will be deployed for future experiments.

References

[1] J. Guiochet, M. Machin, and H. Waeselynck, "Safety-critical advanced robots: A survey," *Robotics and Autonomous Systems*, vol. 94, pp. 43–52, Aug. 2017.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[3] Z. Zhao, J. Wang, J. Cao, W. Gao, and Q. Ren, "A fault-tolerant architecture for mobile robot localization," in *2019 IEEE 15th International Conference on Control and Automation (ICCA)*, July 2019, pp. 584–589.

[4] 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, May 2018.

[5] D. Crestani, K. Godary-Dejean, and L. Lapierre, "Enhancing fault tolerance of autonomous mobile robots," in *Journal of Robotics and Autonomous Systems*. Elsevier, 2015.

[6] C. Lesire, F. Ingrand, and J. Guiochet, "Osmosis : Open-source material for safety assessment of intelligent systems," https://osmosis.gitlab.io, Accessed: 2020-03.

[7] SPARC, "Robotics 2020 multi-annual roadmap for robotics in europe," Horizon 2020 Call ICT-2017 (ICT-25, ICT-27 & ICT-28), Release B 02/12/2016, 2016.

[8] M. L. Visinsky, I. D. Walker, and J. R. Cavallaro, "Layered dynamic fault detection and tolerance for robots," in *[1993] Proceedings IEEE International Conference on Robotics and Automation*, May 1993, pp. 180–187 vol.2.

[9] M. L. Visinsky, J. R. Cavallaro, and I. D. Walker, "A dynamic fault tolerance framework for remote robots," *IEEE Transactions on Robotics and Automation*, vol. 11, no. 4, pp. 477–490, Aug 1995.

[10] S. Zaman, G. Steinbauer, J. Maurer, P. Lepej, and S. Uran, "An integrated model-based diagnosis and repair architecture for ROS-based robot systems," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, 2013, pp. 482–489.

[11] B. Durand, K. Godary-Dejean, L. Lapierre, R. Passama, and D. Crestani, "Fault tolerance enhancement using autonomy adaptation for autonomous mobile robots," in *International Conference on Control and Fault Tolerant Systems (SysTol)*, 2010, pp. 24–29.

[12] A. Bouguerra, L. Karlsson, and A. Saffiotti, "Monitoring the execution of robot plans using semantic knowledge," *Robotics and Autonomous Systems*, vol. 56, no. 11, pp. 942 – 954, 2008.

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

[14] J. P. Mendoza, M. Veloso, and R. Simmons, "Mobile robot fault detection based on redundant information statistics," in *Workshop at IROS'12 on "Safety in human-robot coexistence and interaction: How can standardization and research benefit from each other?", Vilamoura, Portugal*, 2012.

[15] P. Ertle, D. Gamrad, H. Voos, and D. Soffker, "Action planning for autonomous systems with respect to safety aspects," in *IEEE International Conference on Systems Man and Cybernetics (SMC)*, 2010, pp. 2465–2472.

[16] S. Gspandl, S. Podesser, M. Reip, G. Steinbauer, and M. Wolfram, "A dependable perception-decision-execution cycle for autonomous robots." in *International Conference on Robotics and Automation (ICRA)*, 2012, pp. 2992–2998.

[17] B. Lussier, M. Gallien, J. Guiochet, F. Ingrand, M.-O. Killijian, and D. Powell, "Planning with diversified models for fault-tolerant robots," in *Proc. of The International Conference on Automated Planning and Scheduling (ICAPS07), Providence, Rhode Island, USA*, 2007, pp. 216–223.

[18] ——, "Fault tolerant planning for critical robots," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN07), Edinburgh, UK*, 2007.

[19] I. R. Chen, F. B. Bastani, and T. W. Tsao, "On the Reliability of AI Planning Software in Real-Time Applications," *IEEE Transactions on Knowledge and Data Engineering*, vol. 7, no. 1, pp. 14–25, February 1995.

[20] J. Guiochet, D. Powell, É. Baudin, and J.-P. Blanquart, "Online Safety Monitoring Using Safety Modes," in *Workshop on Technical Challenges for Dependable Robots in Human Environments DRHE08, PASADENA, United States*, May 2008, pp. 1–13.

[21] M. Machin, F. Dufossé, J. Blanquart, J. Guiochet, D. Powell, and H. Waeselynck, "Specifying safety monitors for autonomous systems using model-checking," in *The 33rd International Conference on Computer Safety, Reliability and Security (SAFECOMP)*. Springer International Publishing, 2014, pp. 262–277.

[22] M. Machin, F. Dufossé, J. Guiochet, D. Powell, M. Roy, and H. Waeselynck, "Model-checking and game theory for synthesis of safety rules," in *16th IEEE International Symposium on High Assurance Systems Engineering, HASE 2015, Daytona Beach, FL, USA*, 2015, pp. 36–43.

[23] D. Powell, "Distributed fault tolerance: lessons from delta-4," *IEEE Micro*, vol. 14, no. 1, pp. 36–47, Feb 1994.

[24] R. Guerraoui and A. Schiper, "Fault-tolerance by replication in distributed systems," in *Reliable Software Technologies — Ada-Europe '96*, A. Strohmeier, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 38–57.

[25] CPSE-Labs, "CPSE Labs - Cyber-Physical Systems Engineering Labs," http://www.cpse-labs.eu/, Accessed on 2020-03.

[26] Robotnik, "Robotnik summit xl robot specification," https://www.robotnik.eu/mobile-robots/summit-xl/, Accessed: 2020-03.

[27] A. de Ruijter and F. Guldenmund, "The bowtie method: A review," *Safety Science*, vol. 88, pp. 211 – 218, 2016.

[28] M. Lauer, M. Amy, J.-C. Fabre, M. Roy, W. Excoffon, and M. Stoicescu, "Resilient computing on ros using adaptive fault tolerance," *Journal of Software: Evolution and Process*, vol. 30, no. 3, p. e1917, 2018.

[29] D. Doose, C. Grand, and C. Lesire, "Mauve runtime: A component-based middleware to reconfigure software architectures in real-time," in *2017 First IEEE International Conference on Robotic Computing (IRC)*, 2017, pp. 208–211.