THÈSE POUR OBTENIR LE GRADE DE DOCTEUR
DE L'UNIVERSITÉ DE MONTPELLIER

En Systèmes Automatiques et Microélectroniques (SYAM)

École doctorale : Information, Structure, Systèmes (I2S)

Unité de recherche : LIRMM (UMR 5506)

Multi-level fault tolerance for autonomous robots - Application to an underwater robot

Présentée par Adrien HEREAU
Le 17 juin 2022

Sous la direction de Didier CRESTANI
et de Jérémie GUIOCHET

Devant le jury composé de

Didier CRESTANI, Professeur, LIRMM, Université de Montpellier	Directeur de thèse
Jérémie GUIOCHET, Professeur, LAAS-CNRS, Université de Toulouse	Co-Directeur de thèse
Karen GODARY-DEJEAN, Maître de Conférences, LIRMM, Université de Montpellier	Encadrante de thèse
Charles LESIRE, Docteur, ONERA, Université de Toulouse	Rapporteur
Roland LENAIN, Directeur de recherche, IRSTEA, Université Clermont Auvergne	Rapporteur
Jacques MALENFANT, Professeur, Sorbonne Université	Examinateur
Olivier PARODI, Docteur, Naval Group	Invité

UNIVERSITÉ DE MONTPELLIER

Abstract

Multi-level fault tolerance for autonomous robots - Application to an underwater robot

by Adrien HEREAU

English version

The use of mobile robots is growing in many fields of application. These robots can be used in harsh environments where access is dangerous or even impossible for humans. Unfortunately, we have to face many undesired events during these missions. These can significantly impact the mission in progress or the safety of the robot and its environment or the control loop of the robot.

Dependability techniques provide proven solutions in many areas, and in particular fault tolerance which allows to detect and treat these situations at runtime. However, the proposed approaches lack adaptability and genericity, generally not covering the mission, safety and control loop aspects in a joint way. In addition, the lack of compatibility analysis between the recovery mechanisms implemented on different levels of the robot can have consequences on the safety and performance of the robot.

This work describes a framework to limit the occurrence or propagation of faults in mobile robots in order to minimize the undesirable consequences both at the mission and safety levels. We propose an original multi-level approach dealing with 3 categories of undesired events. The first 2 categories concern 'high-level' events: violations of mission and safety constraints. The last category concerns 'low-level' events: faults, errors and failures occurring in the robot.

In a preliminary phase, our approach requires the construction of fault trees for each of the previously identified categories. The study which is then carried out leads to a reinforcement of the reliability of the system in case of faults by implementing local recoveries which act without disrupting the course of the mission. During the running phase, the undesired events are checked thanks to a model extracted from the fault trees. The system can then use local redundancy or recovery to limit the propagation of an undesired event or to suppress it. If an event is detected without a local fallback solution, a global recovery aiming at changing task is then proposed by a mission manager. The contributions are a complete framework to implement all the steps of this analysis, as well as an algorithm to recover from the events identified in the fault trees.

We tested our framework on an underwater robot capable of performing autonomous missions. The tests were carried out within the framework of observation missions of the underwater ecosystem called transect. They demonstrate that our approach allows to limit the occurrence of catastrophic events while allowing an adequate management of the mission.

French version

L'utilisation de robots mobiles se développe dans de nombreux domaines d'application. Ces robots peuvent être utilisés dans des environnements difficiles où l'accès est dangereux voire impossible pour l'homme. Malheureusement, on doit faire faire face à de nombreux évènements indésirables lors de ses missions. Ceux-ci peuvent impacter significativement la mission en cours ou la sécurité du robot et de son environnement, ou la boucle de contrôle du robot.

Les techniques de la sûreté de fonctionnement apportent des solutions éprouvées dans de nombreux domaines, et notamment la tolérance aux fautes qui permet de détecter et traiter ces situations à l'exécution. Cependant, les approches proposées manquent d'adaptabilité et de généricité, ne couvrant généralement pas les aspects missions, sécurité et boucle de contrôle de façon conjointe. De plus, le manque d'analyse de la compatibilité entre les mécanismes de recouvrement implémentés sur différents niveaux de l'architecture du robot peut avoir des conséquences sur la sécurité et les performances du robot.

Ce travail décrit un framework visant à limiter l'apparition de fautes ou la propagation de celles-ci dans les robots mobiles de façon à en minimiser les conséquences indésirables tant au niveau mission que sécurité. Nous proposons une approche multi-niveaux originale traitant 3 catégories d'évènements indésirables. Les 2 premières catégories concernent les évènements 'haut-niveau': les violations de contraintes de type mission et de type sécurité. La dernière catégorie concerne les évènements 'bas-niveau': les fautes, erreurs et défaillances se produisant dans le robot.

Dans une phase préliminaire, notre approche nécessite l'élaboration d'arbres de défaillances pour chacune des catégories précédemment identifiées. L'étude qui est ensuite effectuée conduit à un renforcement de la fiabilité du système en cas de fautes en implémentant des recouvrements dits locaux s'ils agissent sans perturber le déroulement de la mission. Pendant la phase de fonctionnement normal, les évènements indésirables sont vérifiés grâce à un modèle extrait de l'arbre de défaillances. Le système peut alors utiliser une redondance ou un recouvrement local afin de limiter la propagation d'un évènement indésirable ou de le supprimer. En cas de détection d'un évènement sans solution de repli local, un recouvrement global visant à changer de tâche est alors proposé par un gestionnaire de mission. Les contributions sont un framework complet permettant de mettre en place toutes les étapes de cette analyse, ainsi qu'un algorithme de recouvrement des évènements identifiés dans les arbres de défaillances.

Nous avons testé notre framework sur un robot sous-marin capable d'effectuer des missions en autonomie. Les tests ont été effectués dans le cadre de missions d'observation de l'écosystème sous-marin appelées transect. Ils démontrent que notre approche permet de limiter l'apparition d'évènements catastrophiques tout en autorisant une gestion adéquate de la mission.

Acknowledgements

Je tiens tout d'abord à remercier mes directeurs et encadrants de thèse: mon directeur de thèse, monsieur Didier Crestani, pour les échanges scientifiques fructueux que nous avons eu ensemble; mon co-directeur de thèse, monsieur Jérémie Guiochet, pour son apport éclairé autour de la thématique de la sûreté de fonctionnement; mon encadrante de thèse, Madame Karan Godary-Dejean, pour l'encadrement de ma thèse et pour le suivi de la partie expérimentale.

Je remercie également les membres du jury, en particulier messieurs Charles Lesire et Roland Lenain d'avoir accepté d'être rapporteurs de cette thèse. Merci à monsieur Jacques Malenfant pour avoir accepté d'évaluer cette thèse et à monsieur Olivier Parodi en qualité d'invité.

Ce travail de thèse n'aurait pu se faire sans une équipe forte pour accompagner mes différents travaux. A ce titre, je souhaite remercier les membres de l'équipe Explore et associés du LIRMM: messieurs Benoit Ropars et Robin Passama pour leurs apports techniques; messieurs Lionel Lapierre, René Zapata et Tho Dang pour leurs apports scientifiques autour de la robotique sous-marine; mais aussi messieurs Thomas Claverie, José-Luis Vilchis Medina, Quentin Massone, Philippe Lambert, Yohan Breux, Rodolfo Villalobos Martinez, Ferréol Gagey, Benjamin Navarro, Eric Godard, Charles Villard, Clément Rebut et Clément Robert pour leur implication et leur aide.

Sur le plan personnel, je remercie ma famille ainsi que mes amis de Loire-Atlantique et de Montpellier pour leur soutien au cours de ces années de thèse.

Pour finir, je remercie également l'université de Montpellier et la région Occitanie pour le financement de cette thèse, ainsi que le labex Numev pour la participation au financement de la partie expérimentale.

Contents

Abstract iii

Acknowledgements v

1 Introduction 1

2 State of the art 5
2.1 Architectures of mobile autonomous robots . . . 5
2.1.1 Generic mobile robot architectures . . . 6
2.1.2 Control architectures of mobile robots . . . 6
2.2 Dependability concepts . . . 8
2.2.1 Dependability attributes and threats . . . 8
2.2.2 Fault prevention . . . 10
2.2.3 Fault forecasting . . . 12
2.2.4 Fault removal . . . 17
2.3 Fault tolerance . . . 22
2.3.1 Fault tolerance and FDIR approach . . . 22
2.3.2 Error detection and fault diagnosis mechanisms . . . 23
2.3.3 Recovery mechanisms . . . 26
2.3.4 Error handling by passive redundancy . . . 28
2.3.5 Evaluating the efficiency of a fault tolerant system . . . 29
2.4 Autonomy and fault tolerance in robotics . . . 29
2.4.1 Managing safety with fault tolerance . . . 30
2.4.2 Managing mission execution with fault tolerance . . . 32
2.4.3 Managing control loop with fault tolerance . . . 33
2.5 Conclusion . . . 35

3 Proposal of a multi-level fault tolerant framework 37
3.1 Fault tolerant framework overview . . . 37
3.2 Identification and classification of undesired events . . . 39
3.2.1 Guidelines for building the mission fault tree . . . 40
3.2.2 Guidelines for building the safety fault tree . . . 42
3.2.3 Guidelines for building the control loop fault tree . . . 42
3.3 Identification of system's weaknesses . . . 45
3.4 Implementation and execution of event detection and inference . . . 45
3.4.1 Implementation and execution of undesired event detection . . . 46
3.4.2 Execution of event inference . . . 48
3.5 Implementation and execution of recoveries . . . 51
3.5.1 Implementation and execution of local recoveries . . . 51
3.5.2 Implementation and execution of decisional recoveries . . . 57
3.6 Conclusion . . . 59

4 Case study: Experimental platform and context **61**
4.1 A introduction to recent marine ecosystem monitoring robots 61
4.1.1 A classification of underwater robots 62
4.1.2 Notations for underwater robotics 62
4.1.3 Underwater robots for marine ecosystem assessment 64
4.2 The experimental platform of our case study 65
4.2.1 Hardware architecture of the experimental platform 65
4.2.2 Software architecture of the experimental platform 68
4.3 Mission description: the underwater transect 74
4.3.1 An approach to observe the marine environment 74
4.3.2 Overview of the robotic transect mission 75
4.3.3 Definition of the mission constraints and mission properties . . 77
4.4 Fault tree, detection and recovery implementation 82
4.4.1 Implementation of the safety fault tree 83
4.4.2 Implementation of the mission FT 87
4.4.3 Implementation of the control loop fault tree 89
4.5 Conclusion 94

5 Fault tolerance framework: experimental validation **95**
5.1 Validation of our framework: test objectives 95
5.1.1 System under test and mission context 96
5.1.2 Context of the tests in the field 96
5.1.3 Context of the software in the Loop 97
5.2 Calculating an estimation of the recovery coverage 98
5.3 Unit testing of the FTM functionalities 99
5.3.1 UEs and recoveries of interest 100
5.3.2 Results of the unit testing 103
5.4 Functional and non-regression testing of the system 109
5.4.1 Test cases: mission and environmental context 109
5.4.2 Results of the functional and non-regression testing 111
5.5 Conclusion and discussion 116
5.5.1 Respect to the test objectives 116
5.5.2 Limitations and threats to validity 117

6 Conclusion **121**
6.1 Background and problematic 121
6.2 Synthesis of the main results 121
6.3 Limitations 124
6.4 Improvement 125
6.5 Future works 126

A Representation of the fault trees **127**

B Estimating the physical parameters **139**
B.1 Dynamic equation 139
B.2 Mass matrix 139
B.3 Coriolis and centripetal effects 140
B.4 Hydrodynamic matrix 140
B.5 Hydrostatic vector 141

C Implementation of the extended Kalman filter **143**
C.1 A generic implementation of the extended Kalman filter 143
C.2 Implementation of the extended Kalman filter on REMI 144

D Unit testing measurements **147**
D.1 T1: Handling water inside enclosure 147
D.2 T2: Handling robot in the critical collision zone of the seabed 148
D.3 T3: Handling robot in the warning collision zone 2 of the seabed 148
D.4 T4: Handling robot in the warning collision zones of the seabed 150
D.5 T5: Handling collision with an obstacle 151
D.6 T6: Handling robot not on the transect line 151
D.7 T7: Handling poor localization during MP2 152
D.8 T8: Handling MP1 lasts too long 153
D.9 T9: Handling camera coverage too low 153
D.10 T10: Handling not enough energy to perform MP1 154
D.11 T11: Handling freeze of echosounder driver 155
D.12 T12: Handling motor2 out of order 155
D.12.1 T12 in simulation 155
D.12.2 T12 in the field 156
D.13 T13: Handling motor2 and motor 6 out of order 157
D.14 T14: Handling no communication with PC 157
D.15 T15: Handling not enough energy to operate 158

Bibliography **159**

List of Figures

2.1 A generic control loop structure for a mobile robot . . . 6
2.2 A deliberative architecture . . . 7
2.3 Fault occurrence, error propagation and system failure . . . 9
2.4 The fault classes . . . 10
2.5 Fault tree main components . . . 14
2.6 Verification categories . . . 18
2.7 An automated oracle test . . . 19
2.8 Fault tolerance concepts in dependability . . . 22
2.9 FDIR approach . . . 23
2.10 A classification of FDD approaches . . . 24
2.11 Fault tolerant control of an autonomous mobile robot . . . 30

3.1 Overview of the fault tolerant framework . . . 38
3.2 A standard mission FT . . . 41
3.3 An example of a system defined by module and its associated fault tree 44
3.4 An example of fault tree . . . 48
3.5 A fault tree before and after adding the failure of the local recoveries . 52
3.6 Lists of incompatible LRs from a matrix of compatibility . . . 53
3.7 An example of group priority for 3 groups of compatible LRs . . . 57
3.8 The framework UML class diagram (implemented in C++) . . . 58

4.1 The 3 main classes of underwater robots . . . 62
4.2 Reference frames and motion notations . . . 63
4.3 Ulysse underwater robot . . . 65
4.4 REMI robot . . . 66
4.5 Software architecture of the control loop of REMI . . . 68
4.6 Principle of the coupling between USBL and surface GPS . . . 69
4.7 Calculation of the desired speed for a straight line motion between 2 points . . . 71
4.8 The user interface QGroundControl during a simulated mission . . . 73
4.9 The control loop architecture with the FTM . . . 73
4.10 A transect performed by divers . . . 75
4.11 An UUV performing a transect . . . 75
4.12 Use case diagram of the system . . . 76
4.13 A transect mission performed by our underwater robot . . . 77
4.14 The danger zones near the seabed . . . 84
4.15 Safety fault tree . . . 86
4.16 Mission fault tree . . . 88
4.17 Detection of a faulty motor . . . 89
4.18 Control loop fault tree - Actuators module . . . 91

5.1 Testing our robotic system . . . 96
5.2 The validation test at the Crès lake . . . 97

5.3 Implementation of the simulator within the architecture 98
5.4 The inference process during T15 104
5.5 The deduction process during T8 105
5.6 Detection, inference and recovery during T12 106
5.7 Rate of property violation with and without the FTM 112
5.8 Extract of the safety fault tree around SAFE_4 115

A.1 Control loop fault tree 128
A.2 Comm-PC module sub-fault tree 129
A.3 DVL module sub-fault tree 130
A.4 Echosounder module sub-fault tree 131
A.5 Concatenation of all FTs 132
A.6 Mission fault tree 133
A.7 NAV module sub-fault tree 134
A.8 Safety fault tree 135
A.9 Task module sub-fault tree 136
A.10 UPB Power sub-fault tree 137

D.1 Depth and heave command of the robot during T1 147
D.2 Depth of the robot during T2 148
D.3 Context of test during T3 and T4 149
D.4 Depth, altitude and reaction of the robot during T3 149
D.5 Depth, altitude and reaction of the robot during T4 150
D.6 Acceleration measurement during T5 151
D.7 Distance between the robot and TL during T6 151
D.8 Estimated standard deviation of the x position during 2 transects . . . 152
D.9 Surge command and distance to SP during T8 153
D.10 Coverage of the filmed zone during T9. The purple area is the theoretical zone to film whereas the green area is the zone effectively filmed 154
D.11 Difference of intensity between the estimated and the measured current in PWS1 156
D.12 Errors and standard deviations of depth, roll and pitch in simulation during T12 156
D.13 Errors and standard deviations of depth, roll and pitch in the field during T12 157

List of Tables

4.1 SNAME notations for marine vehicles . . . 63
4.2 REMI sensors and their measured variables . . . 66
4.3 Hardware features of the REMI robot . . . 67
4.4 Mission constraints related to MP0 . . . 78
4.5 Thresholds and constants used in MP0 . . . 78
4.6 Mission constraints related to MP1 . . . 78
4.7 Thresholds and constants used in MP1 . . . 79
4.8 Mission constraints related to MP2 . . . 80
4.9 Thresholds and constants used in MP2 . . . 81
4.10 Mission constraints related to MP3 . . . 82
4.11 Thresholds and constants used in MP3 . . . 82

5.1 A static analysis of the FTM . . . 99
5.2 Recoveries of interest for the validation process . . . 100
5.3 Safety and mission related events of interest for the validation process 101
5.4 Control loop related events of interest for the validation process . . . 102
5.5 Synthesis of the unit testing for safety and mission related events . . . 107
5.6 Synthesis of the unit testing for control loop related events . . . 108
5.7 Synthesis of the functional test cases . . . 110
5.8 Property violation rates over all runs . . . 112
5.9 Software activity with and without FTM . . . 116

List of Abbreviations

FT	**F**ault **T**ree
FTM	**F**ault **T**olerance **M**anager
SUT	**S**ystem **U**nder **T**est
MCS	**M**inimal **C**ut **S**ets
UUV	**U**nmanned **U**nderwater **V**ehicule
AUV	**A**utonomous **U**nderwater **V**ehicule
HROV	**H**ybrid **R**emotely **O**perated **V**ehicule
UE	**U**ndesired **E**vent
MP	**M**ission **P**hase
DR	**D**ecisional **R**ecovery
LR	**L**ocal **R**ecovery
USBL	**U**ltra **S**hort **B**ase **L**ine
DVL	**D**oppler **V**elocity **L**og
EKF	**E**xtended **K**alman **F**ilter

List of Symbols

$\{b\}$	Body frame attached to the robot
$\{n\}$	NED (North-East-Down) local frame
$\nu = [u, v, w, p, q, r]^T$	Robot speeds in $\{b\}$
$\tau = [X, Y, Z, K, M, N]^T$	Robot forces and moments in $\{b\}$
$\eta_e = [x, y, z, \phi, \theta, \psi]^T$	Pose of the robot (euler) in $\{n\}$
$\eta_q = [x, y, z, q_w, q_x, q_y, q_z]^T$	Pose of the robot (quaternion) in $\{n\}$
$\chi = [\nu, \eta_q]^T$	State of the robot (quaternion)

Chapter 1

Introduction

The full autonomy of mobile systems is characterized by the decision-making capacity to evolve in their environment. Mobile systems use this autonomy to perform actions considered relevant in time and space without external help from a human operator. They have gained in autonomy during these last decades, mainly thanks to the increase of the computing capacities of the controllers and perception functions. These systems can replace or assist humans in difficult, dangerous, repetitive or unrewarding tasks.

Mobile robotics is divided into several broad categories depending on the environment: space robotics, air robotics, ground robotics, marine robotics and underwater robotics. Satellites and autonomous aerial drones are already widely used in research for operational applications. Satellites ensure continuous attitude and altitude maintenance operations with event management in autonomy or by an operator from earth. Autonomous unmanned aerial vehicles are also widely investigated, for example to prospect strategic locations, to transport packages or to work in swarms with other autonomous robots. Ground-based autonomous robots are used in open environment for exploration, surveillance, scientific analysis, like the Mars rovers ([16]). In industry, factories use mobile robots in production to transport packages in a controlled environment. Increasingly used with evolving technologies, unmanned underwater vehicles benefit from improved decision making to perform more autonomous missions while operating in harsh and unknown environments. These missions can vary from the inspection of structures, to the exploration of the environment or the execution of specific tasks ([67],[171]). This field of mobile robotics constitutes a part of the context in the experimental validation.

However, real autonomy through decision making is still underdeveloped for mobile robots. Indeed, the human often intervenes during missions to solve complex unwanted situations. These unwanted situations may come from the environment in which mobile robots evolve ([172]). The environment is potentially dangerous and uncertain, in particular for underwater robotics which must face multiple constraints, we now detail.

Firstly, electromagnetic waves does not propagate underwater. It is then impossible to use a conventional GPS system using only satellite triangulation. This characteristic complicates the localization of underwater robots and the communication with the surface. Localization is most often provided by acoustic or vision sensors, potentially more expensive and less accurate than conventional GPS sensors. Moreover, the measurements of these acoustic or vision sensors are very dependent on the environmental context (salinity, temperature, luminosity, turbidity, pressure). A cable is often necessary to communicate efficiently between the robot and the surface.

Secondly, robots must be particularly powerful and robust in the underwater environment. The friction forces of the water on the robot are greater than those of the air, especially when countering the sea current, which implies a significant power requirement for the robot to move. With humidity and salinity, corrosion develops rapidly and can then weaken the enclosures and components of the robot. The increasing pressure with the depth results in strong physical constraints on the robot enclosures or the external components. The incompatibility between water and electronic components is also highly restrictive and requires impermeability of the robot enclosure. External elements can also disturb the evolution of the robots in their environment. The pollution of the oceans by plastics and larger solid objects increases with time. This creates additional obstacles, in combination with the marine species that are normally present. Obstacles present can damage robots by colliding or blocking sensitive parts, such as sensors or actuators.

These environmental constraints can lead to operational problems in robotic systems performing autonomous missions, in addition to the internal errors intrinsic to the operation of robotic systems. Surveys indicate that underwater robot suffer from different failures in long autonomous missions ([53],[141]). Errors are found in software or hardware, and may impact mission or safety aspects. In ground robotics, statistical surveys conducted by [24] also list a large number of failures that occur on several mobile robots and widely impact their functioning.

Dependability is the ability of a system to safely produce a desired service ([14]). In order to ensure the autonomous mission and the safety of the robot in its environment, it is then necessary to use the dependability means, which enhance the system to prevent its failures. The most common and evident solution is to increase the reliability and the redundancy of the system components. This approach is widely used in critical systems (space, aeronautics, nuclear). However, the introduction of more reliable components or redundant components results in an overall increase of resources usage like the volume needed, the energy consumed and therefore the energy autonomy. All these factors play a role in the choice of system design, which is often the result of a trade-off between cost and dependability. The dependability issues are therefore even more present in low-cost systems, whose components fail more frequently and provide lower quality services. The dependability problem is also central for the management of complex missions, for which the context may raise difficulties.

In this thesis, we therefore address the following question: *For all robots, including low-cost mobile robots, can an autonomous control architecture be designed to prevent the occurrence of a maximum of undesirable situations that may impact safety or mission execution?*

We believe that it is necessary to add software-based, fault-tolerant and versatile decision making to improve the dependability of a robotic mission.

To address this question, the manuscript will proceed as follows:

Chapter 2 reviews different concepts of the state of the art. There are a multitude of methods in the literature to improve the dependability of mobile robots, including fault tolerance. In particular, we examine the ability of methods and frameworks to deal with adverse events occurring at multiple abstraction levels, as related to the

mission, safety or control loop. We also observe the different types of reactions at different levels: from mission re-planning at a high level to local resource management at a lower level.

Chapter 3 details our contribution. We propose an original and versatile fault tolerance framework. Its objective is to limit the occurrence or propagation of errors in mobile robots to minimize undesirable consequences for the mission, the safety and the control loop functionalities. From the detection of a set of undesired events, the framework provides an event inference process based on fault trees. The recovery processes, based on local or decisional reactions, are then described.

In chapter 4, we apply this fault tolerance framework on an underwater robot. We first present the hardware and software architecture of the robot. We then describe the case study mission that the robot must perform. It is a biological assessment mission called transect. We describe the fault trees according to the mission, safety and control loop aspects. We also describe more specifically the solutions implemented to detect and react to the identified undesired events.

In chapter 5, the various validation processes to evaluate and validate the introduction of the framework in our system are detailed and evaluated. A static analysis is first used to estimate the coverage of the implemented fault tolerant reactions. Testing is then performed in simulation and in the field to validate the behavior of the system. Furthermore, we observe the impact of the framework on the dependability of the system in simulation on a large number of runs. A discussion is finally brought on the validation of these tests.

In chapter 6, we finally review the main original results stated throughout this work. We conclude on the effectiveness of the proposed embedded framework on our system and discuss its limitations. We also consider improvements that could be made, in particular to apply our framework to a more general context.

Chapter 2

State of the art

Contents

2.1 Architectures of mobile autonomous robots . . . 5
2.1.1 Generic mobile robot architectures . . . 6
2.1.2 Control architectures of mobile robots . . . 6
2.2 Dependability concepts . . . 8
2.2.1 Dependability attributes and threats . . . 8
2.2.2 Fault prevention . . . 10
2.2.3 Fault forecasting . . . 12
2.2.4 Fault removal . . . 17
2.3 Fault tolerance . . . 22
2.3.1 Fault tolerance and FDIR approach . . . 22
2.3.2 Error detection and fault diagnosis mechanisms . . . 23
2.3.3 Recovery mechanisms . . . 26
2.3.4 Error handling by passive redundancy . . . 28
2.3.5 Evaluating the efficiency of a fault tolerant system . . . 29
2.4 Autonomy and fault tolerance in robotics . . . 29
2.4.1 Managing safety with fault tolerance . . . 30
2.4.2 Managing mission execution with fault tolerance . . . 32
2.4.3 Managing control loop with fault tolerance . . . 33
2.5 Conclusion . . . 35

The implementation of long autonomous robotics missions requires an interest in the confidence we can place in the system. This confidence is addressed by the dependability concept and more particularly the fault tolerance. We will discuss the hardware and software architectures that underpin the deployment of dependability before considering the different approaches that can contribute to it. We will then focus more precisely on one mean of dependability: fault tolerance. We finally focus on the implementation of these concepts for autonomous robots.

2.1 Architectures of mobile autonomous robots

A mobile robot is a combination of hardware (sensors, actuators, controllers, power supplies...) and software (programs, memories...). The robot can therefore be described through a hardware architecture or a software architecture. It can interact with other entities, it has a boundary, which is a frontier between the system and the environment. A robotic system produces a service or an output.

2.1.1 Generic mobile robot architectures

An autonomous robot perceives its environment thanks to its **sensors**, processes this information with its embedded computer (**controller**), and generates an action with its **actuators** ([119]). The sense-plan-act approach is the very base of any robotic system ([80]). An autonomous robot closes the loop between perception and action (Fig 2.1).

The sensors collect the knowledge of the environment (exteroceptive sensors) or from the robot itself (proprioceptive sensors). In the controller, a more or less sophisticated algorithm calculates then a command to perform a given task and sends it to the actuators. A mobile robot actuation relies on the third principle of Newton: the actuators generates a force on the environment which results in creating an opposite force that moves the robot or makes it interact with his environment.

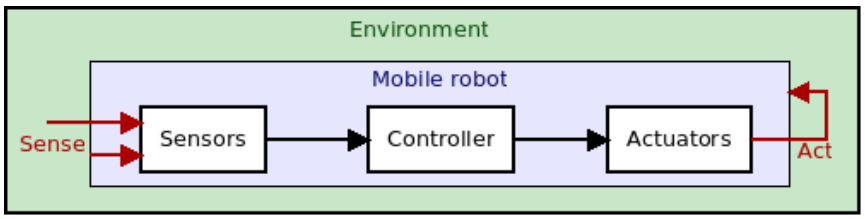


FIGURE 2.1: A generic control loop structure for a mobile robot

The "mobile" character of a robot is its ability to move between locations ([119]). In this manuscript, we take then the definition of [80]: "A mobile robot can be defined as a mechanical system capable of moving in its environment in an autonomous manner". The notions of intelligent behavior and planning are also often associated with the notion of autonomy. An autonomous robot must meet the specifications assigned to it by carrying out a mission composed of one or more robotic tasks, themselves broken down into computer or mechanical functionalities executed in sequence and/or in parallel ([96]).

The component models in the different robotics software share the same structure ([149]). The *system* contains a set of interdependent entities. The *components* or modules are the system constituents that provide the functionalities. They are themselves composed of sub-components or sub-modules. The *ports* are the component's communication end-points. The *interface* constitutes a set of operations which is available outside. The *connection* provide the wiring between the ports of the components and the *data type* represents a classification of data.

Within the controller, there is usually a more or less elaborate control architecture. We will see in next section there are different specialisations of control architectures for mobile robots. It is important to know them in order to identify how the different architectures are affected by errors and to propose the adequate correction.

2.1.2 Control architectures of mobile robots

As stated in the previous section, the control architecture of an autonomous mobile robot is a algorithm localized in the controller, taking sensor measurements as input and generating actuator actions as output. The control architectures are generally derived into ([39], [97]):

- reactive architecture: this architecture propose a quick reaction after the sensing. The decisions are directly taken by functional components. In such an architecture, there is generally no modeling of the environment ([22]). It is often complicated to handle complex mission with such an architecture or to add functionalities on a pre-existing architecture. This architecture has since been abandoned, at least in its simplest form with the emergence of more powerful and faster controllers favoring other architectures.

- deliberative or hierarchical architecture (Fig 2.2): we generally distinguish 3 layers in deliberative architecture ([51]). From the specifications, the user inputs and the feedback from the executive layer, the decisional layer settles the different robotic tasks to be executed. From the task to be executed and the feedback from the functional layer, the executive layer selects the elementary functions that the functional layer has to perform. The functional layer contains elementary functions that form an interface between the hardware (sensors and actuators) and the executive layer. Despite the efficiency of this architecture to manage complex robotic missions, this architecture is rather slow to react to unexpected events because the information must travel through the different layers to be taken into account by the decisional layer.

- agent architecture: an agent is a software or hardware entity that is assigned a mission or task to perform autonomously in cooperation with other agents ([97]). Some systems have been developed using agents interacting with each other, as for multi-robot applications([157], [61]). This architecture offers a great modularity and is particularly efficient to handle complex missions but it often requires a heavy implementation.

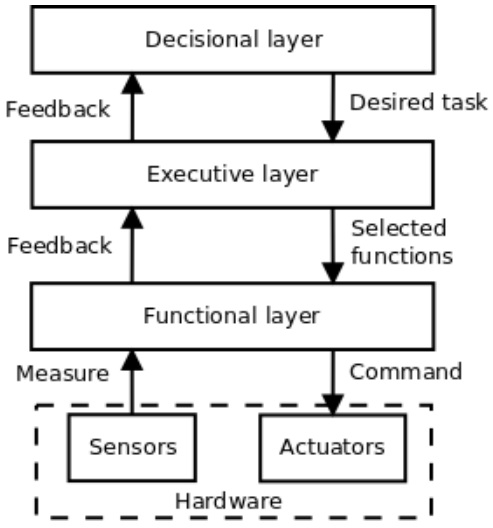


FIGURE 2.2: A deliberative architecture

Nowadays, we rather find combinations of the previously mentioned architectures commonly referred to as hybrid architectures. Hybrid architectures allow the advantages of each architecture to be leveraged. For example, they can possess the layers of a deliberative architecture with direct interactions between layers and modules, that are features of reactive architectures ([39],[120]).

The presented robot control architectures have been used for decades and are well received in the robotic community. Unfortunately, a robot behavior can deviate from the expected behavior due to external or internal threats. Therefore, in order to deal with these potential malfunctions, principles of dependability must be considered and implemented. We present these concepts of dependability in the following section.

2.2 Dependability concepts

In this section, we introduce the attributes of dependability and the threats that a robotic system may face. We will also discuss 3 different approaches developed to enhance dependability, specifically fault prevention, fault prediction and fault elimination.

2.2.1 Dependability attributes and threats

On any system, "dependability is presented as a global concept that includes the usual attributes of reliability, availability, security, integrity, maintainability" ([14]). We give the definitions of these attributes according to [73], [14] and [7]:

- **reliability**: ability of a system to perform a correct service under stated conditions, within a given scope and during a given period. The Mean Time To Failure (MTTF) is defined as the average time of proper functioning before the next failure and is a measure of reliability.
- **availability**: readiness for a correct service. With MTTR the Mean Time To Repair, the availability can be estimated on a system by:

$$availability = \frac{MTTF}{MTTF + MTTR} \tag{2.1}$$

- **safety**: ability of a system not to cause catastrophic consequences on the user or the environment.
- **integrity**: absence of improper system alterations.
- **maintainability**: ability to undergo modifications and repairs.
- **confidentiality**: absence of unauthorized disclosure of information.

The service produced by a system can be altered by threats. We distinguish 3 categories of threats differentiated by their role in the chain of causes and consequences:

- **failure**: a permanent or intermittent interruption of a system's ability to provide a correct service.
- **error**: a deviation of the system behavior that is susceptible to cause a failure.
- **fault**: the adjudged or hypothesized cause of an error.

In a system, a dormant fault is a deviation from the nominal state that does not affect system performance because the fault is masked or because the normally affected functionality is not being used at that time. In opposition, when an internal or external fault becomes active, it causes an error which can spread and cause a system failure (Fig 2.3).

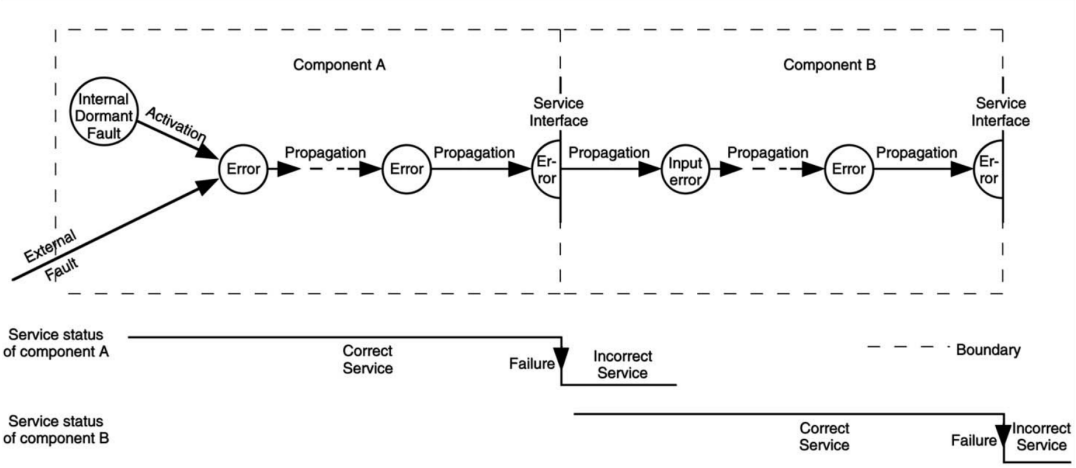


FIGURE 2.3: Fault occurrence, error propagation and system failure
Source: [14]

Faults can be considered in several aspects called classes represented Fig 2.4. The notions of system boundary and fault dimension are important because they provide information on the location of the fault. Similarly, the notion of fault persistence is interesting as it provides an indication on the duration of the fault.

The notion of fault **severity** or magnitude is also found in the work of [14]. The severity of a fault is directly related to the severity of the system failure it causes. The severity of a failure is an indication of the failure impact on the system. Works including risk analysis estimate a quantity or a precise qualification to the severity of failures on pre-established scales ([39]). For example, a minor severity can be a small disruption of a module; while a fatal severity can be the destruction of a critical part of the system, preventing the mission from continuing.

Faults and failures can occur in any autonomous mobile robot. On autonomous mission in known or unknown environment, there is a necessity to deal with faults, errors and system failures at run time. The means to attain dependability are regrouped in 4 categories in the literature: **fault prevention**, **fault forecasting**, **fault removal** and **fault tolerance**, which we detail in the following sections.

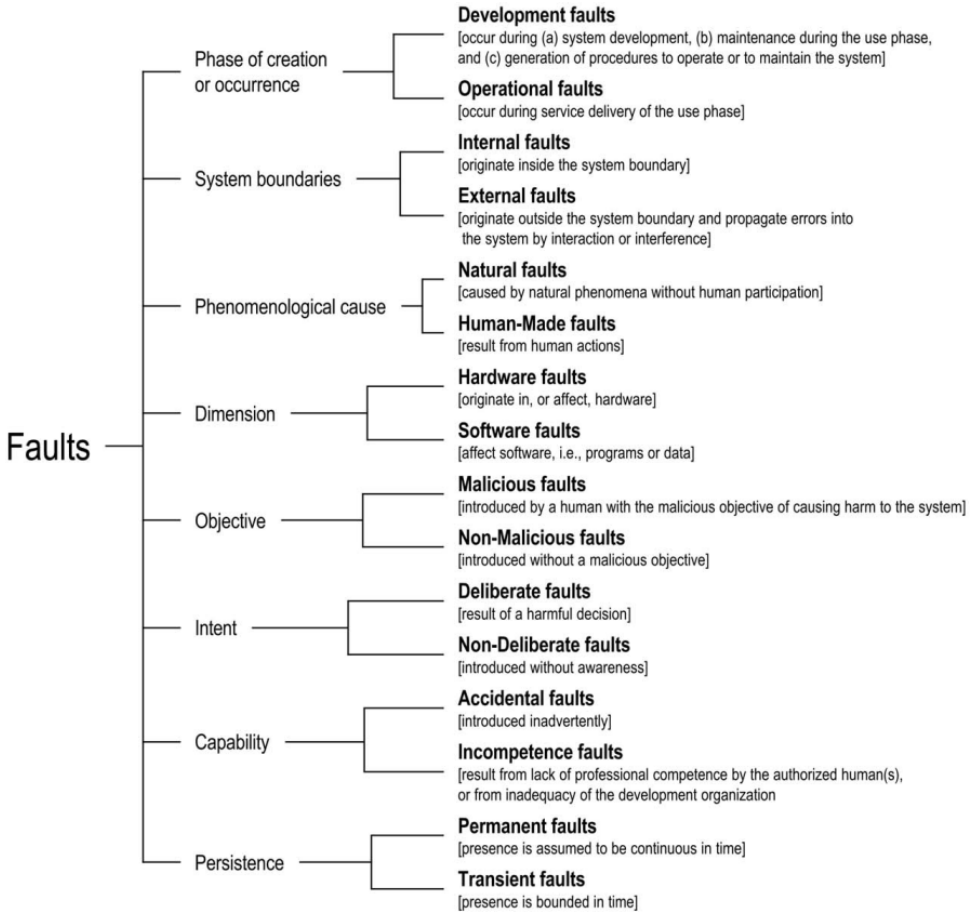


FIGURE 2.4: The fault classes

Source: [14]

2.2.2 Fault prevention

The aim of fault prevention is to reduce the introduction of faults by using proven methods and tools during the system design phase. Fault prevention is also achieved by following specific development rules.

The agile methods and V model are among the most widespread approaches for project organization and risk prevention. Originating from a meeting of scientists in 2001, agile methods emphasize individuals and interaction rather than process and tools, working software rather than comprehensive documentation ([180]).

The V model ([73], [76]) is a representation of the life cycle of a system, from specification to operation. The validation phases are performed by testing the system step by step according to the specifications defined in the associated development phases. When errors are detected, the V-cycle is restarted to correct them.

In the following sections, we present other methods and tools for fault prevention.

2.2.2.1 The robot application development process

The Robot Application development Process (RAP) is a process model for developing robotics applications ([86]). It follows several steps we present in this section.

A first step in the development process is the scenario building which aims at defining the project specifications with the stakeholders. The specifications are

choosing accordingly between the user and the developer. They must be described precisely and be achievable by the system. Most of the time, a formal document and models are established to detail the specifications that the system must meet. Then, platform building consists in configuring the platform that will support the development.

Capability building consists of unitary development and unit testing of isolated functionality that the system must perform. The components or modules are developed independently which facilitates their verification and debugging. Each component can be represented by a black box whose interface is made of inputs and outputs. The work conducted is then combined to assemble the complete system in a laboratory during the system building. This transition can be achieved by using the modularity of the elements resulting from capability building.

The benchmarking phase involves testing the safety and reliability of the system. For example, ISO standards (International Organization for Standardization) and IEC standards (International Electrotechnical Commission) are widely used in the industry to certify that a system meets safety or performance criteria ([161]). ISO-13849 and ISO-12100 aim to define the parts of control systems related to safety and risk acceptance in system design.

Deployment consists of deploying the software application on the actual target system. Testing is also performed to verify that the system still meets the expectations specified at the beginning of the project. Finally, maintenance consists in analyzing the logs, adapting the system and verifying that the system is operational during its life cycle.

2.2.2.2 Model-based development

The use of diagrams, especially UML diagrams ([69]), is particularly effective to discuss and detail the system to avoid errors ([55]), especially for involved people coming from different fields. The 3 main categories of UML diagrams are structure, behavior and interaction diagrams. Structure diagrams are used to detail the static characteristics of a system while behavior and interaction diagrams focus on dynamic characteristics.

Class diagrams are widely used structure diagram designed to express the different implementations of structures describing objects sharing the same characteristics. Use case diagrams are behavioral diagrams. They are often used at an early stage of system design to describe the interactions between system use cases and stakeholders. An example of an interaction diagram is the sequence diagram. It is useful for detailing a use case and expressing the interactions between objects over time.

UML diagrams are already used to describe the functionality of mobile robots ([173], [102], [71]) or to describe robustness concepts in testing for generic computer systems ([110]). They can also be used to automatically generate code ([158]), which accelerates and limits the presence of development-related faults and allows to make a first step towards platform building.

2.2.2.3 Development frameworks

A framework is a structure composed of a set of tools and software components dedicated to the development of applications in a specific domain. Frameworks are often used in the platform building stage to support development for robotics project.

GenoM ([25]) and CLARAty (Coupled Layer Architecture for Robotic Autonomy) are two robotic frameworks. CLARAty encapsulates component technologies developed at NASA centers and partnering universities. It has been used in a deliberative architecture for mobile robots exploring other planets ([120]). Some frameworks also provide safety features for autonomous systems. MISRA-C ([105]), a set of coding rules used for critical systems is used to build a robot that avoids collision with obstacles ([161]).

Programming environments are all kind of tools used during the software development process, including programming languages, IDE and toolchains that facilitate coding and limits mistakes, while saving development time through autocompletion and automatic detection of syntax errors. The development of certified libraries related to programming languages is also a valuable way to accelerate a project and avoid mistakes.

A middleware is a framework dedicated to the development of software architecture that defines the way components of the architecture are communicating. Examples of known middleware are ROS for Robotic Operating System ([139]), Orocos for Open Robot Control Software ([123], [148]) and ContrACT. Often used in the robotics community, ROS is appreciated for its qualities to enhance the modularity between different robotic platforms and thus facilitate the work of system building. This framework is composed of a core launching several nodes, called publishers or subscribers, respectively writing or reading data on topics that are interactions between nodes. Although ROS still suffers from vulnerabilities, especially in terms of safety and real-time constraints, several works have been done to enhance its dependability ([4], [90], [93], [33]).

Real-time middleware allows to bring additional constraints on the execution time of the programs. In the LIRMM, the real-time middleware ContrACT ([126]) is used in the work of [96] and [78]. Through a scheduler, the decisional layer manages the synchronous and asynchronous modules of the executive layer. Some operating systems also provide real time functionalities (FreeRTOS, RTLinux...).

The tools and methods which limit the introduction of faults presented in this section are not sufficient to prevent all the faults. On the one hand fault prevention is not an exhaustive method, on the other hand, it does not cover operational faults. There is therefore a need to identify the possible faults that may occur and their consequences on the system. This is the objective that we present in the next section.

2.2.3 Fault forecasting

Fault forecasting aims at estimating the number of faults and their consequences. It is mainly based on risk analysis, which is the first step in a risk assessment process that consists in analyzing the failures that can occur in the system and estimating their probability of occurrence. We distinguish two broad classes of risk analysis techniques: qualitative and quantitative evaluations.

Qualitative evaluation aims to identify failure modes or combinations of events that lead to system failure (e.g. Failures Modes, Effects and Criticality Analysis), while quantitative analysis aims to evaluate the occurrence of events in terms of probabilities (e.g. Markov chains). Both evaluations depend heavily on the expertise

of the one conducting the techniques. We will also see that fault trees can be used for both qualitative and quantitative evaluations.

2.2.3.1 Risk analysis concepts

HAZOP (HAZard and OPerability) and FMECA (Failures Modes, Effects and Criticality Analysis) are 2 types of risk analysis, more commonly used in the industrial environment. HAZOP consists in identifying how a process can deviate from its nominal behavior, by defining the system boundaries, identifying potential errors, their causes and consequences using process flow diagrams and the means of detection and prevention to counter the errors ([38]). It can be completed by a FMECA analysis. [174] uses SHARD (Software Hazard Analysis and Resolution in Design), a variant of the HAZOP method that analyses each function of a software system using a set of keywords to guide the process.

During FMECA, an expert generally lists the functionalities of a system in a table. The failure modes of the functions are then noted with their consequences on the system. The severity of each failure is informed, along with an estimate of its frequency of occurrence and its potential causes. Severity and frequency estimates take the form of adjectives as seen previously or integers. Criticality is thus often defined as a function taking into account these two parameters. For example, it is not uncommon to consider $criticality = severity \times frequency$ ([39]). The means of detection are also informed as well as the action chosen to reduce or prevent failures ([57], [29]).

In summary, HAZOP and FMECA methods are intended to be synthetic and systematic methods that identify possible failures and propose solutions. However, the temporality and the sequencing of faults and failures are not taken into account in these analyses. The conduct of these analyses can vary depending on the expert who employs them and the establishment of the cause of a failure is not systematically well emphasized. We will see in the next section that logic trees are used to address some of these shortcomings.

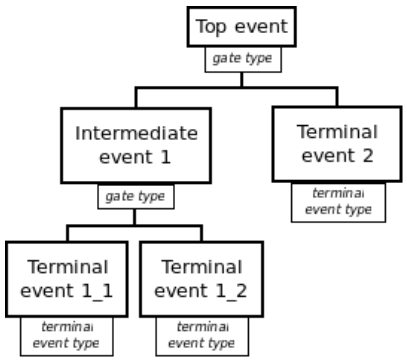
2.2.3.2 Risk analysis using logic trees

Logic tree concepts There are two main types of logic trees for risk analysis: fault trees and event trees. An event tree follows an inductive logic by representing the different consequences induced by an initiating event (fault). It is constituted by layers of several nodes translating the alternatives of the components' functionality. Thus, at each level, they express a probability of an Undesired Event (UE) and its opposite with logical OR operator ([112], [127], [121]). The main objective of event tree analysis is to mitigate bad outcomes by focusing on the consequences.

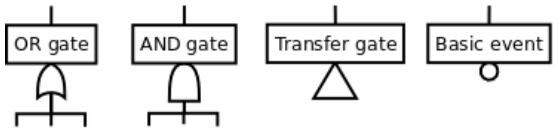
A Fault Tree (FT) follows a deductive analysis by identifying the necessary, sufficient, immediate and probable undesired events causing an observed failure ([154]). The tree is generally arranged from top to bottom, in layers with nodes. Each node details the combination(s) of events, translated with logical operators (AND, OR), that can produce the previous event. The main objective of fault tree analysis is to prevent critical failures by focusing on the causes.

Fault trees appear to be particularly interesting because they provide in-depth information on the connections between undesired events, that was missing for the HAZOP and FMECA methods, and they allow for both qualitative and quantitative analysis. Moreover, they have been widely used in safety critical applications, including robotics.

Fault tree terminology The observed failure is called Top Event (TE). Other undesired events are commonly labelled as intermediate events or terminal events. The designation of each event is usually filled in a rectangle and can follow a custom nomenclature. The top event and the intermediate events are also associated with operators called logical gates (Fig 2.5).



(A) A generic fault tree



(B) Graphic representation of gates and basic events

FIGURE 2.5: Fault tree main components

We identify four main types of terminal event: basic events, undeveloped events, house events and conditioning events. Basic events are initial faults that require no further development. Undeveloped events are not developed further because of insufficient consequences or unknown causes. Domestic events do not represent a deviation of the system from nominal behavior but have a logic-related contribution. Finally, conditioning events specify conditions or restrictions that apply to any logic gate.

We identify six main types of logic gate: OR gate, AND gate, transfer gate, XOR gate, combination gate and Inhibit gate. An OR gate implies that the associated event is true if at least one of the downlink events is true. An AND gate implies that the associated event is true if all the downlink events are true. Transfer gates indicate that downward connections are expanded in an other fault sub-tree to ensure a readable graphical representation. More rarely used, the XOR gate implies that its associated event is true if exactly one of the downlink events is true. The combination gate implies that its associated event is true if $k \in \mathbb{N}$ downlink events are true. Finally, the inhibit gate implies that the associated event is true if the downlink events are true in the presence of a Conditioning event.

Nevertheless, classical fault trees do not take into account the order of occurrence of events. This aspect is studied in dynamic fault trees.

Dynamic fault trees Dynamic fault trees have been created for systems in which the order of occurrence of undesired events has an impact on other events ([154], [19], [26], [11], [13], [40]). Three new gates are introduced: the PAND gate, the FDEP gate and the spare gate. The PAND gate (Priority AND gate) implies that the associated event is true if the downlink events are true in a specific sequence indicated by a conditioning event. The FDEP (Functional DEPendency gate) implies that basic events associated with it are functionally dependent on a triggering event. Thus when the triggering event is true, the dependent basic events are true. Finally, the spare gate implies that the associated event is true if the downlink basic events, forming the failure of the primary and the failure of the succession of the spares, are true.

There are strong similarities between FDEP gates, spare gates and AND gates. The difference is especially visible during the fault tree analysis phase.

Fault tree construction Although fault trees are usually built by safety experts with extensive knowledge of the system, the construction of the trees can vary depending on the approach used. A basic method of building fault trees is to complete them by following the dependency chain of components in a diagram representing the system ([154]). Other construction methods aim to list the different categories of undesired events by type. For example, [176], [177] and [10] define several groups of failures for autonomous underwater robots such as "energy system failure" or "autonomous control fault" and develop the intermediate and basic events in these categories.

Probability laws Probability laws are used during the fault tree analysis, developed in the next paragraph.

Let X be a random variable and Ω be the set of possible outcomes of a random experiment. The function $X : \Omega \rightarrow E$ that associates with each element of $X(\Omega)$ a probability is called the distribution function or probability law of X. Probability laws are often used to characterize the probability of occurrence of faults or failures over time. We can then use the probability density function to calculate the probability of occurrence of the event on a time interval.

Constant, normal and Weibull distributions functions are common for characterising component failure rate ([168]). Non-parametric methods can also be used to estimate the probability of failure without making any assumption with respect to the shape of the distribution ([53], [141]). Surveys have revealed that the probability of loss of a mobile robot is influenced by time, distance traveled and environment ([21]).

The probability laws of basic events are given by a datasheet provided by the component/functionality designer or are estimated by multiple tests. This work becomes cumbersome for complex systems with a large number of components.

Fault tree analysis The qualitative aspect of a FTA (Fault Tree Analysis) lies in the objective of listing the possible cut sets, e.g. the sets of combinations of terminal events that lead to the top event. We call the Minimal Cut Sets (MCS) the smallest combinations of terminal events that trigger in the top event. The MCSs thus represent all possibilities of how terminal events can cause the top event. For example, [68] uses fault trees to find potential MCSs in a hydraulic system. The authors point out that depending on the fault tree construction and the gates used, the potential MCSs found are different which illustrates the importance of the construction phase and the selection of events and gates.

The quantitative aspect of a FTA lies in the objective of estimating the probability of occurrence of the top event. The probability density of the top event is calculated for a time interval using the probability densities of the terminal events and the gate combination. For a given time interval, let $P(E_i)$ be the probabilities of failure of N events under a gate G and P_gate the probability of occurrence of the event associated to G:

- if G is an OR gate, then $P_gate = P(\cup E_i)$
- if G is an AND gate, then $P_gate = P(\cap E_i)$

Most of the time, it is assumed that the events are independent and have a very low probability of occurrence. In that case:

- if G is an OR gate, then $P_gate \approx \sum_{i=1}^{N} P(E_i)$
- if G is an AND gate, then $P_gate \approx \prod_{i=1}^{N} P(E_i)$

It is also possible to calculate the probability density of each MCS, to identify the ones most likely to occur. In other words, it amounts to analyzing the weaknesses of the system, with the final objective of enhancing its reliability. On the one hand, the improvement of the system can be reflected by a decrease in the probability of occurrence of terminal events by improving the quality of components/functionality. On the other hand, the improvement of the system can be induced by introducing new component/functionality (redundancy), which increases the size of each MCS with the introduction of new gates (AND, PAND or PDEP) and events.

It is sometimes impossible to precisely establish probability laws of basic events or to integrate human errors, this is when the fuzzy fault tree analysis enters the picture.

Fuzzy fault tree analysis Fuzzy Fault Tree Analysis (FFTA) ([101]) has also been used by [176] and [45]. Fuzzy logic provides a context in which uncertainty and preference can be effectively modeled. FFTA consists in analyzing a fault tree without a precise probability of the basic events. The fuzzy probability of the top event is calculated from the fuzzy probability of basic events and MCSs.

Automated approaches to fault tree construction and analysis Tools have been developed to provide a more generic construction and analysis of faults from block system description software. The use of these tools requires meticulous work and a good knowledge of the system.

For example, HiP-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies) ([1]) is a software developed to run with Matlab Simulink. From an architecture described in blocks, the software can perform a fault tree analysis and a FMEA of the designed system. Its purpose is to accelerate the dependability analysis of complex system by providing semi-automatic construction of fault trees and FMEA.

AADL (Architecture Analysis and Design Language) ([44], [107]) is a system description language that can be used with a textual or graphical API. The programmer declares the components of the system which can be of various types, from software entities (programs, threads...) to electronics equipment (devices). These entities connected to each other through their ports feed on inputs and generate outputs. EMV2 (Error Model Annex V2) is an AADL annex that defines the error models in different components and the propagation of these errors in the system. EMV2 allows to represent the type of output failure and how it affects the input of another entity. From EMV2, [44] developed a tool to perform an automatic construction and analysis of fault trees to determine the dependability of a system. These analyses consider OR, AND, Exclusive OR, Priority AND and Voting OR gates. It should also be noted that work exists to bridge the gap between AADL and Hip-HOPS ([107]).

Fault trees are an intuitive graphical representation of the cause and effect relationships between faults, errors and failures. They allow eventually to reduce the probability of occurrence of failures. However, as every risk analysis method, the

construction of the fault trees requires a colossal amount of work and is largely dependent on an expert, the laws of probabilities are not always easy to obtain and the temporality is only approached in a limited way through the dynamic fault trees.

In the next section, we will address the System-Theoretic Process Analysis as a last fault forecasting technique.

2.2.3.3 System-Theoretic Process Analysis

System-Theoretic Process Analysis (STPA) is a hazard analysis based on the System-Theoretic Accident Model and Processes (STAMP). "In addition to component failures, STPA assumes that accidents can also be caused by unsafe interactions of system components, none of which may have failed" ([95]). These unsafe interactions in the system control loop may cause safety or mission constraint violations which should be avoided for critical complex systems.

According to [95], STPA is divided into 4 steps. The first step is to at define the purpose of analysis, e.g. to identify the system boundaries, losses, risks and constraints. The second step is to model the control loop of the system, for example by using graphs. The third step is to identify unsafe control actions which may lead to a hazard in a particular context. A control action can be potentially dangerous if 1) it is not provided, if 2) its content is inadequate, if 3) it is provided too early or too late, or if 4) it is applied too long or stops too soon. The fourth and last step is to identify the possible scenarios that lead to unsafe control actions, like an unsafe controller behavior.

This technique allows the analysis of very complex systems. This techniques also allows to identify errors that could not be found with classic hazard analysis, in order to eliminate or mitigate them later. However, this fault forecasting technique is also very time consuming and requires an in-depth knowledge of the system.

Fault forecasting only allow to identify some faults in the system and their criticality. However it does not remove faults. This is the role of fault removal techniques that we address in the next section.

2.2.4 Fault removal

The purpose of fault removal is to reduce the number of faults and their severity. This process is necessary to validate the proper behavior any system. It takes different forms during the development and operation phases of a system's life cycle.

2.2.4.1 Static and dynamic verification during development phase

During the development phase of the system, the fault removal consists in repeating in loop the pattern: verification, diagnosis and correction. The correction is dependent on the system and does not require further attention. The diagnosis will be studied in the section 2.3. The verification can be conducted in a static or dynamic approach (Fig 2.6).

The **static verification** consists in verifying the system without executing it. This verification can be performed on the system itself using static analysis or theorem proving, or on a behavior model using model checking ([128]).

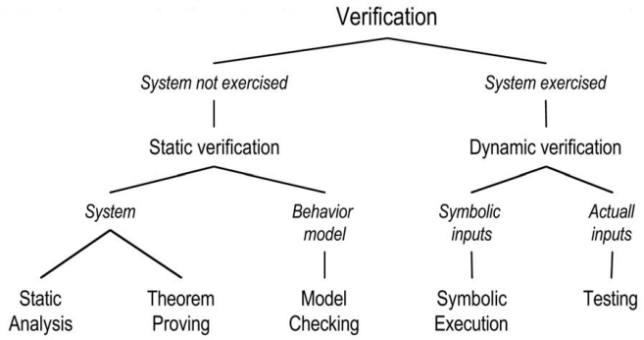


FIGURE 2.6: Verification categories
Source: [14]

The **dynamic verification** consists in evaluating the system by running it. Testing is widely used to certify that a system meets the requirements set by the specifications and to ensure its safety. Particular attention must be paid to the test vectors to ensure the validity of the testing. We study in the next section the main families of tests used to validate autonomous systems.

2.2.4.2 A testing classification

Although it is impossible to test a robotic system exhaustively, several categories of tests can be used to validate various system characteristics. Among the main categories of validation for computer and robotic systems, we find:

- **Functional testing** is used to verify the system complies with the specifications. This testing is often performed in the industry where the system needs to respect specific norms. It is often part of the final stage of system integration.
- **Testing to the limit** consists in testing the functionalities of the system to the limits of its specified requirements. **Stress testing** consists in testing the system beyond the limits of its specified requirements.
- **Non-regression testing** checks that the new mechanism does not negatively impact the behavior of the system. In the case of fault tolerant mechanisms, it could be either by overloading the system, by overreacting to non-present events or by creating new catastrophic failures.
- **Usability testing** involves a third party user whose purpose is to test the system from an external point of view than the developers. Usability testing is efficient to test system ergonomics and to deepen the tests previously performed. This test is widely used in the development of software and video games.
- **Unit testing** verifies each component or software module or functionality individually before they are integrated into the system. This step is effective in deducing the characteristics and reliability of each component separately.

These various testing can be carried out according to 2 different approaches: **white box** approach and **black box** approach. White box testing involves testing the internal structure of a robotic system, such as the code or internal functionality. Black box testing does not focus on the internal details of the system but treats it as a black box that feeds on inputs and generates outputs.

Belonging to this last category, oracle test is widely used in computing science to determine if a system verify specifications (Fig 2.7). This method compares the expected results to the measured results on a multitude of tests. Given the context and the test inputs, the oracle provides a verdict on the validation or not of the system ([147], [108]).

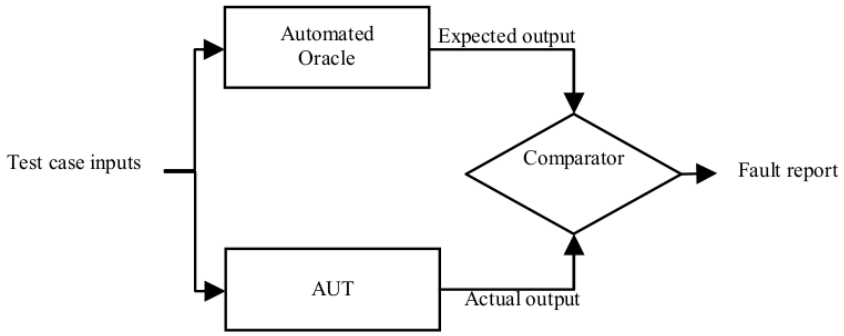


FIGURE 2.7: An automated oracle test
Source: [147]

Both [136] and [65] use an adaptation of the oracle test for autonomous vehicles, that is based on the evaluation of invariant properties characterizing mission and safety among others. In these works, oracle test aims to define the failure rate of these properties and to establish whether or not the systems meet expectations.

To increase the confidence of any test process, developers often use a large number of runs ([8]) which allows to average the impact of random parameters that may not be identified. However, the testing time is necessarily bounded, which can prevent the appearance of some faults and failures. We will therefore see in the next section how to deal with this point with fault injection.

2.2.4.3 Fault injection process

Fault injection consists of deliberately causing a fault or an error in the targeted system. It is often used during tests to evaluate the response of systems to injected faults and to validate their behavior ([8], [27],[97]). This method ensures the occurrence of the targeted fault but also allows to control the time of its emergence. This allows the system response to be analyzed with additional precision, since the difference between the fault detection time and the fault occurrence time is known. Fault injection can also be used to test the behavior of the system in response to faults that do not often occur to be present in more traditional tests.

Nevertheless, this raises the question of the choice of the injected faults. These faults are traditionally chosen from a representative set of undesired events that the system may potentially encounter after a fault forecasting study. However, this study can only predict a part of all the faults that the system may encounter in the field. Moreover, there is no guarantee that the injected faults will be faithful to those naturally present without the intervention of the fault injection process.

A fault injection test sequence is composed of several elementary experiments for which faulty conditions are applied to the system. The input domain of a test sequence is made up a set of injected faults and a set of activation parameters that

defines the activation conditions of the injected faults. The injected faults can be introduced through physical or electrical alterations, or by software variable alterations.

The output domain is made up a set of readouts or measurements collected during the test and a set of measures obtained by analysing the injected faults, activity and readouts. These measures should indicate whether the system deals with the fault correctly. In the literature, the FARM (Fault, Activity, Readouts, Measures) ([7]) characterizes a fault injection test sequence.

We will discuss in the next section that the tests can be performed on a symbolic model or on the real system.

2.2.4.4 Tests in simulation versus tests in the field

The validation of complex systems may follow a meticulous process. During the Deep Space 1 project ([152]) for the design of a satellite, functionalities are tested independently at first, then a first model is simulated. Subsequently, a scale model test is performed, then the real system is tested in the laboratory, and finally the real system is tested under real conditions.

We find different degrees of simulation of models or environments ([136]):

- Model in the Loop (MiL): tests are performed on a model of the target system.
- Software in the Loop (SiL): tests are performed on the real software of the target system and in a simulator.
- Hardware in the Loop (HiL): tests are performed on the target system and sensors are simulated.
- Test in the field (TiF): tests are performed on the target system evolving in the real environment.

Testing with a simulated model Tests are easier and faster to perform on simulated models as they are often less complex than the final system. However, the simulation of a system model leads to an inherent simplification of the reality, resulting in a potential drift of the set of faults and failures that may occur.

Testing with a simulated environment (MiL, SiL, HiL) Similarly, the autonomous system can be designed to evolve in a real or in a simulated environment. Simulations are efficient to rapidly detect the different faults in a system ([137], [153]). Different simulators have been developed for that purpose, for instance Gazebo or Morse. Gazebo ([52]) can be easily interfaced with ROS ([43]) while Morse is dedicated to academic robotics and can be extended to the simulation of the marine environment ([63]). Companies also develop their own simulator for a better adaptation to their needs. Thus, [96] and [76] developed underwater robot simulators for biological assessment and ship inspection.

Simulators are often integrated into a larger framework. For example, ORCCAD ([17]) is a programming environment for robotic systems providing tools for code implementation, formal verification and simulation. Simulators can also offer additional functionality. For instance, [30] introduces SESAME (Software Environment

for Software Analysis by Mutation Effects), a fault injection tools that mutates the program generated by several languages to verify the robustness of the generated program.

A simulated environment generally provides a controlled context of the mission and allows for ground truth, which brings an advantage for the validation of a system. Although simulated environments also provide a simplification of reality, work is underway to generate these environments more easily and to make them more realistic. For example, [136] simulates a farming robot in a simulated field based on the reproduction of real cases or generative approaches.

Testing in the field (TiF) Many bugs can be found through simulation, but it is far from being an exhaustive method for testing mobile robots. Tests in the field are a more realistic way to validate a mobile robot. They enable to consider unexpected faults and resilience of the model in reality. In addition to evaluate component performance, the test in the field allows to determine the potential failure mode of the sensors and the thresholds for fault detection ([62]). In that perspective, the development of test benches is useful to verify the components ([87]). Work is also underway to develop standard test methods for underwater robots performing various tasks [77].

In an open environment, extensive test campaigns offer a reliable representation of the system performance and potential failures. For example, test campaigns of underwater gliders carried out over several years have made it possible to improve the reliability of the glider over time ([141]).

However, these tests remain more complicated to carry out because large resources are often required, especially for underwater robotics. An important challenge with test in the field campaign, is that most test cases require to compare results to what is called "ground truth" (e.g. the correct localization of the robot). The ground truth is often difficult to obtain for autonomous robots, particularly for underwater localization ([65]), as it is usually obtained through the instrumentation of the environment and the use of accurate external sensors. Furthermore, the ground truth is not always clearly defined in the studies ([150], [91]).

Fault removal must be based on fault forecasting and requires a lot of expertise (selection of the right tests) and time. This method uses interesting test tools (simulation, model checking), which allow to eliminate some faults by testing the behavior of a system. Unfortunately, to our knowledge, no work has been done to take advantage of all these approaches to achieve the most exhaustive tests possible.

Despite fault prevention, fault forecasting and fault removal, necessary but not sufficient means presented in this section, it is impossible to avoid every faults during the execution of an autonomous mission, especially if the robot evolves in an unknown or harsh environment. Fault prevention prevent the introduction of some development faults but does not cover all faults. Fault forecasting and fault removal are useful to inventory possible faults, but they rely heavily on the vision of the engineer in charge of the study, which is therefore not certain to be exhaustive. Therefore, it is necessary to deal with runtime faults and their consequences that occur in the system. This is the objective of fault tolerance which we present in the next section.

2.3 Fault tolerance

Fault tolerance aims to prevent system failure despite the presence of faults. Mainly based on the principle of redundancy and recovering, it consists in detecting an error and performing a recovery to overcome the malfunction. Therefore, a recovery aims to eliminate errors or limit their propagation, so that they do not cause a failure. Fault tolerance is a fundamental dependability mean for autonomous systems, as some faults may be unpredictable or cannot be avoided systematically.

In the following sections, we will first introduce the views around fault tolerance and before focusing on error detection, diagnosis and recovery approaches.

2.3.1 Fault tolerance and FDIR approach

In the dependability community ([14]), fault tolerance is basically composed of 2 steps: the detection of an error and the associated recovery (Fig 2.8). After detecting an error, recovery can be carried out using 2 different approaches.

The first approach called error handling is to eliminate or limit an error that occurs in the system. The fault is not necessarily investigated but the robot must be in a state that eliminates the error from the system. The detected error is handled by performing a rollback, a rollforward or by using the internal compensation of the system.

The second approach addressed as fault handling, intends to diagnose the fault, i.e. to find the causes of the detected error, to isolate the fault, i.e. to ensure that the fault no longer produces an error, to reconfigure the system in order to assign a new component/process, and finally to reinitialize to update the configuration.

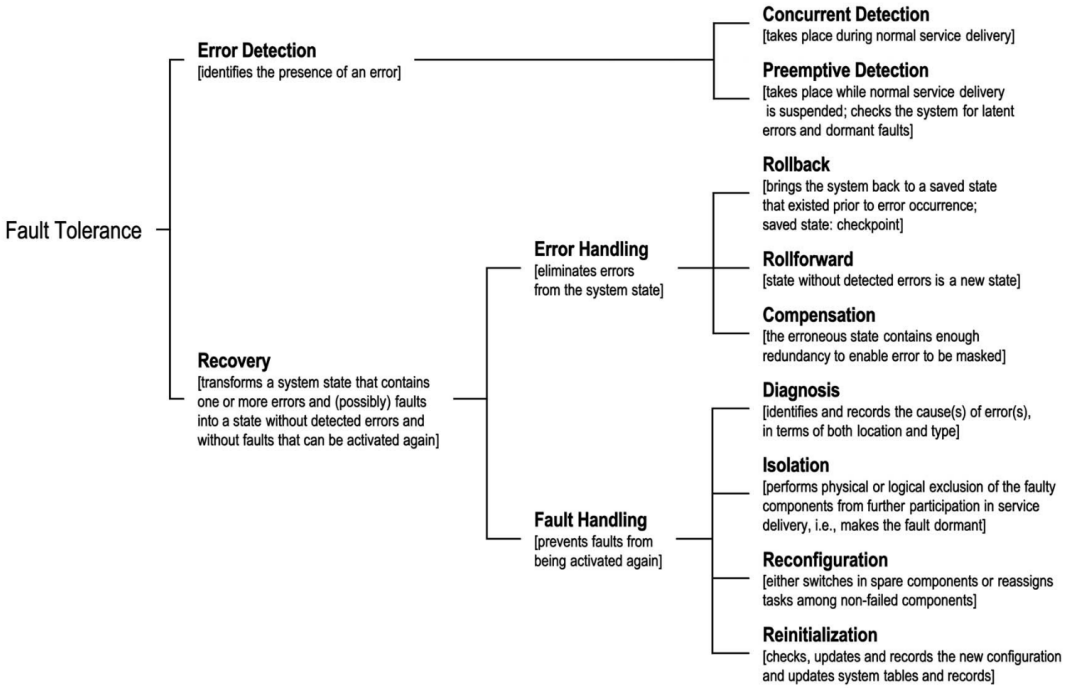


FIGURE 2.8: Fault tolerance concepts in dependability

Source: [14]

The FDIR (Fault Detection Isolation and Recovery) is an instantiation of these concepts, mainly in aeronautics and robotics domains ([184]). The FDIR approach is divided into 3 mains steps: fault detection, fault isolation and recovery.

In the literature, the terminology used in FDIR slightly vary from the classic fault tolerance terminology. A fault is described here as a deviation of a system property or parameter from the acceptable state ([49]). Fault detection is redefined here as detecting an error in the system, that is a deviation from the nominal behaviour of the system. Fault isolation, performed in system equipped with a diagnostic mechanism, is understood here as determining the type and location of a fault ([74]). Finally, the recovery is identified as the reaction of the system to counter the deviation of the system behavior from the nominal behavior.

In complement to the FDIR, we also find the acronym FDD which stands for Fault Detection and Diagnosis ([82], [184]). The fault identification refers to determining the size, the severity and persistence of a fault. Therefore, in this approach, fault diagnosis refers to both fault isolation and fault identification ([184]).

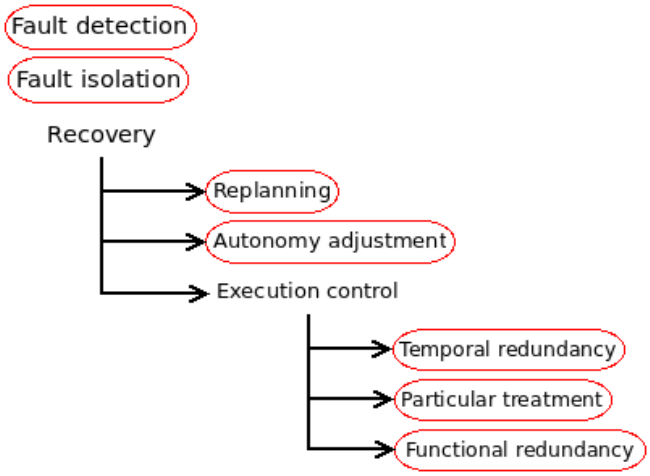


FIGURE 2.9: FDIR approach

In the rest of this section, we will detail the different elements of the mentioned approaches, from the structure of the FDIR approach (Fig 2.9). However, in the rest of this thesis, we will stick to the terms from the dependability that we consider more generic.

2.3.2 Error detection and fault diagnosis mechanisms

Error detection is the first step of any fault tolerant process. It consists in detecting a deviation of the system behavior from the nominal behavior of the system. Error detection can occur during the nominal runtime of the system or during a specific phase that facilitates error detection.

The notions of faults, errors and failures are relative to the system definition. Fault diagnosis and error detection can therefore be similar depending on the granularity of the system. For that reason, most of the papers consider the error detected is a fault. This imprecision can be dangerous, because eliminating an error does not mean eliminating the fault. However, the methods of error detection and fault diagnosis are similar, as we learn in the literature through FDD techniques ([82],[183]).

We will first describe basic error detection methods based on the work of [98]. We will then divide the more advance error detection and fault diagnosis methods into 2 main categories ([183], [49], [50]): model-based and data-based approaches.

2.3.2.1 Basic error detection methods

We will focus on two widely used methods of error detection: timing checks and likelihood checks. Timing checks are used in many autonomous systems under the form of watchdogs ([9]). They allow the detection of breaks and time deviations in the service of the subsystems. It is ensured that the monitored element does not get locked in an determinate state by checking, for example, that it produces its results or emits a dedicated signal within a given time period ([146], [160]).

Likelihood or consistency checks verify that the data produced meet specified validity domains, in order to remove the outliers. This method includes, for example, compliance with the operating ranges of the sensors and actuators specified by the manufacturers, but also compliance with the values produced by the algorithms. In many cases, this method is not integrated into a structured approach, but rather is integrated in the course of time. Likelihood checks can be integrated into safety checks to avoid producing a command that would endanger the system ([85]).

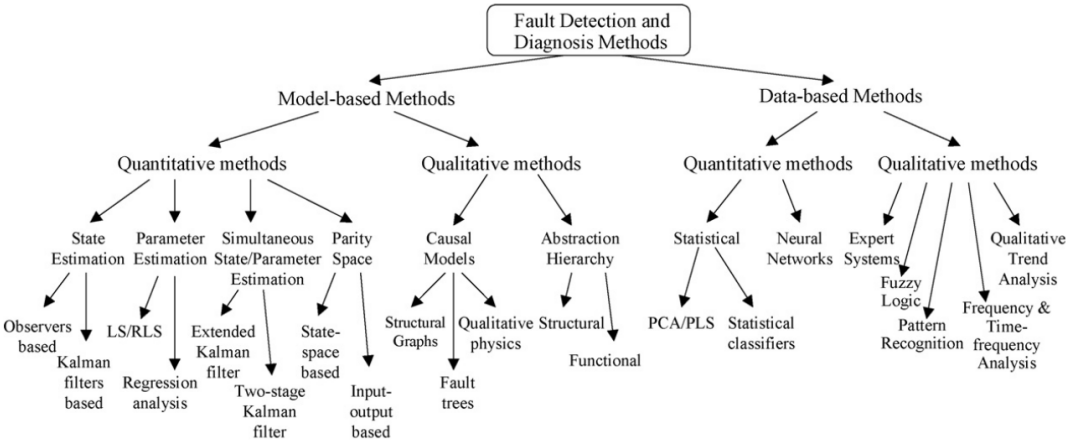


FIGURE 2.10: A classification of FDD approaches

Source: [183]

Some FDD methods use a comparison of a value to a threshold to deduce the error. It should be noted that the thresholds used are not always easy to define and can lead to misdiagnosis. We will see the remaining more advance categories of FDD techniques which are represented Fig 2.10.

2.3.2.2 Model-based fault detection and diagnosis

Model-based fault diagnosis checks the consistency between the measured outputs of the estimated real system and the predicted outputs of a reference system model identified either by physical principles or by system identification techniques ([74]). We review the different approaches in this category.

Diagnosis using state observers The principle of a state observer is to estimate the state of a system from inputs, outputs and model knowledge. Variants of Kalman

filters and particle filters are particularly used in this context, including in robotics. We classically find applications in which several Kalman filters are used to diagnose faults ([178],[15]). A bank of residuals, identified as the differences between the fused models and the inputs, is usually obtained to identify the faulty component.

Other work focuses on particle filters to perform fault detection and diagnosis ([165]). Given sensor data, the algorithms calculate the probability that the robot is in each of the faulty and operational states. In addition, particle filters can handle noisy sensors and non-Gaussian behavior patterns. Subsequent work improves this methodology, for example by restricting most of the computation to the most relevant regions of the state space and applying it to a rover ([164]).

Diagnosis using parametric identification Parametric identification approach considers that faults affect the parameters of the established model. By mathematically isolating the parameters and continuously measuring the inputs and outputs, a parameter estimate can be obtained. The estimated parameters are then compared to the reference parametric values. A flag is then raised if the difference between the estimated parameters and the nominal parameters is too large ([73]).

Diagnosis using parity space The parity space is a space in which all elements are parity vectors (residuals). A parity vector is generated from linear combinations of inputs and sensor outputs, taken over a finite time. The parity equations are chosen to obtain parity vectors that are zero when the components are functioning perfectly, but that may have a subset that deviates from zero when a particular component of the system is malfunctioning ([129]).

Diagnosis using causal model Causal models describe the cause-consequence relationship of a system and can be useful for diagnosis ([79], [125]). We addressed in the section 2.2.3.2 that fault trees can be used to establish a set of minimal cut sets representing the combinations of possible faults that trigger a system failure. For example, [68] uses fault trees to find minimal cut sets of a hydraulic system. It is also possible to use fault trees in a quantitative diagnostic approach ([12],[35]). The probability of occurrence of each basic event or cut set induces a statistical diagnosis of the fault location.

Model-based techniques are numerous and are often used for work on the safety of robotics operations. They are complementary to the data-based approaches which we will address in the next section.

2.3.2.3 Data-based fault detection and diagnosis

In contrast to model-based diagnosis, data-based fault diagnosis methods rely on a history of previous runs. This knowledge-based method uses a database formed with training and learning processes. When a fault occurs, the coherence between the inputs and outputs of the system is then checked in the light of the knowledge base. A diagnostic is then performed using a classifier. For a mobile robot, by comparing the history of sensor outputs and actuator outputs, a failure diagnosis can be made ([167]).

Quantitative analysis Statistical diagnosis and neural networks ([169]) belong to quantitative analysis. These methods consist in solving the diagnostic problem as a pattern recognition problem. Statistical analysis methods consist primarily of the Principal Component Analysis (PCA), Partial Least Squares (PLS) and statistical pattern classifiers. A voting process can be used to eliminate a faulty component or to average the result and minimize the noise.

Qualitative analysis Qualitative analysis relies on expert systems, pattern recognition or Qualitative Trend Analysis (QTA). Emerging in the mid-1980s, expert systems for fault diagnosis are rule-based and use the expertise of a human to form a set of rules ([72]). From a list of faults and failures, an incidence matrix can be used to relate these faults and failures ([39], [59]). However, as a fault diagnosis method, an expert system has low generality and scalability. QTA identifies process trends from noisy process data and separates nominal trends from erroneous trends in the database.

Data-based approaches are not well adapted and not much used in robotics but rather in production engineering. They require data archiving which is difficult to obtain, except with large infrastructures or the use of simulation. However, the interest of these approaches is to learn from its experiences to enrich the database and improve the diagnosis.

Hybrid fault diagnosis Hybrid approaches provide both model-based and data-based diagnostics. For example, in [170], detection of faulty components is based on detecting the deviation of resources compared to learned expected resources. Work also focuses on the calculation of adaptive thresholds for error detection and fault diagnosis. For example, [155] detects residues between a model and the sensor on a Roomba robot using detection thresholds calculated by an adaptive law.

Once the error has been detected and the fault possibly diagnosed, an autonomous system must react to prevent a system failure. We will see in the next section the different possible reactions.

2.3.3 Recovery mechanisms

In this section, we detail the different recovery mechanisms that generally depend on the failures/errors/faults identified during the detection/diagnosis phase and the actions that can be taken to counter the adverse events. We identify 3 main categories of recoveries dedicated to robotics systems ([29],[92]): recoveries by execution control, recoveries by replanning, and recoveries by autonomy adjustment.

2.3.3.1 Recovery by execution control

In order to correct a deviation of the system behavior, the execution control consists in choosing an action to achieve a given plan. The different possible actions are ([39]):

- particular treatment: reconfiguration of the system, reconfiguration of the algorithm, use of a fault tolerant control ([183]). This approach is generally used when the severity of the fault is small.

- functional redundancy: use of another component or algorithm to perform the failed task (comparable to compensation in Fig 2.8). We distinguish hardware redundancy (for sensors, actuators, devices...) and software redundancy (for processes, threads...). Hardware redundancy is often used in aircraft where many components are tripled ([179]) in case of a component failure. In underwater and aerial robotics, [64] and [143] separate faulty actuators from the command chain. The high cost resulting from the choice of hardware redundancy and the additional physical space required for it are often a hindrance when building robots. As a result, software redundancy is often used to limit cost and space. Redundancy can be qualified as cold when components are activated when those already active fail; hot when components work in parallel and a non-failing component can take over from the other failing components; or warm when components are idle before taking over ([78]).

- temporal redundancy ([36]): repeat the faulty process/action, or slightly modified, hoping that the context will be more favorable (comparable to compensation and rollback in Fig 2.8). Rollback consists in returning the system to a former stable state/saved checkpoint. This is impossible for a complete mobile robot due to the physical dimension, but it remains feasible for a part of its software. For example, restarting of a erroneous process ([43]) or component ([145]) is possible. In this case, the restarted entities return to an initial state without errors. In these examples, the restarted processes may not run after a certain time, because the initial fault has not been corrected, which is why it is often important to diagnose the fault.

These actions can be decided locally by a specific module, for example like the R2C component ([132]) or by a decisional manager ([17]). However, when the severity of the fault is significant, execution control is not always sufficient to recover from an adverse situation, it may be necessary to replan.

2.3.3.2 Recovery by replanning

Planning aims at defining a succession of robotic tasks to accomplish the mission. For deliberative and hybrid architectures, the replanning decision can come from the deliberative layer or from an annex mechanism dedicated to replanning ([39], [98]). When the mission can no longer be carried out in an optimal configuration, actions at the planning level are possible:

- anticipation recovery: the system anticipates adverse situations and avoids getting into a delicate position

- plan repair: in case of an error requiring a change of plan, plan repair aims to change the plan locally without the cost of replanning the entire mission. The degraded modes ([142]), i.e. the execution of the same or similar robotic task with less constraining parameters, can be used to facilitate the mission execution in case of adverse situations. The planner can also decide to use safety hold ([92]).

- complete replanning: the tasks of the mission change completely or the mission changes. This may imply a high temporal cost depending on the context.

Replanning is comparable to rollforward recovery in Fig 2.8, which consists in bringing the robot in a new stable state. This state often involves a risk during the transition to the new stable state. For mobile robots, the notion of rollforward can be extended to the return to a safe position. Thus, errors will disappear or have almost no impact on the integrity of the robot ([160]).

Thus, we notice that the autonomous recovery techniques are varied and offer a local reaction by execution control or a global reaction by replanning. However, in some situations, it may be necessary to have a human intervention. While autonomous, robots often have the ability to be controlled by an external operator. In this case, we consider recovery by autonomy adjustment.

2.3.3.3 Recovery by autonomy adjustment

Autonomy adjustment consists in allowing an external operator to control some of the robot's functionalities, in order to bring the human's decisional intelligence to escape from a delicate situation. Most of the time, the operator can control the speed or the trajectory of the robot to avoid critical or unknown situations ([39], [75]). In this case, a transition must be studied to adapt the autonomy without introducing instability.

For some applications, the robot cannot be controlled directly (latency, interference, maneuverability...). In this case, systems have been developed in which the operator can decide or validate each step of complex missions. For example, Mars rovers ([16]) have been deployed for multiple scientific surveys. For sensitive operations, the rovers await orders from operators on Earth before performing the next steps of the mission. In this case, the mission lasts much longer, but the valuable robot remains safe.

We have reviewed the different recovery methods for autonomous robotics. Sometimes an recovery may not be necessary since the robot has already an internal passive redundancy, which we will address in the next section.

2.3.4 Error handling by passive redundancy

The passive redundancy of the system is the expression of redundant elements without initiating any specific additional action. The use of several entities performing the same function is often useful to prevent an error from triggering a catastrophic failure. The error is still present but masked by the redundancy, which makes passive redundancy a form of compensation (Fig 2.8).

The use of filters (e.g., the sliding average filter) is an effective passive way to mask sensor noise. A fusion can also be considered between sensors pointing to the same state variable, allowing to eliminate measurement noise and to reduce erroneous measurements. The Kalman Filter (KF) is often used for this purpose. The Extended Kalman Filter (EKF) is its simplest form for nonlinear models, but it remains less efficient than the Unscented Kalman Filter (UKF) which requires more computing resources ([3]) and than particle filters ([162]).

Passive redundancy can also be translated in hardware. For over-actuated robots, an actuator failing may not have a significant impact. In some cases, the trajectory can be maintained using the functional motors and without changing the motor configuration. Passive redundancy is therefore a widely used solution that provides flexibility despite the high material and financial cost.

2.3.5 Evaluating the efficiency of a fault tolerant system

When adding a fault tolerance mechanism in a computing system, there is an introduction of additional potential faults ([7]). Respectively noting $P_{FT}\{fault\}$ and $P_{NFT}\{fault\}$ the probabilities of a fault occurring in a system with and without a fault tolerance mechanism, we have $P_{FT}\{fault\} > P_{NFT}\{fault\}$. The authors of [7] affirm that the probability of having an error knowing a fault $P\{error/fault\}$ also increases with a fault tolerance mechanism. As a consequence, defining dependability as in Eq 2.2, a fault tolerant system with improved dependability should respect the assertion $P_{FT}\{failure/error\} << P_{NFT}\{failure/error\}$.

$$Dependability = 1 - P\{failure/error\} \times P\{error/fault\} \times P\{fault\} \tag{2.2}$$

The coverage of a fault tolerant process is defined as the probability of the system recovery given that a fault exists ([131]). More generally, it is considered as "the measure of effectiveness of any given fault tolerance technique" ([14]). Nevertheless, the effectiveness of a fault tolerant mechanism is a subjective quantity.

For example, [60] defines the effectiveness of a fault tolerant mechanism taking into account the fault tolerance rating f and the performance/cost rating p:

$$efficiency = k_1(f)^2 + k_2(p)^2 \tag{2.3}$$

The fault tolerance rating is estimated as the number of subsystem failures tolerable out of the number of available subsystems. The performance rating is calculated as the sum of the speed performance rating, the recovery time rating and the cost of incorporating fault tolerance.

For more classical studies of reliability estimation of autonomous systems, tools such as Mean time between failure (MTBF) or mean time to repair (MTTR) are generally employed to characterise the dependability of a system ([24], [53]), for instance by comparing MTBF with and without the FT mechanism.

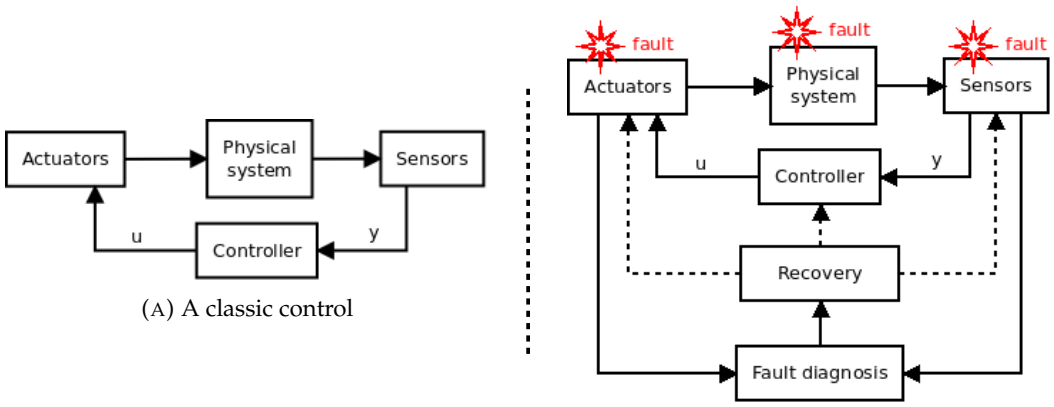
The notion of resilience is a very similar concept to fault tolerance. In mechanics, resilience is the ability of a material object to withstand shocks. In software engineering, the notion of resilience of a robot stands for recovering its original function after a partial damage ([182], [42]).

After defining dependability terminology, the next section focuses on how fault tolerance can contribute to the success of autonomous missions using mobile robots.

2.4 Autonomy and fault tolerance in robotics

In this section, we will address how fault tolerance is implemented on robots in the field during autonomous missions. The integration of fault tolerance in a robotic architecture is shown Fig 2.11. External faults may affect actuators, sensors or the physical system. Some papers also consider internal faults in the controller ([43]).

Since an open environment is mostly unknown, decisions made by an autonomous mobile robot can lead to dangerous situations. Moreover, it is impossible to prevent all faults from occurring. It is therefore necessary to implement fault tolerance to



(A) A classic control

(B) A generic fault tolerant control

FIGURE 2.11: Fault tolerant control of an autonomous mobile robot

prevent catastrophic system failures and to continue the mission despite the occurrence of some adverse events.

The success of an autonomous mission requires ensuring the safety of the robotic system (integrity) and its environment, achieving performance objectives while maintaining operational control loop mechanisms ([29], [78], [89]). We analyze in the following sections how fault tolerance helps to manage autonomy.

2.4.1 Managing safety with fault tolerance

Safety includes the preservation of the robot and its environment (harmlessness). A general approach that we find in several types of systems consists in designing a fault management controller, if possible physically separated from the main controller. The independence of the fault tolerance mechanism has the advantage assuring safety properties even in case of critical failure of the main controller. For example, [85] introduces the notion of safety-bag, which is an independent safety component that intercepts the actions to be performed by a user or a component and checks their validity according to previously established safety rules.

Depending on the context and the scope of the system, we will see that other approaches can be taken.

2.4.1.1 Safety in critical systems

Robots working in critical domains must be particularly safe, because the consequences of a safety defect could cause catastrophic consequences. This is particularly true in the medical, nuclear and avionics sectors, where the human factor is at the heart of safety issues.

[167] points out that medical robot must be as light as possible and consume little power so that they do not endanger patients. In the medical field, the use of a safe controller to limit the risk of collision with a human is of paramount importance ([111]). In the nuclear or space fields, robots are designed to be resistant to environmental radiation, which can damage components or cause bit flips in the software. These means of prevention on their own are nevertheless insufficient to ensure an "acceptable" security.

In these areas, fault tolerance can be achieved using hardware redundancy ([179]) despite the resulting higher cost or by implementing emergency safety procedures. A reliable remote agent can also be used to improve dependability. For example,

[114] introduced a Remote Agent equipped with a MIR (Mode Identification and Reconfiguration) that can detect a deviation from the optimal state of the spacecraft and reconfigure it by finding a sequence of commands that restores lost functionality or repairs failed components.

2.4.1.2 Safety in industrial robotics

In conventional industry, safety is guaranteed by the precise knowledge of the environment and the implementation of physical or immaterial barriers between the robot and potential workers ([56]). In such repetitive and controlled environments, fault prevention, fault forecasting and fault removal are predominant over fault tolerance to ensure safety.

Many industrial robots are able to check that their joint positions, joint speeds and estimated torques are within an acceptable range. Standards or technical specifications indicate for example that in the presence of an operator in the proximity, the cartesian speed of the end effector should not exceed 0.25m/s. Monitoring joint torques, energies, velocities or momentum is also an effective way to track collisions ([58]). If a collision is detected, the preferred solution is usually an emergency stop.

In his survey, [20] note that despite a multitude of approaches regarding robotics (platforms, openness, robot cardinality, levels of abstraction, research contribution), there is a lack of rigor and industrial relevance in research, because the applications carried out in laboratories are generally not particularly applicable in the context of industry.

2.4.1.3 Safety in mobile robotics

Unlike industrial robotics, mobile robots require autonomous decision-making in an uncertain environment. For harsh environments, such as the marine environment ([104]), Environmental Survey Hazard Analysis (ESHA) intends to identify potential hazards that may be associated with the environments or another agent ([34]).

Underwater robotics must meet specific safety constraints related to the environment. The pressure of water on the enclosure and the incompatibility of water and electronics, for example, require special measures to be taken in case of a leak. [122] have designed an architecture that takes this characteristic into account and proposes an AUV with a standard mode for nominal operation and an emergency mode in case of leakage whose solution is to surface.

We also find more sophisticated architectures to deal with security threats. For example, [175] propose a fault-tolerant system including subsystem risk analysis to improve the safety of an underwater robot and using a fault tree model of the robot integrating sensor, actuator, and operating state information. The risk level of the robot is analyzed and evaluated based on adaptive learning and fuzzy inference and a decision can be made to ensure the safety of the robot.

Collision with the environment or an obstacle is a common failure in mobile robotics. They can be detected by the IMU (Inertial Measurement Unit) or contact sensors, and anticipated by distance sensors, e.g. sensors measuring the distances between the robot and the surrounding obstacles. Collision responses generally vary depending on the context, from a speed reduction to an emergency stop. Safety rules using domain-specific languages can be generated in case of altered perception of safety sensors ([70]). Many laboratories are proposing robotic architectures that incorporate functionality to respond to this particular area of safety. For example, [48]

presents a ROV architecture performing inspections and navigating to defined positions while avoiding obstacles. In its architecture for a mobile robot in a known environment, [78] verifies that the energy of a potential collision remains below 4 joules.

The improper location of robots can also lead to catastrophic events. [43] proposes a fault tolerant architecture for a mobile robot that can detect undesired events, including its presence in a forbidden area. If two events are detected at the same time, the authors use the combination of a fault tree and a decision tree to decide which recovery should be initiated. Progress has also been made in recent decades on Unmanned Aerial Vehicles (UAVs), which are now available to the general public and can perform inspection tasks in a wide range of areas ([118]). Drone manufacturers may also include features to prevent UAVs from going into restricted areas or to return to a home position if communication with the user is lost.

The lack of residual energy is also a recurrent problem and can pose safety issues ([78]), especially for aerial and underwater robots. Simplistic approaches continuously check the voltage at the robot's battery terminals to ensure that this voltage does not drop below a threshold ([64]). The associated reaction may then be to prevent further motion or to return to a safe position if the remaining energy allows. Despite the possibility of estimating the energy consumption of a desired motion robot ([133]), only few works take into account the energy in mission management ([88], [78]).

Little research has been done to check for inconsistencies between actions that may affect system safety. We can cite, for example, the work of [94] which describes a technique called STPA (System-Theoretic Process Analysis). This technique consists in identifying the potential control sequence of the system that could lead to a dangerous state and in identifying the conflicts leading to a violation of the security constraints.

Work on improving the safety of robotics, and particularly mobile robotics, is generally effective within its domain of validity but does not necessarily take into account mission constraints in parallel, which we will see in the next section.

2.4.2 Managing mission execution with fault tolerance

In this section, we focus on how fault-tolerant robotic architectures manage the mission in the presence of faults. As seen in section 2.3.3, it is sometimes necessary to modify or interrupt the mission in the presence of faults or failures. We have identified that the 3 main categories of recovery are execution control, mission replanning or autonomy adjustment.

Execution control supervises the execution of each of the actions necessary to the implementation of a specified task. During a recovery of this nature, the mission retains the same sequence of tasks, but modifications can be made locally. The mission does not change from an external point of view if the system finds another resource (functional redundancy), which is therefore an option to prioritize over other recoveries if possible. For example, [88] work on an architecture for mobile robots that identifies a new resource allocation if mission performance cannot be achieved with the current configuration.

In some cases, the execution controller has to modify the parameters of the mission to preserve the safety of the robot or its environment, or because the mission can no longer be executed in an optimal way ([75]). This consists for example in modifying the trajectory temporarily to avoid obstacles, to decrease the speed of the robot to reduce the risk of violent collisions, or to increase the speed of the robot to validate temporal performance criteria to respect the mission specifications.

Replanning is generally used when execution control is no longer possible. The sequence of robotic tasks changes, either in order to accomplish the mission objective or to preserve the robot's integrity. Several works are interested in replanning. For example, [98] develop an architecture that uses the LAAS IxTeT planner. The detection principles are based on watchdogs, a plan analyzer and online goal checking. Two recovery mechanisms are used: successive planning where the planner is changed each time an error is detected and distributed planning where alternatives are executed simultaneously.

Replannification can also be expressed spatially, as computing new paths is common practice in mobile robotics. For example, [66] describe a reconfiguration of mobile robot in case of a detected fault. If the path is no longer reachable, a path planner calculates a new reachable path. [116] proposes an architecture that uses an external supervision system that checks the mission characteristics with the potential obstacle on the path and the possible failure of the system components.

When an operator is available, the autonomy adjustment is effective to ensure a safe transition to continue the mission later, or to return the robot to a safe state. For example, [39] propose a fault tolerant architecture in which a FMECA is established to identify failures to avoid. After the detection of faults, the diagnosis process is based on incidence matrices. Depending on the criticality of the failures, the robot can switch to different levels of autonomy.

In summary, when the safety criteria allow it, the fault tolerant robotic architectures that manage the mission generally focus on limiting the performance loss.

2.4.3 Managing control loop with fault tolerance

Control loop functionality includes the ability of the robot to execute the loop: sense, compute the command, and operate the actuators. A compromised control loop can impact safety or mission performance over time and depending on the severity of the damage.

In this section, we will first look at component-based architectures before addressing the management of the control loop in the presence of faults.

2.4.3.1 Example of component-based fault tolerant architecture

Several works deal with component-based fault-tolerant architectures. The works of [92] propose an architecture defined by skills able to execute an action and to check for errors when generating actions. [2] work on a framework for a component-based system built on the OPRoS platform using a priori defined rules in which a fault manager is able to choose the task and resources to allocate based on the detected error. [83] propose a framework that uses a directed graph of a mobile robot and learns a model of the functional mapping. A metric is provided by the user that helps detect a change in the mapping and a likely fault. [54] developed a monitoring system where each received and sent component message is checked and compared

to a specified behavior. An error emission or time constraint violation is collected for maintenance purpose.

2.4.3.2 Sensor management with fault tolerance

Sensors are key devices for sensing the environment and estimating the robot's state. Two types of failures are possible for sensors:

- sensor degradation: the sensor does not measure the physical phenomenon accurately. Approaches often model such a sensor with a bias or a high measurement noise.
- sensor out of order: the sensor does not provide measurements or the measured values are erratic.

Basic approaches use banks of residuals to diagnose the faulty sensor. For example, [140] use multiple model Kalman filters to produce residuals to diagnose a faulty sensor on the Pioneer I. **[lu_multi-localization_2017]** presents a method to detect and diagnose sensor faults on a Pioneer 3-DX. The authors use residual from the combinations of the different sensors. They validate the method by comparing the estimate produced with a ground truth obtained by an external camera.

[82] presents a fault detection mechanism for sensors using the data collected over time and performing a likelihood check. Offline correlation detection is performed to compare the actual patterns with the expected ones online.

Recovery generally consists in isolating the faulty sensor, reconfiguring the system and continuing the mission if redundancy is available. For sensor degradation, if the fault has been clearly identified, another approach is to reconfigure the sensor to eliminate the potential bias or noise.

2.4.3.3 Control management with fault tolerance

The goal of fault tolerant control is to maintain the control stability of a system despite the presence of erroneous inputs, while keeping its performance as close as possible to the nominal performance. We have noted previously that a first detection of an erroneous command can be carried out by a consistency check. For example, [43] checks the coherence of the commands of its robot in order not to enter a prohibited zone.

More generally, there are various reconfiguration mechanisms for control ([183]):

- Optimization methods: Linear Quadratic (LQ), H_∞, Linear Matrix Inequality (LMI), Model predictive control (MPC).
- Switching methods: Multiple Model (MM), Gain Scheduling (GS)/Linear Parameter Varying (LPV), Variable Structure Control (VSC)/Sliding Mode Control (SMC).
- Matching methods: Pseudo-inverse, Eigenstructure Assignment (EA).
- Following methods: Model Following (MF), Model predictive control (MPC).
- Compensation methods: additive compensation or adaptive compensation.

2.4.3.4 Actuator management with fault tolerance

Actuators are key devices to drive the robot. Their types differ depending on the vehicle and the environment. Many papers deal with actuators failure diagnosis and recovery, essentially based on the actuation configuration of the system.

Energy calculation-based methods are useful for monitoring actuator failures. [181] propose to identify the degree of actuator failure by checking the actuator control voltage. [32] propose a method for detecting faults on underwater robot actuators based on energy consumption whereas other methods use more classical approaches already discussed in this thesis. For example, [99] use a nonlinear observer to generate residuals to diagnose faults on several sensors and actuators.

The reaction to an actuator failure depends on which actuator is affected. [46] identify that, in an antagonist system, the same motor failure mode may produce different effects on the system depending on the configuration of motors and therefore the triggered reaction should be different. For mobile robots, we identify two types of actuator failures:

- actuator degradation: the actuator is still functional but its behavior is altered. The diagnosis calculates the new characteristics of the actuator to regain an efficient control. For example, [134] propose a recovery of an actuator efficiency loss for a quadrirotor aerial vehicle.
- actuator out of order: the actuator is not functional. A failed actuator can be removed from the model but the mission is not necessarily lost depending on the redundancy left. For example, [143] present a fault-tolerant control for an octorotor vehicle and use compensation when an actuator fails. [72] describe an intelligent monitoring and emergency system (IMES) that performs a knowledge based diagnosis and test it on an AUV. The IMES can perform actions such as removing an actuator from the model, changing task parameters or changing the task itself.

The approaches detailed in this section target fault tolerance problems around safety, mission management and control loop aspects. Even if these 3 aspects are independently addressed in the state of the art, the implementation in robotic architectures does not generally offer joint perspectives around these 3 notions.

2.5 Conclusion

In this chapter, we first introduced the main current mobile robot architectures. We then observed that a mobile robot can be subject to multiple faults and failures during its development and operational life cycle. There are means to prevent, forecast and remove a various number of faults in the system.

Nevertheless, in order to ensure the dependability of mobile robots, we have established the need to employ fault tolerance to react in real time to the various undesired events that may occur. The fault-tolerance approaches ensure system dependability through undesirable event detection and recovery mechanisms. They are often efficient in context as they prevent catastrophic events from occurring, and often allow robots to continue their mission to the extent possible.

The most common and evident solution to achieve dependability is to increase the reliability and the redundancy of the system components. However, the introduction of more reliable components or redundant components results in an overall increase of resources usage like the volume needed, the energy consumed and therefore the energy autonomy. All these factors play a role in the choice of system design, which is often the result of a trade-off between cost and dependability. The dependability issues are therefore even more present in low-cost systems, whose components fail more frequently and provide lower quality services.

The dependability problem is also central for the management of complex missions, for which the context may raise difficulties. When validating fault-tolerant systems, work often assumes the single-fault hypothesis and proceeds to a single recovery at a time, which is a strong simplistic assumption that is often poorly representative of reality. Currently, these difficulties are often overcome by human intervention, thus revealing shortcomings in achieving fully autonomous missions.

Furthermore, there is no conjoint mission and safety handling, which is a limitation for a complete dependability. Fault tolerance architectures may also be designed to provide fault-tolerant control at a lower level without having a clear perspective on the mission and safety consequences. In addition, there is limited verification of the consistency of recoveries. The lack of compatibility analysis between recovery mechanisms at different levels can have consequences for robot safety and mission performances.

Finally, existing fault-tolerance approaches also lack adaptability and genericity, as their implementations are often specific to the application and robotic architecture. This lead to difficulties in adapting these fault tolerance implementations to other contexts, resulting in additional costs and unreliable results.

To address these points, we believe that it is necessary to add software-based, fault-tolerant and versatile decision making to improve the dependability of a robotic mission. We therefore propose in the next chapter a new fault-tolerant framework for autonomous robots, that we developed during this thesis. In addition to addressing mission and safety constraints, the proposed framework also takes into account adverse events occurring in the control loop, that may lead to the violation of these higher-level constraints.

Chapter 3

Proposal of a multi-level fault tolerant framework

Contents

3.1 Fault tolerant framework overview 37
3.2 Identification and classification of undesired events 39
3.2.1 Guidelines for building the mission fault tree 40
3.2.2 Guidelines for building the safety fault tree 42
3.2.3 Guidelines for building the control loop fault tree 42
3.3 Identification of system's weaknesses 45
3.4 Implementation and execution of event detection and inference . 45
3.4.1 Implementation and execution of undesired event detection 46
3.4.2 Execution of event inference 48
3.5 Implementation and execution of recoveries 51
3.5.1 Implementation and execution of local recoveries 51
3.5.2 Implementation and execution of decisional recoveries . . . 57
3.6 Conclusion 59

The previous chapter highlights that there is no generic approach that can handle failures occurring in the system and, at the same time, that can manage the constraint violations of safety or mission constraints for autonomous mobile robots. In this chapter, we propose a multi-level framework that considers these undesired events through different phases of the system design to provide a more efficient fault tolerant solution.

We will first give a general presentation of the proposed framework, before addressing the identification of Undesired Events (UEs) using fault tree construction. We will also address the identification of system's weaknesses. We will then focus on the mechanisms of detection, inference and recovery before concluding.

3.1 Fault tolerant framework overview

In this section, we present the fault tolerant framework that we propose to improve the dependability of mobile robots and we describe its main steps presented Fig 3.1. We assume that the mobile robotics system has a mission manager, modules/components performing the control loop and a fault tolerance manager implementing our framework.

The process starts with the design phase, which follows a process repeated in a loop in which:

- **Identification and classification of undesired events**: we identify the different undesired events that can occur in the system. To accomplish this, we first list the major control loop failures, mission and safety constraint violations, that we identify as the main categories of threats to any autonomous mobile robot. This work is carried out from the knowledge of the robotic system for the control loop aspect, the environmental context for the safety aspect and the mission specification for the mission aspect. The notion of constraint violation refers to a quantifiable deviation of the system's behavior on an abstract level related to the mission or the environmental context. We therefore refer to it as a **high-level undesired event** as opposed to a failure/error/fault occurring in the system which we qualify as a **low-level undesired event**. We then build Fault Trees (FTs) to list the potential causes of the previously identified undesired events. The fault trees are expressed in a file which will be interpreted during the running phase.

- **Identification of system's weaknesses**: we identify the weaknesses of our mobile robot. We have seen in the previous chapter that fault tree analysis allow to identify the weaknesses of a system with both a qualitative and a quantitative approach. With a thorough knowledge of the system, it is therefore possible to estimate the probabilities of occurrence of the various basic events. However, it is difficult to estimate the probability of occurrence of a high-level constraint violation. We must then concentrate on the study of the minimum cut sets of the fault trees and rely on rough estimates of probabilities of occurrence of events, associated with their severity, based on an expert's perspective.

- **System enhancement**: based on the identification of the system weaknesses, we implement event detection and local recovery mechanisms to avoid the occurrence of a more critical event. We introduce specific recoveries to handle undesired events locally based on redundancy management. The introduction of this redundancy leads to a modification of the fault trees.

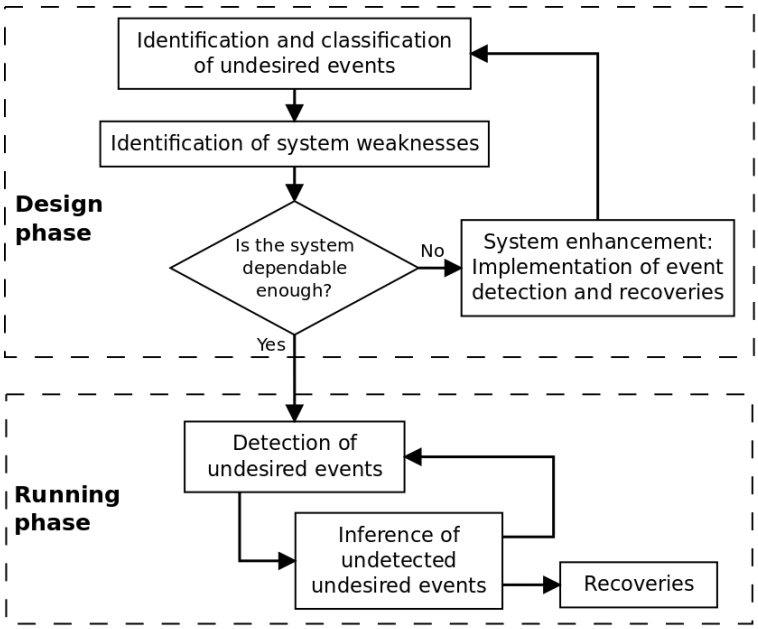


FIGURE 3.1: Overview of the fault tolerant framework

The design phase is completed when the expert considers that the system is dependable enough. In general, we can consider that the system is sufficiently dependable if there are no events whose criticality (frequency of occurrence $\times$ severity) is higher than a threshold, defined in agreement between the different stakeholders of the project.

The running phase, during which fault trees are used to perform runtime fault tolerance, also follows a repeated loop process in which a Fault Tolerance Manager (FTM) executes:

- **Detection of undesired events**: runs the implemented detection algorithms to identify the potential occurrence of the concerned events.
- **Inference of undetected undesired events**: infers events that could not be detected based on the logic inherent to fault trees. The inference mechanism thus provides potential causes (diagnosis) or consequences (deduction) of the detected events.
- **Recoveries**: allows the initiation of recoveries to ensure acceptable robot behavior despite the presence of these events. We distinguish two types of recoveries: **Local Recoveries** (LRs) which aim at managing events with local resources (without changing the robotic task) and **Decisional Recoveries** (DRs) which imply a change of robotic task at a higher level.

This framework is deployed in a control architecture of a mobile robot to improve its dependability. We now explain in detail different steps of the framework, starting in the following section with the construction of fault trees.

3.2 Identification and classification of undesired events

The implementation of a fault tolerant architecture first relies on establishing the different threats that the robot may face. In this work, we decide to group threats, consisting of mission or safety constraint violations or their sources (failures, errors and faults), under the common name of **Undesired Events** (UEs).

Among the fault forecasting approaches presented in the chapter 2, we choose to use fault trees to establish cause and effect relationships of undesired events. We choose this approach as it is widely used to estimate the weaknesses of critical systems. It also proposes a simple, logical and graphical way to link undesired events with logic gates. We build 3 fault trees:

- the mission fault tree contains the mission constraint violations and their direct causes.
- the safety fault tree contains the safety constraint violations and their direct causes.
- the control loop fault tree contains the undesired events occurring in the control loop of the system.

Some undesired events of the control loop fault tree can lead to undesired events in the other faults trees, but this highly depends on the context and the events in question. Thus, to link the control loop fault tree with the two other fault trees, it would be necessary to have a different instance of the control loop fault tree with

different parameters for each undesired event of the mission and safety fault trees. We prefer to keep the control loop fault tree independent from the 2 other fault trees to simplify the framework, as it may be complex to logically link the control loop undesired events with the high level constraint violations.

We propose here to detail a fault tree construction guideline for the 3 fault trees (based on this guideline, the fault trees of our case study will be explicitly described in chapter 4).

3.2.1 Guidelines for building the mission fault tree

In this section, we focus on building the fault tree composed of the mission related UEs. We first address the identification of constraint violations related to the mission before studying the construction of the fault tree itself.

3.2.1.1 Identification of mission constraint violations

A mission constraint violation is a violation of a mission objective or sub-objective stated in the specifications. The sub-objectives are required to achieve the mission objectives but are not necessarily defined by the user. Objectives or sub-objectives are characterized by properties that logically or mathematically express the constraints. The properties must be determined exhaustively to take into account all the mission constraints that the robot must respect. For example, noting $pos = [x, y, z]^T$ the cartesian position of the robot in the North-East-Down (NED) frame at a time t, a property that imposes the robot to be at a target position pos_{target} between a time $t1$ and a time $t2$ can be expressed by:

$$\forall t \in [t1, t2], |pos(t) - pos_{target}| < threshold \tag{3.1}$$

The threshold is here a positive integer which characterizes the level of requirement of the property. The violation of the mission constraint, i.e. the violation of the property expressed in eq 3.1 is therefore logically expressed as:

$$\exists t \in [t1, t2], |pos(t) - pos_{target}| >= threshold \tag{3.2}$$

The choice of mission objectives and properties must be carried out as exhaustive as possible, in agreement with the different stakeholders of the project. The ultimate user usually provides the mission objectives that the robot should accomplish, and then the designer validates, invalidates, or refines these objectives for example by modifying the thresholds associated with the properties or breaking down objectives in sub-objectives.

For mobile autonomous robots, broad classes of properties are often considered to qualify mission objectives or sub-objectives:

- spatial properties: these properties affect the spatial characteristics and impose, for example, requirements on the position or attitude of the robot or of its potential end-effector. They may also require a minimal accuracy in the estimation of the position of the robot.
- temporal properties: these properties affect the temporal characteristics and impose, for example, requirements on the execution time of a robotic task.

- energetic properties: these properties affect the energy characteristics and impose, for example, requirements on the electrical energy to be kept to perform the mission.
- informational properties: these properties affect the informational characteristics and impose, for example, requirements on the processing result of an algorithm running in the robot controller.

Naturally, we can also consider combinations of these property classes. Thus, a property implying that the speed of the robot is within a range of values has both spatial and temporal characteristics.

3.2.1.2 Construction of the mission fault tree

We divide robotic missions into several phases called Mission Phases (MPs), executed consecutively to achieve the objectives. Handled by a mission manager at a decisional level, these missions phases are constituted by one or several robotic tasks. A mission objective or sub-objective is generally related to one or more mission phases and is not necessarily desired throughout the mission.

As stated in section 2.2.3.2, fault trees are generally built from the identification of a feared failure and are developed up to the basic events, potential faults of this failure. In our approach, we consider violations of mission constraints as equivalent to failures and we develop in a classical way several fault trees from these constraint violations with the help of the developer's expertise. In order to simplify the final format of the result, we group these different fault trees by mission phases under an OR gate. Finally, we group the set obtained under another OR gate in order to get a single fault tree, composed of several sub-fault trees (Fig. 3.2).

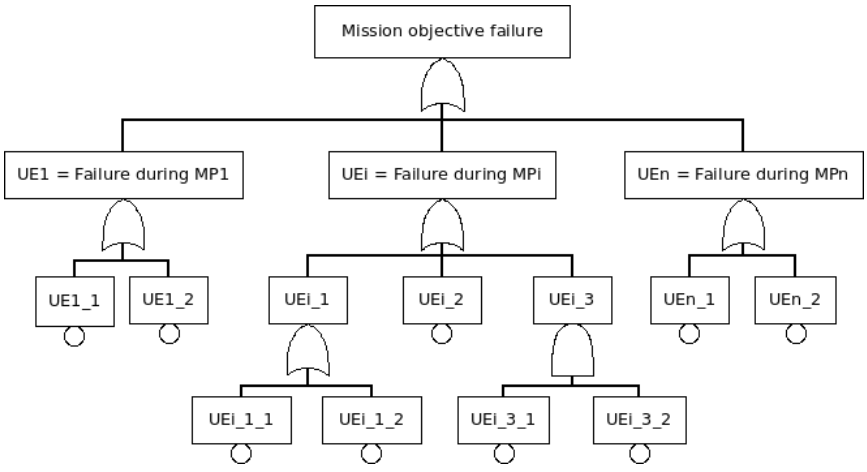


FIGURE 3.2: A standard mission FT

In Fig. 3.2, the mission constraint violations are represented as UEi_j with $\{i, j\}$ natural integers. It is possible that a constraint violation cannot be expressed by a logical and direct combination of events. In this case, it will be expressed by an undeveloped event or a basic event (UE1_1 for example).

The method of the mission fault tree construction being detailed, we will focus on the construction of the safety fault tree in the next section.

3.2.2 Guidelines for building the safety fault tree

For critical embedded systems, the safety aspect is often addressed with fault trees ([154]). We choose to express the fault tree top event as a constraint violation, which is useful to express safety undesired outcomes. In our case, a safety expert determines safety constraint violations, mostly related to the system and the environmental context. These safety constraints must ensure the integrity of the robot and of its environment.

3.2.2.1 Identification of safety constraint violations

The first factor to consider for safety is human in the proximity of a robot, as this situation induces risk with high degree of severity. In order to be as exhaustive as possible, it is advisable to imagine the possible damages to the human by hazard category: physical hazards (e.g. winding, entanglement, dragging, shock, crushing, cutting, pinching, shearing, stinging, projection), burning, chemical burning, electrocution, poisoning, contamination.... The potential hazards are identified by asking the question: to what extent can the robot injure a person? It is also important to consider only probable hazards. It is for instance possible to consider collisions between a person and the robot, or pinching/jamming between a person's extremities and an actuator.

The second factor to consider is the integrity of the robot itself. The integrity of the robot can be compromised by its loss or by several types of degradation such as mechanical (e.g. shock, vibration, fatigue), electronics (e.g. short circuit, high intensity, broken component), thermal, chemical, software (e.g. bit flip) degradation. For example, in a marine environment, it is essential to take into account water leaks to avoid water damage to the system.

The third factor is the degradation of the environment (excluding humans) which can take several forms: physical, chemical, radioactive, degradation.... For example, it is imperative that an underwater robot does not damage nearby corals.

3.2.2.2 Construction of the safety fault tree

Similarly to the mission fault tree, the causes of the identified top events must be deduced thanks to the expertise of the robot designers following the classic steps of fault tree construction. This results in several fault trees related to the safety of the robot. These fault trees are then grouped under an OR gate into a single safety fault tree. If the causes of a safety constraint violation are not identified, we consider this event as a non-developed or basic event.

We will finish by addressing the construction of the fault tree related to the control loop in the next section.

3.2.3 Guidelines for building the control loop fault tree

We now desire to identify the various undesired events in the control loop of the robot as, they may lead to short or long term negative consequences on the mission or the safety of the robot.

3.2.3.1 Modeling of the control loop

We have noted in the chapter 2 that a control loop is divided into several steps: sense, plan, act. Each of these actions is performed by one or more modules that interact with each other (e.g. devices, programs, threads). A module can for example process data from a sensor, estimate the position of the robot or calculate the command to perform a motion. Fig 3.3a represents an example of a system in which the modules C and D perceive the environment and generate respectively the outputs C_o and D_o (modeled by arrows). These outputs are seen as the inputs B_i1 and B_i2 by the module B which itself generates the output B_o. Finally, perceiving B_o as A_i, the module A acts on the environment.

It is challenging to build the fault tree of the control loop because it must reflect the hardware and software complexity of the system. A systematized approach would therefore be valuable.

3.2.3.2 Construction of the control loop fault tree

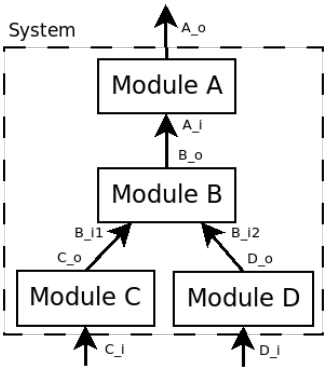
In the previous chapter, we identified that tools using AADL or HiP-HOPS system description software allow to build and analyze fault trees from the schematic block description of a system. But these tools require a substantial effort to describe the block architecture. In our context, we can achieve similar results using a method inspired by the existing ones: we build the control loop fault tree using the modularity of the control loop of the robot. We choose to build a sub-fault tree for each module by considering the top event as a failure of the module output (Fig 3.3b).

Within the sub-fault tree representing the failure of a module output, we first differentiate an internal error (the cause is attributed to a fault internal to the module) and an external error (the cause is attributed to a fault external to the module). We develop the external error by assuming it can come from an invalid input (data, sensor measurement...) or from a power supply failure of the module. We develop then if the error comes from an invalid input: we assume the error can come from the invalid output of the previous module, or from a connection breakdown between the modules. This could be explained by a defective wire (if the modules are on different devices) or an error in the communication protocol.

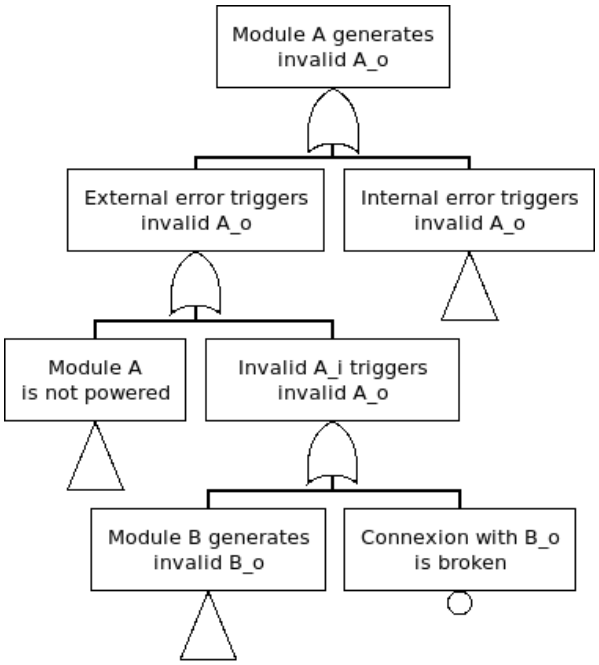
We develop in Fig 3.3c the internal error according to the failure domain of the output ([14]), which can be classified as below:

- **output content failure**: the content of the output deviates from the nominal content.
- **output timing failure**: the time of arrival or the duration of the output deviates from the nominal timing.
- **output halt failure**: the output generation is halted (both content and timing failures). This failure may occur if the module itself is out of order.
- **output erratic failure**: the output is delivered but there is content and timing failures.

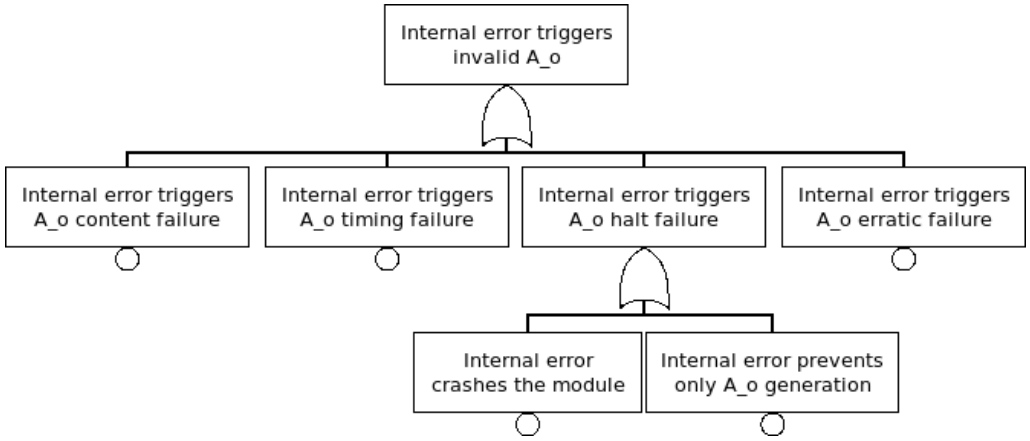
We then link the sub-fault trees together to build a single control loop fault tree. For that, we make the assumption that the failure of a module output leads to an error of an associated module input. Since the goal of the control loop is to drive



(A) Structure of our control loop architecture



(B) Sub-fault tree related to invalid output of module A



(C) Our classification of internal errors

FIGURE 3.3: An example of a system defined by module and its associated fault tree

the robot in its environment, the top event of the resulting fault tree is an invalid actuation.

This method of the fault tree construction associated with the control loop allows to give a systematic general form to the fault tree. However, it may be necessary to complete the structure to take into account the specificity of each module. For instance, a module can manipulate redundant inputs and generates a fusion as an output. In that case, the expert in charge of designing the fault tree can insert an AND gate at the relevant location to represent the redundancy. In addition, the expert has the possibility to enrich the representation of the control loop fault tree by expending specific undesired events related to a module.

In the next section, we will address the next steps of our framework: the identification of the robot's weaknesses and the system enhancement.

3.3 Identification of system's weaknesses

In this section, we will discuss the identification of system weaknesses following the construction of our fault trees. As stated in chapter 2, the fault trees are useful to determine the weaknesses of a system, using quantitative or qualitative analysis.

Fault Tree Analysis allows first to study the Minimal Cut Sets (MCSs) to identify which combinations of events are likely to cause the top event (qualitative analysis). The MCSs of order n (n positive integer) correspond to the MCSs composed of n undesired events, i.e. the occurrence of these n events is sufficient to produce the top event. Among these MCSs, it is therefore appropriate to pay particular attention to the low-order MCSs.

Fault Tree Analysis usually allows to estimate the probability of occurrence of top event with a sufficient knowledge of the probabilities of occurrence of basic events (quantitative analysis). But this approach is not applicable to systems which components have not been sufficiently tested, or more generally in cases where this probability is not measurable. That is the case when the context strongly influences the occurrence of the basic events. In new and low-cost systems, it is difficult to accurately estimate the probability of an event since extensive tests are not especially performed before the conception of the system.

In addition, the severity of the undesired events has also to be considered, as some events might be totally unacceptable under any circumstances, as for instance, events involving serious human injuries. An analysis of fault trees alone does not allow to estimate the severity of the undesired events. This estimation is usually performed with a more classical risk analysis, like FMECA, before building the fault trees.

Therefore, there is a choice to make according to the means at disposal and the trade-off between cost and dependability. By taking into account an estimation of the probability of occurrence of the undesired events and their associated severity, an expert decides whether a part of the system should be enhanced.

In the next section, we will address how to implement and execute the two first phases of our framework: event detection and inference.

3.4 Implementation and execution of event detection and inference

From the identified weaknesses of the system, the expert will target hardware and software implementations to improve the system. These implementations may take several forms:

- improve the reliability of the hardware and software. This can be done by replacing unreliable components with more reliable ones. Unfortunately this method can be complicated to implement for a low-cost mobile robot, because it requires more financial resources, space and power for hardware components. Improving the software reliability is complicated, time consuming and not always guaranteed. The visible result on the fault trees is the decrease of the probability of occurrence of UEs.
- add software and hardware passive redundancy (compensation). Again, this method may require a substantial financial, space and power contribution. The

visible result on the fault trees is the introduction of AND gates modeling the redundancy of the components.

- set up specific actions to remove an event that occurs or to stop the propagation of an error. This requires the implementation of event detection and the implementation of recoveries. In our framework, the different actions that the robot can perform are divided into two categories: Local Recoveries (LRs) and Decisional Recoveries (DRs). The execution of event detection, inference and LRs are performed on an independent module from the classic control loop called Fault Tolerance Manager (FTM).

In order to perform recoveries, we will first see how we detect UEs before addressing the inference step to target appropriate recoveries.

3.4.1 Implementation and execution of undesired event detection

In this section we will address the implementation phase and the execution phase of detection mechanisms.

3.4.1.1 Implementation of undesired event detection

Notations We note $UE_{express}$ the set of undesired events expressed in the fault trees. The detection of an Undesired Event (UE) expressed in a fault tree allows to check if this UE is present (true) or not (false). The implementation of UE detection depends on the UEs itself and the available means of detection. The detection can be performed on constraint violations, intermediate events or basic events. We assume only a subset of $UE_{express}$ can be detected, and we call this detectable subset UE_{detect}. Similarly, we denote $UE_{\overline{detect}}$ as the complementary of UE_{detect} in $UE_{express}$:

$$\begin{cases} UE_{detect} \cup UE_{\overline{detect}} = UE_{express} \\ UE_{detect} \cap UE_{\overline{detect}} = \emptyset \end{cases} \tag{3.3}$$

Main categories of detection processes A constraint violation can be detected at runtime with the resources at disposal by checking the associated property assertion. For example, the detection algorithm of the violation of a mission constraint which stipulates that *the robot should stay close to the optimal mission path* takes a simple form. Indeed, noting δd the distance between the robot and the optimal mission path and δd_{max} the maximal distance tolerated, this mission constraint violation is detected as true if $\delta d > \delta d_{max}$.

We also propose a guideline to detect inappropriate outputs in the control loop fault tree. Thus, in accordance with the classification described in section 3.2.3.2:

- output content failure detection: this can be verified firstly by checking that the output of the module is in a specific range. This detection is performed directly in the module itself to avoid direct error propagation. Secondly, with a freeze check, the algorithm verifies whether the output remains unchanged for a time or for a predefined number of iterations. This time depends on the nominal period of the module output and the expected average variation of the output between successive iterations. Thirdly, if the output is characterized by a confidence value, the algorithm checks whether the confidence of the

output is adequately confident with a confidence check. Finally, when redundancy is available in sufficient quantity, a comparison with other sources or the calculation of residuals can also qualify whether the output is faulty.

- output timing failure detection: a timing failure can be checked by comparing whether the output generation time is properly synchronized, i.e. if it is within a specific range.
- output halt failure detection: this failure may occur if the module itself is out of order. To detect a halt failure, the difference between the current time and the last time of the generation of the output must be greater than a threshold. An output halt failure can be caused by a crash of the module that generates the output. In this case, it is also possible to detect this phenomenon by placing a watchdog that monitors the execution of the module.
- output erratic failure detection: the detection of an output erratic failure uses the output content failure and output timing failure detection algorithms, but with different thresholds.

However, it is not always simple to implement fault detection mechanisms. First, the system must have adequate resources to detect an undesired event. Second, false detection (an event is false but it is detected as true) and non-detection (an event is true but it is detected as false) can occur. Thus, more reliable detection means have to be implemented. This can be done, for instance, by merging resources to obtain redundancy (average, sliding average, Kalman filter...) or by waiting for the data to stabilize over time to increase their confidence.

We will address in the next section how the FTM actually performs the detection of undesired events in runtime.

3.4.1.2 Detection of undesired events during runtime

During runtime, the detectability of undesired events, especially those related to the mission, is sometimes conditioned by the robotic task in progress. We consider that an undesired event that expresses the violation of a constraint linked to a mission phase should only be detectable during that mission phase. In other words, UE_{detect} is mission phase dependant.

The current set UE_{detect} is evaluated by the FTM at constant frequency using the implemented algorithms. The FTM identifies then UE_{detect_T}, composed of the events detected as true and UE_{detect_F}, composed of the events detected as false:

$$\begin{cases} UE_{detect_T} \cup UE_{detect_F} = UE_{detect} \\ UE_{detect_T} \cap UE_{detect_F} = \emptyset \end{cases} \tag{3.4}$$

At the end of this step, the FTM has identified the 3 complementary sets UE_{detect_T}, UE_{detect_F} and $UE_{\overline{detect}}$. We will address the next step of our framework in the next section: event inference using the fault tree logic.

3.4.2 Execution of event inference

The inference step is used to infer the state of undesired events in $UE_{\overline{express}}$ using the fault tree logic. The purpose of this step is to identify new sets of undesired events: $UE_{\overline{infer}}$, composed of the non-inferred events and UE_{infer}, composed of the inferred events. UE_{infer} is itself divided into 2 sub-sets: UE_{infer_T}, composed of the events inferred as true and UE_{infer_F}, composed of the events inferred as false:

$$\begin{cases} UE_{infer} \cup UE_{\overline{infer}} = UE_{express} \\ UE_{infer} \cap UE_{\overline{infer}} = \emptyset \end{cases} \tag{3.5}$$

$$\begin{cases} UE_{infer_T} \cup UE_{infer_F} = UE_{infer} \\ UE_{infer_T} \cap UE_{infer_F} = \emptyset \end{cases} \tag{3.6}$$

We will detail the inference algorithms in the following sections.

3.4.2.1 Basic inference using AND and OR gates

The inference algorithm uses an iterative approach based on the fault tree structure and the logic behind OR and AND gates. The inference of UEs can be carried out bottom-up, as a **deduction** to find the immediate consequences of UEs, or top-down, as a **diagnosis** to find the immediate causes of UEs. Nevertheless, fault tree construction often leads to the use of OR gates for non-redundant systems. In that case, inferring the original fault may be complicated, because there can be multiple plausible logic causes.

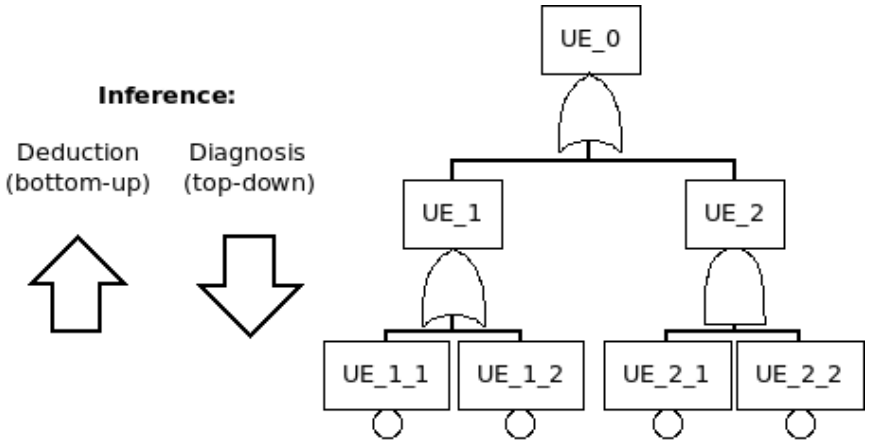


FIGURE 3.4: An example of fault tree

We assume for the moment that there is no conflict between inference and detection steps (we will remove this assumption later). We present below 8 Inference Rules (IRs) and propose to illustrate them on the basis of the fault tree presented Fig 3.4. These rules are established on the basis that the immediate causes of an undesired event are represented in the fault tree if they are necessary and sufficient to trigger the latter undesired event ([154]).

Deduction (bottom-up) through OR gate

IR1: the event associated to the gate is inferred as true if at least one of the downlink events is inferred or detected as true. In the example:
$UE_1 \in UE_{infer_T}$ if (UE_1_1 or UE_1_2) $\in (UE_{detect_T} \cup UE_{infer_T})$.

IR2: the event associated to the gate is inferred as false if all the downlink events are inferred or detected as false. In the example:
$UE_1 \in UE_{infer_F}$ if (UE_1_1 and UE_1_2) $\in (UE_{detect_F} \cup UE_{infer_F})$.

Deduction (bottom-up) through AND gate

IR3: the event associated to the gate is inferred as true if all the downlink events are inferred or detected as true. In the example:
$UE_2 \in UE_{infer_T}$ if (UE_2_1 and UE_2_2) $\in (UE_{detect_T} \cup UE_{infer_T})$.

IR4: the event associated to the gate is inferred as false if at least one of the downlink events is inferred or detected as false. In the example:
$UE_2 \in UE_{infer_F}$ if (UE_2_1 or UE_2_2) $\in (UE_{detect_F} \cup UE_{infer_F})$.

Diagnosis (top-down) through OR gate

IR5: if the event associated to the gate is detected or inferred as false, then the downlink events are all inferred as false. In the example:
(UE_1_1 and UE_1_2) $\in UE_{infer_F}$ if $UE_1 \in (UE_{detect_F} \cup UE_{infer_F})$.

IR6: if the event associated to the gate is detected or inferred as true and if the downlink events are all detected or inferred as false but one, then the latter event is inferred as true. In the example:
$UE_1_2 \in UE_{infer_T}$ if [$UE_1 \in (UE_{detect_T} \cup UE_{infer_T})$ and $UE_1_1 \in (UE_{detect_F} \cup UE_{infer_F})$].

Diagnosis (top-down) through AND gate

IR7: if the event associated to the gate is detected or inferred as true, then the downlink events are all inferred as true. In the example:
(UE_2_1 and UE_2_2) $\in UE_{infer_T}$ if $UE_2 \in (UE_{detect_T} \cup UE_{infer_T})$.

IR8: if the event associated to the gate is detected or inferred as false and if the downlink events are all detected or inferred as true but one, then the latter event is inferred as false. In the example:
$UE_2_2 \in UE_{infer_F}$ if [$UE_2 \in (UE_{detect_F} \cup UE_{infer_F})$ and $UE_2_1 \in (UE_{detect_T} \cup UE_{infer_T})$].

Hypothesis of partial knowledge under OR gates We make the hypothesis that the set of downlink events of an OR gate is only partially known, as an expert cannot imagine all possible combinations of UEs leading to another UE by their occurrence. To avoid inferring false-positive and false-negative UEs, we therefore remove IR2 and IR6 from the inference rules used in our framework.

However, we do not make this assumption for AND gates, because these gates reflect a stronger logic. We assume that the set of downlink UEs of an AND gate has been identified completely.

Hypothesis of conflict between inference and detection We now hypothesize that conflicts can occur between inference and detection for an undesired event. Conflicts between inference and detection of an UE can result in 3 forms:

- C1: an UE can be both in UE_{detect_T} and UE_{infer_F}.
- C2: an UE can be both in UE_{detect_F} and UE_{infer_T}.
- C3: an UE can be both in UE_{infer_T} and UE_{infer_F}.

For C1 and C2, the inference process indicate a different UE state than the detection process. For C3, two inference processes indicate a different UE state. Thus, the FTM must choose to which set the conflicting UE should be assigned. The ideal solution would be to assign the conflicting event to the most likely set on a case-by-case basis. However, this solution is tedious and not necessarily simple to implement, simply because we do not always know the most likely set. We have instead opted for an automatic resolution of these conflicts by introducing 2 hypotheses: *priority_on_occurrence* and *priority_on_detection*.

The *priority_on_occurrence* hypothesis favors the occurrence of an UE over its non-occurrence. It is therefore a conservative approach considering the worst scenario for the system to guarantee the maximum reliability. For the 3 forms of conflicts listed above, if *priority_on_occurrence* is true, then UE_{detect_T} should be chosen for C1, UE_{infer_T} should be chosen for C2 and UE_{infer_T} should be chosen for C3. We consider this hypothesis by default in our framework.

The *priority_on_detection* hypothesis overrides the *priority_on_occurrence* hypothesis for C1 and C2. It favors direct detection by detection algorithms over inference by fault tree logic, thus considering that the logic provided by the tree is less reliable than direct detection. In the cases listed above, if *priority_on_detection* is true, then UE_{detect_T} should be chosen for C1 and UE_{detect_F} should be chosen for C2. We do not consider this hypothesis by default in our framework.

It should be noted that in the current configuration of our implementation (*priority_on_occurrence* true and *priority_on_detection* false), the detection or inference affirming the non-occurrence of an UE are only informative. Indeed, they cannot prevent the detection or inference affirming the occurrence of an UE by other rules.

In the next section, we present the complete inference algorithm that uses the presented basic inference rules.

3.4.2.2 Inference algorithm

The complete algorithm relating the inference of UEs is given in the algorithm 1. From an event *ev*, the function $local_infer : ev \rightarrow \{E_T, E_F\}$ infers the immediate connected events of *ev* according to the inference rules IR1, IR3, IR4, IR5, IR7 and IR8. E_T is the set of immediate connected events inferred as true whereas E_F is the set of immediate connected events inferred as false. The sets of local inferred events are returned by the function which follows additional rules to avoid infinite loops and conflicts:

- if an event *ek* is in (UE_{detect_T} or UE_{infer_T}) before the execution of $local_infer$, and if *ek* should normally be inferred in E_T when $local_infer$ returns, then $local_infer$ does not infer it.
- if an event *ek* is in (UE_{detect_F} or UE_{infer_F}) before the execution of $local_infer$, and if *ek* should normally be inferred in E_F when $local_infer$ returns, then $local_infer$ does not infer it.
- $local_infer$ may take into account *priority_on_occurrence* or *priority_on_detection* hypothesis according to developer wishes (in our implementation, we only make the hypothesis of *priority_on_occurrence*).

Algorithm 1: Inference of UEs by the FTM

$UE_{infer_T} \leftarrow \emptyset$; $UE_{infer_F} \leftarrow \emptyset$;
let E and E' be two sets in $UE_{express}$;
$E' \leftarrow UE_{detect}$;
while $E' \neq \emptyset$ **do**
  $E \leftarrow E'$; $E' \leftarrow \emptyset$;
  foreach $ev \in E$ **do**
    $\{E_T, E_F\} \leftarrow local_infer(ev)$;
    $UE_{infer_T} \leftarrow UE_{infer_T} \cup E_T$;
    $UE_{infer_F} \leftarrow UE_{infer_F} \cup E_F$;
    $E' \leftarrow E' \cup \{E_T, E_F\}$;
  end
end
return UE_{infer_T} and UE_{infer_F} ;

At the end of the inference step, the FTM has therefore determined the complementary sets in $UE_{express}$: UE_{detect_T}, UE_{detect_F}, UE_{infer_T}, UE_{infer_F} and $UE_{unknown}$ ($UE_{unknown} = \overline{UE_{detect}} \cup \overline{UE_{infer}}$). We define UE_T as:

$$UE_T = UE_{detect_T} \cup UE_{infer_T} \tag{3.7}$$

After the detection and inference of UEs with our framework, we now turn to the implementation and execution of recoveries.

3.5 Implementation and execution of recoveries

In this section, we present the implementation of the local and decisional recoveries and their execution during run-time.

3.5.1 Implementation and execution of local recoveries

We begin by presenting the Local Recoveries (LRs), which aim to eliminate or prevent the consequences of a particular UE by using local/embedded resources. To increase the reliability of the system, we will first address how the expert decides on the implementation of local recoveries to react to UEs.

3.5.1.1 Implementation of local recoveries

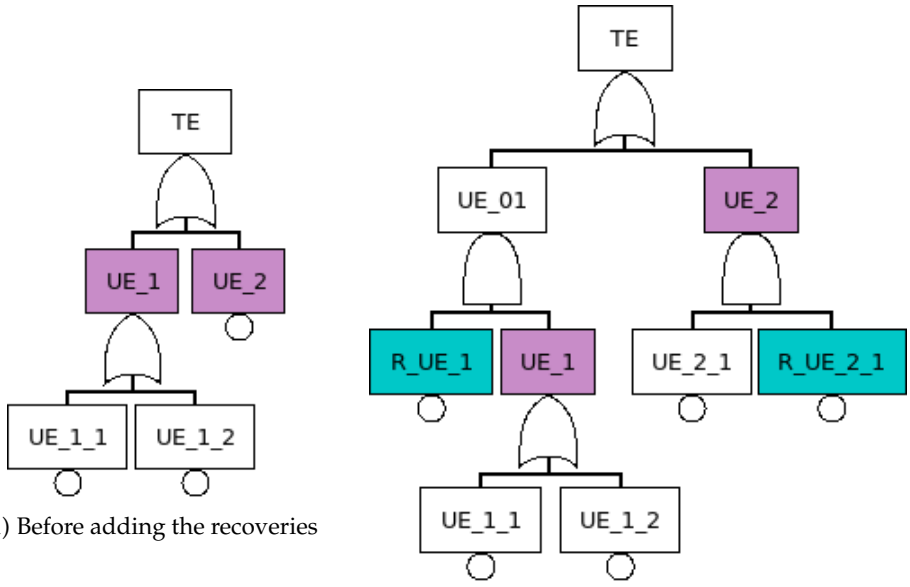
LRs are defined in the Fault Tolerance Manager (FTM), which implements the core of our framework. A LR is associated with one or more UEs: it is triggered when this or these UEs are detected or inferred as true. We also define that the LR reacts to this or these UEs. An UE can either not trigger or trigger several LRs, classified in 3 categories according to their actions:

- override configuration parameters: these LRs aim to modify configuration parameters of the system (e.g. control gains, thresholds).
- manage modules execution: these LRs aim to manage the execution of modules (e.g. restart, change execution period)

- override control parameters: these LRs aim to slightly degrade the current task (e.g. deviate trajectories, change desired speeds, override actuator commands)

Implementation of the local recoveries in the fault trees We represent the integration of a LR in the FT by the representing an additionnal event corresponding to the failure of this LR, next to the UE to which it is associated, and under an AND gate. As an illustration, Fig 3.5 represents a fault tree before and after including the LRs, whose failures are noted R_UE_1 and R_UE_2_1. The LRs react respectively to the two events UE_1 et UE_2_1.

There are two ways to include LR failures, depending if LRs react to UEs already defined in the FTs or if they react to new UEs inserted by a logic necessity. For example, on the left side of the tree, the LR whose failure is R_UE_1 reacts to the event colored in pink UE_1 and prevents the newly defined event UE_01 to occur. On the right side of the tree, the LR whose failure is R_UE_2_1 reacts to a newly defined event UE_2_1 to prevent UE_2.



(A) Before adding the recoveries

(B) After adding the recoveries

FIGURE 3.5: A fault tree before and after adding the failure of the local recoveries

As for any UEs, a LR failure may be detected by a dedicated algorithm. For example, it can be considered that a LR is failing if it launched a predetermined number of times while the UE supposedly covered is still present. A failure of one or more UEs can also be inferred through diagnosis from the UE associated with the AND gate. For example, in Fig 3.5, if UE_2 is detected as true, then this implies the occurrence of UE_2_1 and the failure of its associated LR (R_UE_2_1).

In the next section, we discuss the establishment of the LR's characteristics, necessary for the implementation and proper functioning of the framework.

Determination of LR's characteristics In order to ensure the proper functioning of the local recovery process, several parameters must be defined and established regarding the LRs.

	LR_1	LR_2	LR_3	LR_4
LR_1	0	1	1	1
LR_2	1	0	0	1
LR_3	1	0	0	1
LR_4	1	1	1	0

(A) The matrix of compatibility between LRs

List of incompatible LR with LR_1	$\{LR_1\}$
List of incompatible LR with LR_2	$\{LR_2, LR_3\}$
List of incompatible LR with LR_3	$\{LR_2, LR_3\}$
List of incompatible LR with LR_4	$\{LR_4\}$

(B) Lists of incompatible LRs

FIGURE 3.6: Lists of incompatible LRs from a matrix of compatibility

We first define the **recovery step** as the current state of a LR. A LR can be in 3 different states: *idle*, *in progress* or *finished*. In *idle* step, the LR is not active and is ready to be initiated if necessary. In *in progress* step, the LR is executing the action for which it has been implemented. During the *finished* state, the LR is no longer active and waits for the framework to put it back in *idle* state.

In complex systems, many LRs can be considered. With realistic assumptions, several faults or errors are possible at the same time, and therefore one can potentially be led to execute several LRs at the same time. For this purpose, we define specific LR parameters that are established during the LR definition (see class diagram Fig 3.8):

- **interruptibility**: this parameter indicates whether the LR can be interrupted during execution (while the LR is *in progress*). We consider that LRs cannot be interrupted if their interruption would cause an unwanted change in the system configuration. For example, we consider that a LR that restarts a module cannot be interrupted, because the LR must first shut down the module (if it is still operational) before restarting it. An interruption of the LR in the middle of these phases could then cause a simple shutdown of the module.

- **incompatible recoveries**: it is a list of LRs incompatible with the execution of the current LR. This list is elaborated from a symmetrical compatibility matrix between the local recoveries (Fig 3.6). This matrix contains 1 in a specific cell if the LR of the row and the LR of the column are compatible, and 0 otherwise. LRs are considered incompatible if they write on the same resource and more generally if their actions interfere with each other. For example, for example, two LRs that both shift the trajectory of the robot in the same or in the opposite direction are obviously incompatible.

- **incompatible tasks**: it is a list of robotic tasks that are incompatible with the execution of the LR. This list is also built from a compatibility matrix between LRs and possible robotic tasks. Similarly, the matrix contains a 1 if the LR of the row and the robotic task of the column are compatible and 0 otherwise. A LR is considered incompatible with a robotic task if the LR directly prevents the robot to fulfill the objective of the task. For example, a LR that changes the trajectory of the robot is not compatible with a current robotic task that specifies the robot to stay at the same position.

Furthermore, we have stated that LRs are associated to UEs, since a LR must prevent the consequences of an UE and/or eliminate it. A LR may have a different priority and nuance of action depending on the UE to which it is associated. Other parameters, established during the association between LRs and UEs, must therefore be defined to represent this association:

- **triggered event**: the ID (integer) of the associated UE that triggers the implemented LR.
- **priority**: a real number superior than or equal to 1. The highest priority is represented by the lowest value. This parameter reflects the importance that the LR has in the context for which it was implemented and is to be seen in relation to the priorities of other LRs. This priority is determined by an expert and depends mostly on the criticality of the associated UE and the efficiency of the LR. Thus, the higher the criticality of the UE associated with the LR, the higher the priority of the LR. Similarly, the more effective the LR, the higher the priority of the LR.
- **input parameter**: the value of a potential input parameter of a LR. This allows to nuance or differentiate the action that the LR initiates during its implementation. Let's take the example of a LR whose action is to override the desired control force in a direction to avoid an obstacle. The input parameter can set the force value, and thus nuance the action of the LR to deal with the UE in question.

We will address in the next section the principle of execution of LRs and the management of their incompatibilities.

3.5.1.2 Execution of the local recoveries

We have stated that the LRs are not necessarily compatible with each other. Thus the FTM must not execute two incompatible LRs at the same time. The FTM must also deal with the LRs that are currently being executed. A decision must then be made about which local recoveries to initiate. The problematic in this section is then how to choose and perform LRs. We present here our LR selection algorithm.

Step 1: stopping incompatible and unnecessary LRs From the detection and inference, we have obtained the set of estimated true UEs (UE_T). We suppose that the running LRs are listed in a set Λ (LRs in *in progress* state). From there, the framework plans first to interrupt and remove from Λ:

- the interruptible LRs whose event is no longer estimated as true.
- the interruptible LRs that are not compatible with the current task. That may be the case if the task has changed between 2 iterations of the FTM execution.

The interrupted LRs are then put back in the *idle* state.

Step 2: building groups of compatible LRs In a set Λ', we then group the different running LRs of Λ with the other LRs that should be executed because their associated UEs are in UE_T. From the LRs in Λ', the goal of this phase is to form groups of compatible LRs. A LR can be in one or more groups. We impose that a group

should not be represented as a combination of other groups to maximize the size of the groups and thus the number of executable LRs in parallel. We have developed the following algorithm to constitute the groups of compatible LRs:

Algorithm 2: Constitution the groups of compatible LRs

```
// Purpose: obtain G_1 to G_m groups of compatible LRs
let LR_1 to LR_n be the n ∈ ℕ LRs of Λ';
if n > 0 then
    i ← 1 ; m ← 1;
    G_1 ←{LR_1} ;
    while i < n do
        // Put LR_i in the m groups
        i ← i + 1 ; k ← 1 ; l ← 0;
        while k ≤ m do
            if LR_i compatible with all LRs in G_k then
                G_k ←G_k ∪{LR_i} ;
            else
                // Add a new group
                l ← l + 1;
                G_{m+l} ← {LR_i} ∪ G_k \{LRs incompatible with LR_i} ;
            end
            k ← k + 1;
        end
        // Remove groups included in other groups
        while l > 0 do
            if ∃j ∈ ℕ; j ∈ [1, m + l]\{m + 1}; G_{m+1} ⊂ G_j then
                remove G_{m+1};
            else
                m ← m + 1;
            end
            l ← l − 1;
        end
    end
end
```

Following the algorithm, we determine a set of G_1 to G_m groups of compatible LRs. As we can initiate only one of these groups, we will address which group to choose.

Step 3: choosing the appropriate group of compatible LRs We recall that the objective of LRs is to avoid or contain UEs while pursuing the mission. We also explained that there is a weighting between the importance given to LRs responding to UEs, established by the individual priority of LRs. The purpose of this step (step 3) is to establish a group priority criterion with the ultimate goal of executing only the LRs contained in the group with the highest priority. We use 3 criteria to select the priority group:

- the number of LRs in each group. We want the more LRs in a group, the higher the priority of that group, as it will theoretically allow us to react to more UEs.

- the individual priorities of the LRs within each group. We need the higher the individual priorities of the LRs in a group, the higher the priority of that group. This will theoretically allow the system to react to high criticality UEs first.
- the number of non-interruptible LRs that are in-progress in each group. We have identified that it could be problematic to interrupt LRs that are considered as non-interruptible. If such LRs are in progress, it will be necessary to choose a group that contains them.

First, if non-interruptible LRs are in progress, and thus necessarily represented in one or more groups, the FTM eliminates the other groups of LRs. These groups cannot be initiated in the current circumstances. We implement a function $calculate_gp()$ which takes as input a group of compatible LRs and returns the group priority. The function maximizes the priority with the number of LRs and with high individual LR priorities. We consider the highest group priority to be represented by the lowest value.

In order to balance the importance of the criteria of number of LRs and individual priorities, we define 2 positive parameters α and β, set according to the preferences of the user and the expert. α balances the criterion of the number of LRs between groups and β weights the individual priorities of the LRs in a group. For a group of n LRs, with individual priorities p_i, the group priority gp is expressed as follows:

$$gp = (n^{\alpha-1} \sum_{i=1}^{n} ({p_i}^{-\beta}))^{-1} \tag{3.8}$$

Based on the calculation of the priorities of the different groups of compatible LRs, the FTM keeps only the group with the highest priority (lowest value). If two groups have the exact same highest priority, then the FTM chooses randomly among these 2 groups.

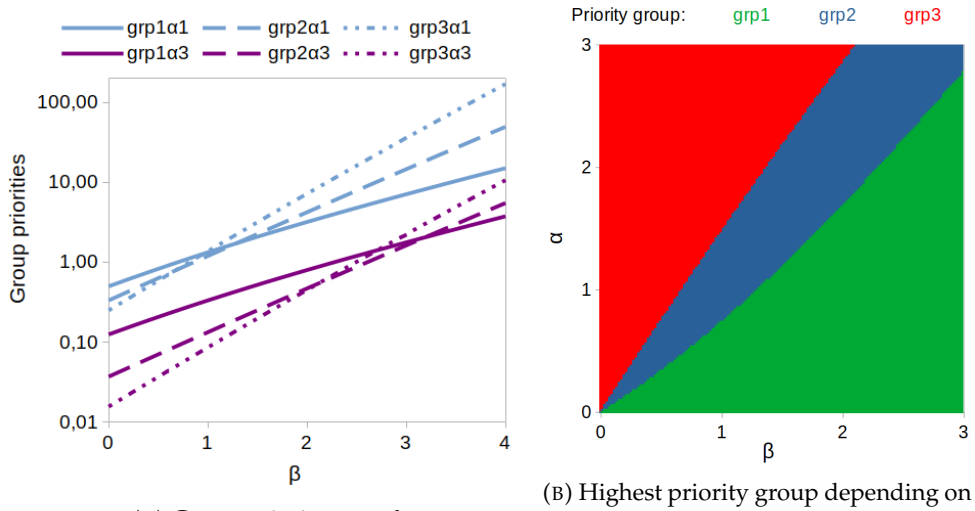
Let's take an example with 3 groups:

- group1 has 2 LRs with individual priorities of 2 and 4.
- group2 has 3 LRs with individual priorities of 3, 4 and 4.
- group3 has 4 LRs with individual priorities of 4, 6, 6 and 7.

We plot Fig. 3.7a the calculated priorities of these three groups over β, with $\alpha = 1$ and $\alpha = 3$. Fig. 3.7b indicates which group has the highest priority (i.e. the lowest value of calculated priority) according to α and β.

If α is high, $calculate_gp()$ tends to maximize the importance of the number of LRs in the group for the group priority calculation. For example, when $\beta = 1$, group3 has more priority over group2 and group1 when α is higher because group3 has more LRs.

If β is high, $calculate_gp()$ tends to maximize the importance of the individual priorities of LRs for the group priority calculation. For example, when $\alpha = 1$, group1 has more priority over group2 and group3 when β is higher, because the highest priority of group1 is 2 while the highest priorities of group2 and group3 are respectively 3 and 4.



(A) Group priority over β

(B) Highest priority group depending on α and β

FIGURE 3.7: An example of group priority for 3 groups of compatible LRs

Step 4: launching the LRs The last step is to execute the LRs in the selected LR group. Interruptible LRs that are in progress but are not part of the selected group are then interrupted (put in *idle* state). The LRs of the selected group that were in progress before the group selection process remain in progress if not finished. The remaining LRs of the selected group are initiated.

The termination of a LR can take place in two different scenario. In the first scenario, the LR continues to execute its action until the FTM interrupts it and sets it back in the *idle* state. It can be the case to execute other LRs that are not compatible with it or if its associated UE is no longer considered true. In the second scenario, the LR runs and switches to the *finished* state at the end of its action.

We will address in the next section how the decisional recoveries are implemented and executed.

3.5.2 Implementation and execution of decisional recoveries

In critical cases, where the safety of the robot is compromised or when the mission is no longer feasible, at least in its original form, LRs are not sufficient to mitigate severe UEs and more generally to guarantee the reliability of the mobile robot. In the case of severe events and to avoid an even more catastrophic consequences, more significant reactions must be considered.

It may then be necessary to change the current robotic task to avoid wasting unnecessary time when the mission is failing and to preserve the safety of the robot. From the perspective of the fault tolerance manager, a Decisional recovery (DR) results in a change in the robotic task that must be considered for subsequent of robotic activities. Similarly to LRs, DRs are actions that are intended to react to UEs. The DRs can vary depending on the detected UE, e.g. stopping the robot, moving the robot in a safe position, launching a diagnostic task.

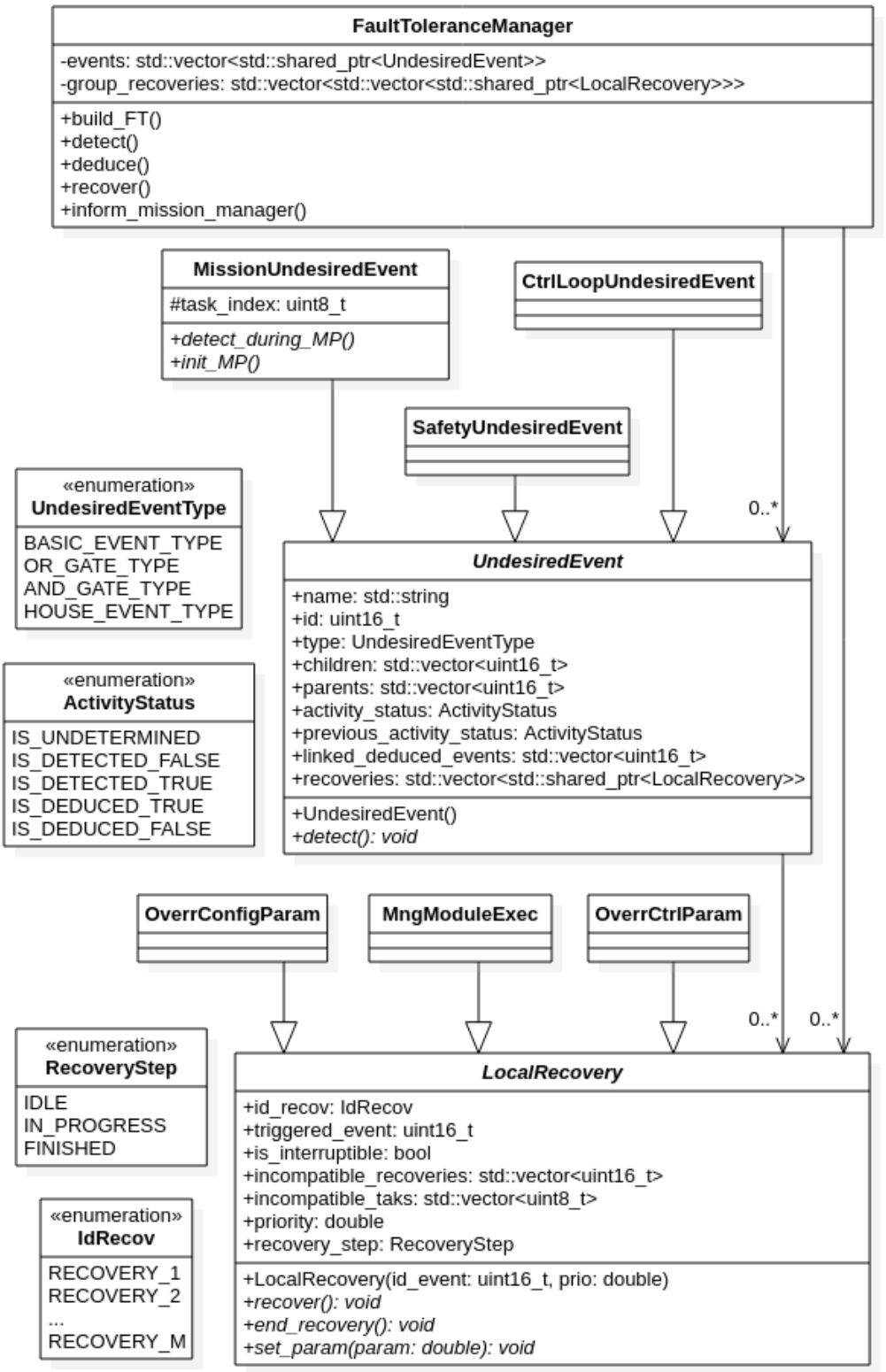


FIGURE 3.8: The framework UML class diagram (implemented in C++)

DRs are chosen at a decisional level by a mission manager. In our work, we represent the mission manager as a black-box module. It considers the characteristics of the mission and the occurrence of a predefined set of UEs (noted E_{DR}) as inputs. It generates the characteristics of the new robotic task to be performed as an output.

In order to launch the DR, the FTM informs the mission manager of the occurrence of UEs in E_{DR} that lead to the deduction of a top event. Such a scenario means that there is no redundancy available or that the redundancies (including LRs) were not sufficient to prevent the deduction of the top event.

3.6 Conclusion

In this chapter, we present our new fault tolerant framework for mobile robots which integrates several tools and methods studied in the chapter 2. This framework has the ability to react to undesired events on several levels of abstraction. First, an expert identifies potential Undesired Events (UEs) that may occur in the robotic system. This identification results in the creation of two Fault Trees (FTs), organized around the safety and the mission for the violation of high level constraints. A third fault tree is built around the undesired events occurring in the the control loop. We provide a guideline to construct the fault trees.

After the construction of the fault trees, the expert first identifies the weaknesses of the system. He then has the possibility to implement the detection of a set of UEs, in addition to Local Recoveries (LRs) to enhance the dependability of the system. During the running phase, the Fault Tolerance Manager first evaluate the set detectable UEs. An inference is then carried out from the FTs previously obtained to diagnose the immediate causes and deduce the immediate consequences of the UEs detected as true. LRs may be initiated to react to predetermined UEs using local resources. A study of compatibility between LRs is then conducted with the goal of initiating the compatible and priority LRs. Decisional Recoveries (DRs) may also be launched when local recovery solutions are no longer available or have failed.

In the next chapter, we will discuss the test case of the proposed fault tolerant framework: an underwater robot carrying out a transect mission.

Chapter 4

Case study: Experimental platform and context

Contents

4.1 A introduction to recent marine ecosystem monitoring robots . . 61
4.1.1 A classification of underwater robots 62
4.1.2 Notations for underwater robotics 62
4.1.3 Underwater robots for marine ecosystem assessment 64
4.2 The experimental platform of our case study 65
4.2.1 Hardware architecture of the experimental platform 65
4.2.2 Software architecture of the experimental platform 68
4.3 Mission description: the underwater transect 74
4.3.1 An approach to observe the marine environment 74
4.3.2 Overview of the robotic transect mission 75
4.3.3 Definition of the mission constraints and mission properties 77
4.4 Fault tree, detection and recovery implementation 82
4.4.1 Implementation of the safety fault tree 83
4.4.2 Implementation of the mission FT 87
4.4.3 Implementation of the control loop fault tree 89
4.5 Conclusion 94

In this chapter, we will present a case study of an underwater robot that observes marine biodiversity. This case study gathers a context that will allow us to test the fault tolerant framework proposed in chapter 3: autonomy, complex marine environment, target mission given by non robotic users.

First, we will introduce how underwater robots monitor the marine ecosystem. We will then describe the Unmanned Underwater Vehicle (UUVs) that support our framework. We will then discuss the mission specifications that we have chosen to illustrate our approach. Finally, we will detail the implementation of the proposed fault tolerant framework on the robot.

4.1 A introduction to recent marine ecosystem monitoring robots

In this section, we will first provide a brief introduction to underwater robots before focusing more broadly on underwater robots that perform missions involving environmental monitoring.

4.1.1 A classification of underwater robots

Unmanned Underwater Vehicles are being used more and more as technology advances. We can classified them into 3 main categories according to their control: **ROV** (Remotely Operated underwater Vehicle), **AUV** (Autonomous Underwater Vehicle) and **glider**.

ROVs are underwater vehicles piloted by an operator on the surface (Fig 4.1a). The connection between the operator and the ROV is often provided by a cable linking the robot and the surface equipment. They are rather used to monitor human infrastructures and can benefit from the intelligence of the human to solve decision making. Most of them are equipped with several thrusters to maximize the Degrees of Freedom (DoF) of motion. ROVs are increasingly used with autonomous functionalities([117], [135], [48]). The term Hybrid ROV (**HROV**) is used to describe ROVs that can be remotely operated by an operator and carry out missions in quasi or total autonomy ([18]).

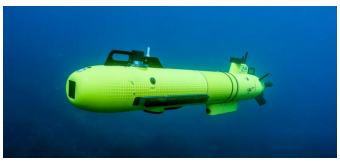
AUVs are fully autonomous and have no cable connecting the robot to the surface. They are rather used in the surveillance of large areas and often take the form of a torpedo (Fig 4.1b), with a thruster at the rear and fins for steering. The elongated torpedo shape allows to minimize the water friction and thus to realize longer missions at higher speeds. The missions carried out by AUVs generally take place far from the seabed or from any obstacle. However, with technological advances, with the ability to better perceive the environment, and more generally with increased autonomy, more AUVs are designed with more DoF and operate with more interaction with the environment ([37],[124]).

Also autonomous, gliders usually do not possess thrusters and let themselves be carried by the sea currents. In the shape of torpedoes (Fig 4.1c), they can be equipped with fins and ballasts to stabilize and control the depth. They are used for long missions, sometimes over distances of several thousand kilometers ([141],[81]), mainly for environmental monitoring.



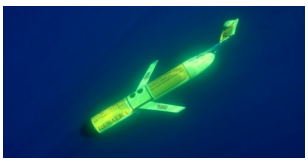
(A) BlueROV ROV

Source: syera.fr



(B) ECA A9 AUV

Source: meretmarine.com



(C) Slocum glider

Source: [96]

FIGURE 4.1: The 3 main classes of underwater robots

4.1.2 Notations for underwater robotics

In this section, we define terms specific to mobile and underwater robotics from [47]. The defined terms will be used later in this chapter, particularly to describe mission constraints.

4.1.2.1 The reference frames

The Earth-Centered Earth-Fixed (ECEF) (x_e, y_e, z_e) is a fixed reference frame to the Earth. Its origin is located at the center of the planet. x_e points towards the intersection of the equator and the Greenwich meridian. z_e points towards the north pole.

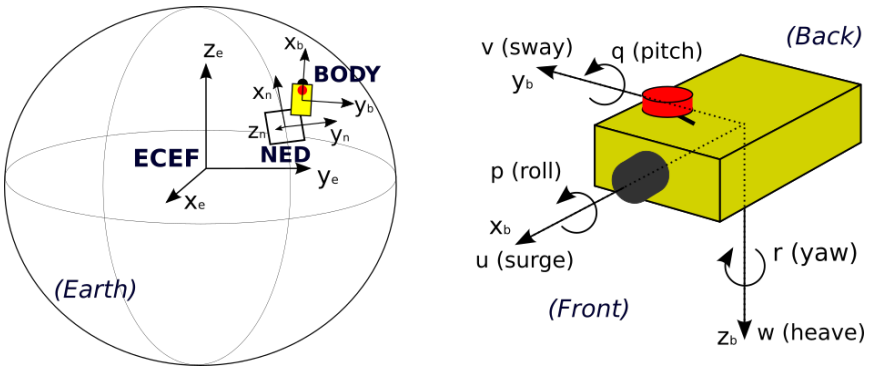


FIGURE 4.2: Reference frames and motion notations

The North-East-Down (**NED**) frame $\{n\} = (x_n, y_n, z_n)$ is defined with respect to the reference ellipsoid of the Earth (WGS84). Located on the ellipsoid, its origin is specified by a latitude and a longitude. The plan (x_n, y_n) is tangent to the ellipsoid. x_n points towards the north and y_n points towards the east. For an underwater robot near the NED frame, we can assume that the NED frame is inertial and that Newton's laws apply.

The **BODY** frame $\{b\} = (x_b, y_b, z_b)$ is a reference frame attached to the robot. Its origin is often located at the center of the robot, e.g. between the center of gravity and the center of buoyancy of the robot. x_b points towards the front of the robot while y_n points towards the right of the robot.

4.1.2.2 The motion notations

We use the Society of Naval Architects and Marine Engineers (SNAME) notations to qualify the positions, velocities and forces of underwater robots. The forces and velocities are listed in Tab 4.1 according to the DoF involved.

DoF	Forces and moments	Linear/angular velocities
Surge (in the x_b direction)	X	u
Sway (in the y_b direction)	Y	v
Heave (in the z_b direction)	Z	w
Roll (about the x_b axis)	K	p
Pitch (about the y_b axis)	M	q
Yaw (about the z_b axis)	N	r

TABLE 4.1: SNAME notations for marine vehicles

Furthermore, we call:

- $\nu = [u, v, w, p, q, r]^T$ the robot velocities in $\{b\}$
- $\tau = [X, Y, Z, K, M, N]^T$ the robot forces and moments in $\{b\}$
- $Pos = [x, y, z]^T$ the position of the robot in $\{n\}$
- $\Theta_e = [\phi, \theta, \psi]^T$ the attitude of the robot in $\{n\}$ (Euler formalism)
- $\Theta_q = [q_w, q_x, q_y, q_z]^T$ the attitude of the robot in $\{n\}$ (quaternion formalism)
- $\eta_e = [x, y, z, \phi, \theta, \psi]^T$ the pose of the robot in $\{n\}$ (Euler formalism)
- $\eta_q = [x, y, z, q_w, q_x, q_y, q_z]^T$ the pose of the robot in $\{n\}$ (quaternion formalism)
- WC (water column) the distance from the surface to the seabed (given x and y)
- $alti$ (altitude of the robot) the vertical distance between the robot and the seabed

Underwater robots can have several uses, e.g. inspection of structures, archaeology, cartography, rescue mission, defense, environmental survey, diving companion, underwater operation (welding, handling), scientific research. We will address in the next section how UUVs are employed to assess the marine ecosystem.

4.1.3 Underwater robots for marine ecosystem assessment

The ecosystem study is an aspect of the environmental survey and consists in identifying marine species and their behavior in their natural habitat. [163] divides the means of measuring the environment and marine species by underwater robots into 3 categories: acoustic (using passive or active monitoring), electro-optical imaging (using RGB or IR cameras) and animal-borne tags (using acoustic tags).

In general, acoustic-based approaches are used for applications where the robot is at a large or medium distance. This can help identify the presence of schools of fish, or estimate the distance and type of seabed (sand, coral, algae).

Visual-based approaches are used for applications where the robot is close to the target, because visibility drops rapidly with the absorption of light by water. These approaches are still needed to accurately identify marine species. The automatic identification of species by underwater robot is often related to works using convolutional neural networks (CNN), which allow to automate the identification of marine species ([100]). Recent underwater robots, mainly ROVs, have therefore been developed to film species up close ([156]). For example, [109] and [113] develop AUVs to identify coral reefs whereas [171] uses an AUV to study benthic habitat. [37] develop an AUV that visually identifies starfish before injecting them with a product to neutralize this invasive species.

The LIRMM ([96]) develops a HROV performing biological monitoring protocols close to coral reefs (Fig 4.3). The studied protocols of observation of the ecosystems are the transect, which we will detail later in this manuscript (section 4.3), the localized observation around a point of interest, and the tracking of species. To guarantee the mission and safety, the robot must be maneuverable (at least iso-actuated), and have a small size to be less carried away by the sea current. The results relative to

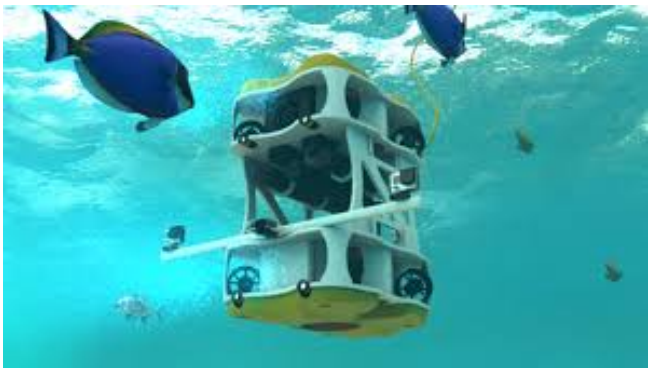


FIGURE 4.3: Ulysse underwater robot
Source: [23]

the ecosystems provided by the robot on the one hand, and provided in a more traditional way by a diver carrying out the same protocol on the other hand, seem to be similar ([103]).

Based on these previous works and a review of basic notations regarding underwater robotics, we will focus on our case study: an underwater robot developed at LIRMM.

4.2 The experimental platform of our case study

In this section, we detail the hardware and software architectures of our case study: an underwater robot developed to carry out observation missions of the marine ecosystem.

4.2.1 Hardware architecture of the experimental platform

Since the work of [96], we have decided to standardize the electronic and computer architectures of all our underwater robots. From the experience of these years, we built our own underwater robot called REMI to meet our needs. REMI has been developed in partnership between the LIRMM, and marine biologists from Marbec and CUFR of Mayotte.

REMI was designed in 2 phases. During the first phase (2018-2019), the robot was equipped with a Dropix embedded controller which monitored actuators and received data from navigation sensors (IMU, pressure sensor and echosounder). A program running on a surface computer communicated with the Dropix to close the control loop. After performing operational tests in Montpellier and Mayotte ([65]), we concluded that:

- the size and power of the robot were adequate to achieve our goals.
- there was a lack of redundancy and sensors to provide accurate localization and obstacle avoidance.
- closing the control loop at the surface was not desirable, because it affected the performance of the robot and its autonomy.

These tests led to the second phase of development (2019-2021) that we detail in the rest of this section. REMI is a HROV of less than a meter long and weighing approximately 30 kg (Fig. 4.4).

With 4 vertical and 4 horizontal thrusters (M1 to M8), REMI has 6 degrees of freedom. The robot has 2 sealed enclosures: the front enclosure contains the new UP-board controller and components related to the operation of the robot and the rear enclosure contains components related to the control of motors (electronic speed controllers and powerswitches).

The robot uses 7 sensors for navigation (Tab 4.2): an echosounder for altitude measurement, an IMU for acceleration, rotation velocity and attitude measurement, a pressure sensor for depth measurement, a vertical camera for flow optic measurement, a GPS for latitude and longitude measurement, an USBL beacon with an integrated pressure sensor, coupled to another USBL beacon on the surface associated with a GPS to measure the position of the robot at depth, and a DVL for speed and altitude measurement.

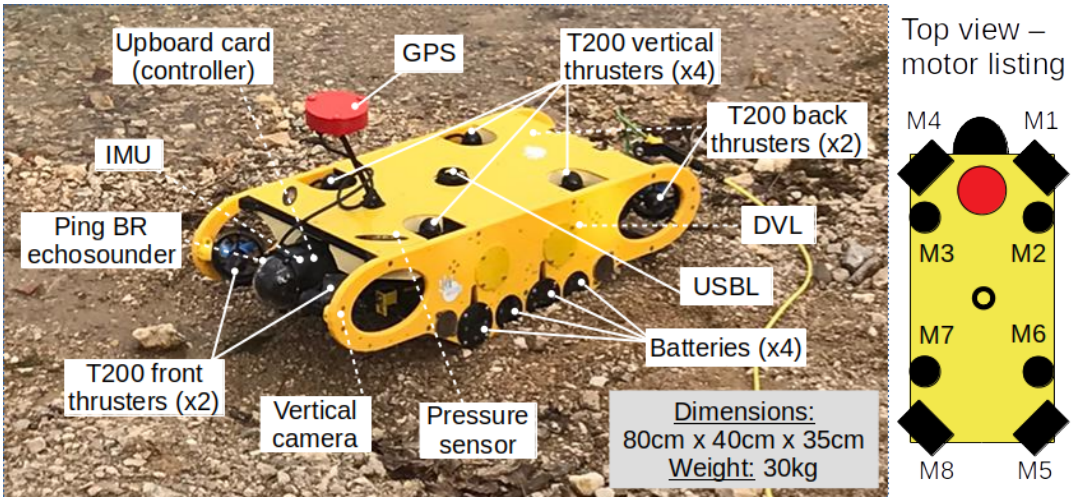


FIGURE 4.4: REMI robot

Sensor	Variables measured by sensors							Conditions
	$x\ y$	z	$alti$	$\phi\ \theta\ \psi$	$u\ v$	w	$p\ q\ r$	
GPS	X				X*			$z < 0.15m$
USBL	X	X						$z > 0.1m$
DVL			X		X	X		$0.1m < alti < 50m$
Vertical camera					X			$0.5m < alti < 5m$
IMU				X	$\int$X	$\int$X	X	
Pressure sensor		X				X*		
Echosounder			X					$0.5m < alti < 30m$

TABLE 4.2: REMI sensors and their measured variables

X $\rightarrow$ the measurement of the sensor is used normally. X* $\rightarrow$ the measurement of the associated sensor is derived. In that case, roll and pitch angles should be small. $\int$X $\rightarrow$ the measurement of the associated sensor is integrated.

Other non-navigation sensors are also present. Thus the pressure sensor and the USBL beacon are equipped with thermometers for the measurement of the sea temperature. The IMU is equipped with a thermometer for the measurement of the internal temperature (front enclosure). The powerswitches (PWSs) and the power supply devices are equipped with voltmeters and ammeters. A front camera provides real-time visual feedback to the operator on the surface. Sponge sensors function as water leak detectors and can detect the presence of water in the front or rear enclosure. Two pairs of stereo cameras, facing forward and downward, are added to the robot in order to record visual data of interest for biologists.

The 4 batteries allow up to 4 hours of energy autonomy depending on the conditions of use. The communication between the surface and the robot is ensured by a 200 meters cable of neutral buoyancy to avoid its hooking on the seabed. The details of the hardware is given Tab. 4.3. We also inform the costs of the hardware components to underline that they remain low, compared to other components found on ROVs or AUVs intended for the observation of underwater ecosystems. The average cost of our robot's components is 15000€. In comparison with the market and to our knowledge, an USBL is worth between 7000€ and 30000€, a DVL is worth between 5000€ and 25000€ and an IMU is worth between 10€ to more than 50000€ for the most efficient ones.

Type	**Denomination**	**Reference**	**Price (€)**	**Number**
Actuator	Actuator/Thruster	T200 BlueRobotics	200	8
Sensor	GPS/Embbedded GPS	DP0104 drotek	50	1
Sensor	USBL transducer	X150 Seatrac	7000	1
Sensor	USBL transponder	X010 Seatrac		1
Sensor	Surface GPS	BU-353-S4 GlobalSat	40	1
Sensor	DVL	Water Linked A50	5000	1
Sensor	Vertical camera	IMX IP camera	70	1
Sensor	Front camera	IMX IP camera	70	1
Sensor	Stereo camera	Go Pro HERO3	80	4
Sensor	IMU	BNO055	30	1
Sensor	Pressure sensor	MS5837	50	1
Sensor	Echosounder	Ping BlueRobotics	200	1
Controller	Controller	UP board	180	1
Battery	Battery	(NiMH)	60	4

TABLE 4.3: Hardware features of the REMI robot

After the review of the hardware architecture of the robot, we propose to focus on the software architecture, on which the implementation of the fault tolerance framework will be based.

4.2.2 Software architecture of the experimental platform

The software architecture of the control loop integrated in the UPboard controller is illustrated Fig 4.5. It is composed of several independent entities (drivers, managers, functionalities), executed sequentially in modules launched by a module manager and implemented in C++. We use the PID framework ([130]) to implement our drivers and our architecture, in order to standardize the software components as much as possible for reuse within the laboratory.

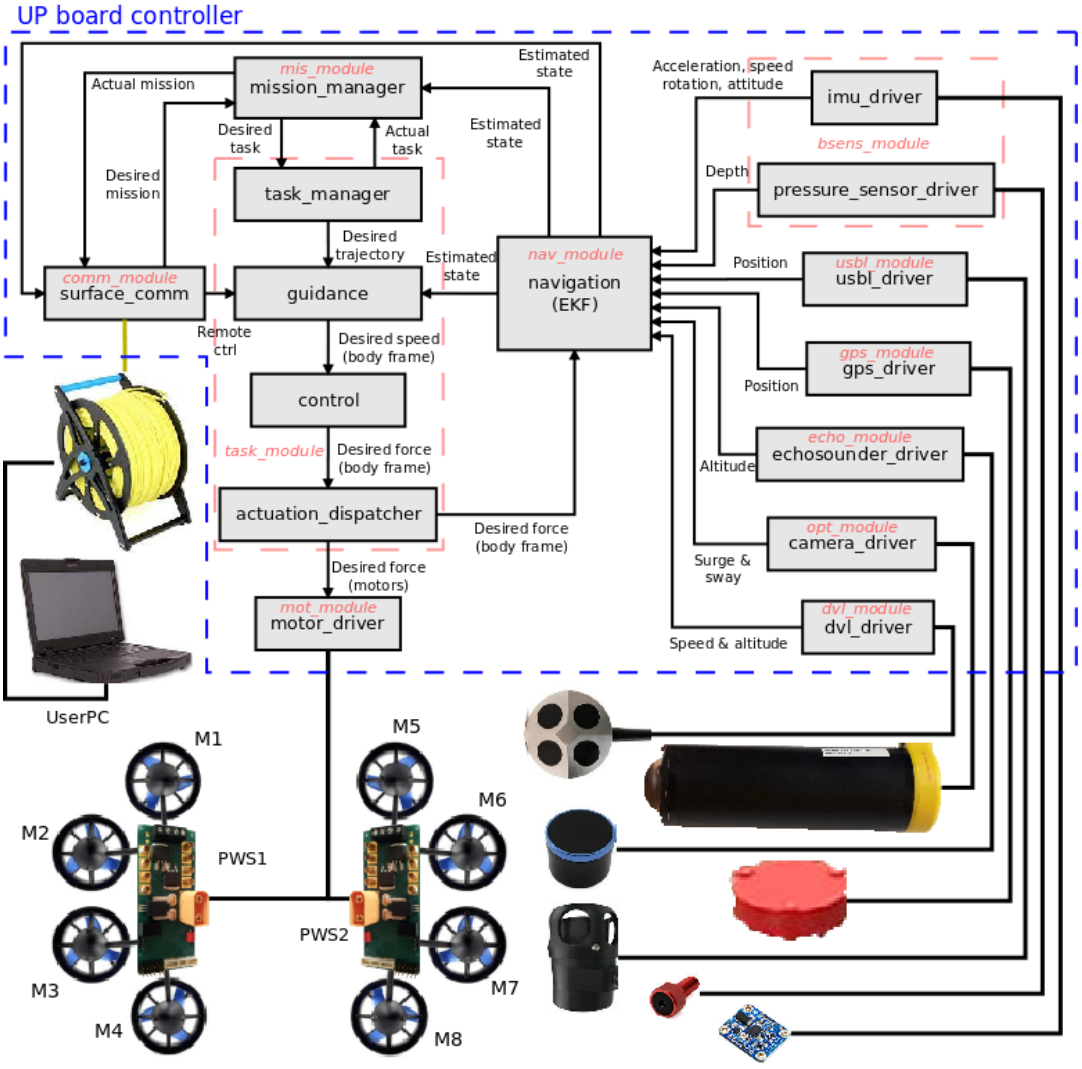


FIGURE 4.5: Software architecture of the control loop of REMI

In the following sections, we will detail the modules used in the control loop, starting with the sensor modules.

4.2.2.1 Description of the sensor modules

Six independent sensor modules including drivers work at different frequencies and receive data from the sensors. The general operating principle of the sensor modules is the same. They first connect to their associated sensors via their respective communication protocol. They interrogate their associated sensor in a loop to obtain measurements, which are stored in a sliding table with their timestamps and

their confidence values. The confidence values are represented by the standard deviations of the estimated measurement noises. These standard deviations can be directly provided by the sensors, or are estimated by the sensor modules.

The modules can then apply different filters to either enhance or reject the sensor measurement. For example, a module may refuse to provide a measurement whose measurement noise exceeds a threshold value (FilterA). A module can also apply a sliding average filter on a number of values from the current measured value and a history of measured values, to reduce the effects of noise on the sensor measurement (FilterB).

The *bsens_module* The *bsens_module* runs the water leak detectors, the *imu_driver* and the *pressure_sensor_driver* consecutively. It receives information from these sensors at a frequency of 15Hz. We have grouped these drivers on the same module because they constitute the basic sensors of the common architecture for all the underwater robots of the LIRMM Explore team.

An IMU measures the robot's attitude (ϕ, θ, ψ), rotational speeds (p, q, r) and accelerations of the robot. These accelerations are of poor quality, as the sensor we use is of very low cost. The pressure sensor measures the ambient pressure around the robot and correlates it with the depth of the robot (z), since the pressure increases by 1 bar every 10 m under water. We also derive the pressure sensor measurement to have a heave velocity measurement (w) when ϕ and θ are small enough.

The *usbl_module* The *usbl_module* runs the *usbl_driver*. The USBL (Ultra-Short BaseLine) coupled with the surface GPS measures the 3D absolute position of the robot (Fig 4.6). The surface beacon (transducer) pings the embedded beacon (transponder) which replies with a predefined acoustic signal. Thanks to its 3 antennas, the transducer is then able to calculate the relative position of the transponder. Finally, the transducer sends its GPS coordinates and the xy transponder's relative position to the transponder. The *usbl_module* obtains a measured depth at a frequency of 10 Hz. However, a new xy position is received every 4 seconds at best, due to the message exchange and processing time between the 2 USBL beacons.

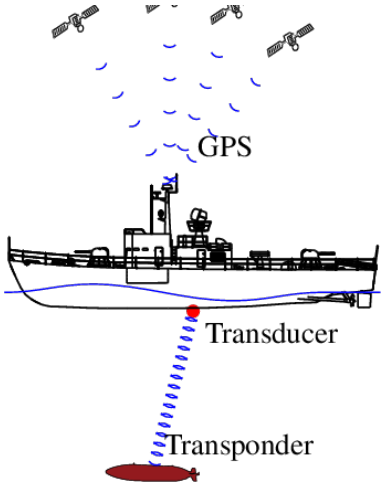


FIGURE 4.6: Principle of the coupling between USBL and surface GPS
Source: [84]

The *gps_module* The *gps_module* runs the *gps_driver* and receives information from the embedded Global Positioning System (GPS) sensor at a frequency of 1.25 Hz at best, when the robot surfaces. The GPS provides the latitude and longitude of the robot based on the WGS 84 model. If the *gps_module* or the *usbl_module* position the robot more than 10km from the origin O_n of {n}, then O_n is replaced by the new measured position of the robot. Consequently, the robot can navigate in a 10 km radius disk with the same NED origin.

The *dvl_module* The *dvl_module* runs the *dvl_driver* and receives information from the DVL (Doppler Velocity Log) sensor at a frequency of 10 Hz at best. The Doppler effect allows to calculate the speed of a moving object thanks to the phase shift of an acoustic wave. The DVL sensor uses this effect to measure the relative speed of a reference in 3 directions and the distance to it. As we have oriented the sensor downwards, it can therefore measure the speed of the robot, and the altitude.

The *echo_module* The *echo_module* runs the *echosounder_driver* and receives information from the echosounder sensor at a frequency of 10 Hz at best. The echosounder measures the altitude of the robot by sending an acoustic wave directed towards the seabed and calculating the response time of the echo.

The *opt_module* The *opt_module* runs the *camera_driver* and receives information from the vertical camera at a frequency of 10 Hz at best. The vertical camera continuously films the seabed and uses the optical flow method to estimate the speed of the robot from a sequence of images over time. Although we represented it in the architecture, we did not use this sensor in the experimental validation of the framework. This sensor did not demonstrate an acceptable performance in the unit test, mainly due to poor camera quality and water turbidity.

All the sensor modules provide the measurement information from the sensors to the *nav_module*, which we detail in the next section.

4.2.2.2 The *nav_module*

We define the navigation state χ of the robot as the robot speed in {b} associated with the robot's pose in {n} (Eq 4.1). The objective of the *nav_module* is to provide an estimate of the navigation state of the robot and an estimate of the altitude (*alti*).

$$\chi(t) = [\nu(t), \eta_q(t)]^T \tag{4.1}$$

We choose to implement an Extended Kalman Filter (EKF) ([162]) to estimate this state in real time. The EKF first provides a state prediction based on a nonlinear dynamic model of the robot described in Appendix B. The EKF then corrects this prediction by merging it with the new information provided by the sensor modules. The implementation of the EKF is described in Appendix C. An EKF is less accurate than an Unscented Kalman Filter (UKF) or a particle filter but is easier to implement and consumes less computational resources ([3]).

This module provides the state to the *task_module* that we describe in the next section.

4.2.2.3 The *task_module*

The *task_module* runs the *task_manager*, the *guidance*, the *control* and the *actuation_dispatcher*. Its final purpose is to calculate the desired Pulse Width Modulations (PWMs) of the motors to perform the current task:

- the *task_manager* handles the high level of the robotic task and defines the desired trajectory to perform. The robot trajectory is defined by a target point and a motion mode, e.g. desired path or u_{cruise} (desired cruise speed).
- from the desired trajectory and the estimated state, the *guidance* computes the desired speed ($\nu_{desired}$) in {b}. A Virtual target Point (VP) is set if the desired path must form a straight line between 2 points (Fig 4.7). The distance K_{path}, set by default to 1 m, represents the distance between the robot projection on the virtual line and VP. Given K_{pos} the position gain and $\delta^{VP,\{b\}}$ the relative position error between VP and the robot in {b}, the desired speed is calculated as:

$$\nu_{desired} = min(K_{pos} \times \delta^{VP,\{b\}} \ , \ u_{cruise}) \tag{4.2}$$

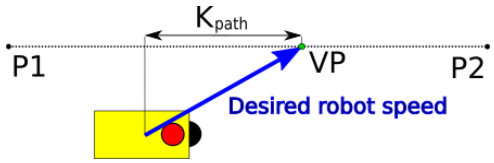


FIGURE 4.7: Calculation of the desired speed for a straight line motion between 2 points

- from the desired speed, the *control* computes the desired forces that the actuators must apply in {b} ($F^{\{b\}}_{desired}[6]$). To achieve this, a PID controller first computes $F^{\{b\}}_{PID}[6]$, taken as input the difference between the current speed and the desired speed in {b}:

$$F^{\{b\}}_{PID}[6] = PID(\nu - \nu_{desired}) \tag{4.3}$$

We note $K_{spe}[6]$ the proportional gains of the PID controller (these gains will be used later in the manuscript). $F^{\{b\}}_{PID}[6]$ are then subtracted by the estimated friction and static forces that apply on the robot in order to obtain $F^{\{b\}}_{desired}[6]$:

$$F^{\{b\}}_{desired}[6] = F^{\{b\}}_{PID}[6] + D(\nu)\nu + g(\eta) \tag{4.4}$$

with $D(\nu)$ the hydrodynamic matrix and $g(\eta)$ the hydrostatic vector (see Appendix B).

- from $F^{\{b\}}_{desired}[6]$ and the Moore-Penrose pseudo-inverse of the actuator configuration matrix A ([138]), the *actuation_dispatcher* distributes the desired forces to each actuator ($F^{motor}_{desired}[8]$):

$$F^{motor}_{desired}[8] = A^T(AA^T)^{-1}F^{\{b\}}_{desired}[6] \tag{4.5}$$

The desired PWMs are then calculated for each motor from its characteristics.

The *task_module* provides the desired calculated PWMs to the *motor_driver* which sends this information to the 2 powerswitches (PWS1 and PWS2), which control the motors through speed controllers. In return, the *motor_driver* also receives the voltage and current at the terminals of each powerswicth.

4.2.2.4 The *mis_module*

The *mis_module* runs the *mission_manager* which selects the robotic task to launch from the knowledge of the current robotic task, the mission desired by the user, the current state of the robot and the undesired events provided by our fault tolerance manager. Mission management constitutes a research topic in itself ([166]).

For this thesis, the *mission_manager* is implemented as a simple state machine, where each state (or mission) is a sequence of one or more of the following robotic tasks:

- **stop_task**: the actuators perform a controlled stop.
- **stay_task**: the robot maintains itself in the same pose, despite external factors.
- **surface_task**: the robot surfaces using a constant heave force.
- **teleop_task**: the robot can be controlled by an operator with a joystick.
- **go_to_pose_task**: the robot moves to a desired pose with a desired motion mode.
- **go_to_ship_task**: the robot heads towards the ship.
- **test_actuators_task**: the actuators are activated one by one with a constant PWM for a few seconds.
- **specific tasks**: these tasks are implemented so that the robot achieves the objectives of the biological mission. We will address this point in the section 4.3.

4.2.2.5 The *comm_module*

The *comm_module* runs the *surface_comm* which communicates with the surface PC using the Mavlink protocol. The user receives in real time the supervision information (e.g. navigation status, video feedback, status of the robotic task in progress). From the user interface on the surface PC, the user has the possibility to define and launch a mission, or to remotely control the robot in teleoperation mode using a joystick. We use the QGroundControl interface (Fig 4.8), an interface already employed to control the blueROV (Fig 4.1a) marketed to the general public.

In the following section, we describe the implementation of the Fault Tolerance Manager (FTM) in the robot, where are implemented the fault tolerance principles proposed in chapter 3.

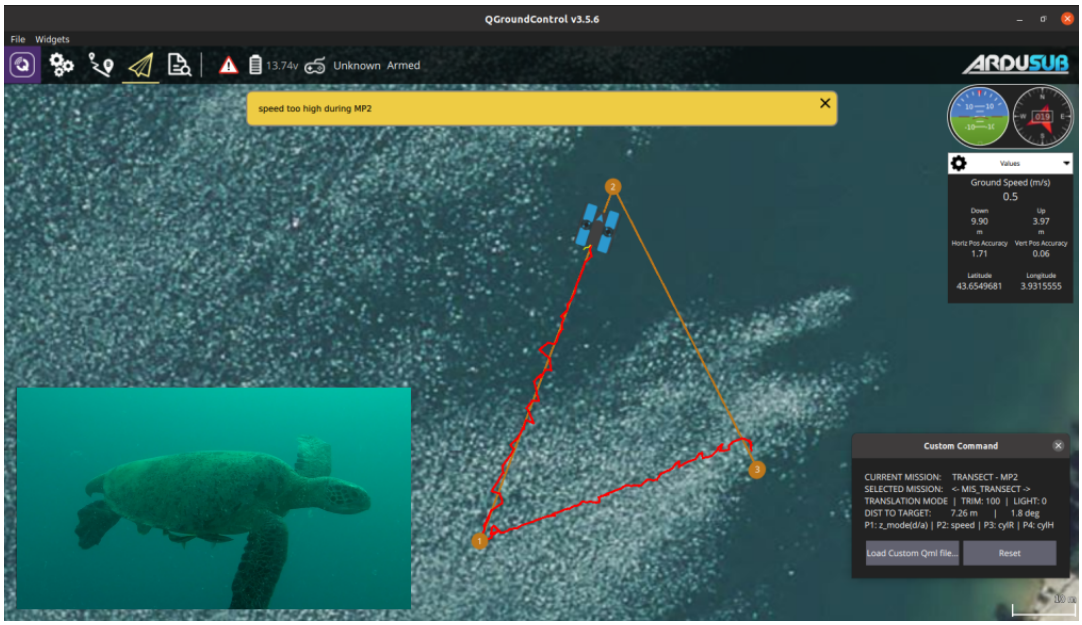


FIGURE 4.8: The user interface QGroundControl during a simulated mission

4.2.2.6 Implementation of the Fault Tolerance Manager

The FTM is deployed as an additional module beside the standard control loop architecture (Fig 4.9). The FTM collects the information from the *comm_module*, *sensor_module* (x6), *mot_module*, *nav_module* and *task_module*, which is used by the detection algorithms. From the detection of undesired events, the FTM infers the non-detectable undesired events based on the logic of the implemented fault tree (which are described later in this chapter).

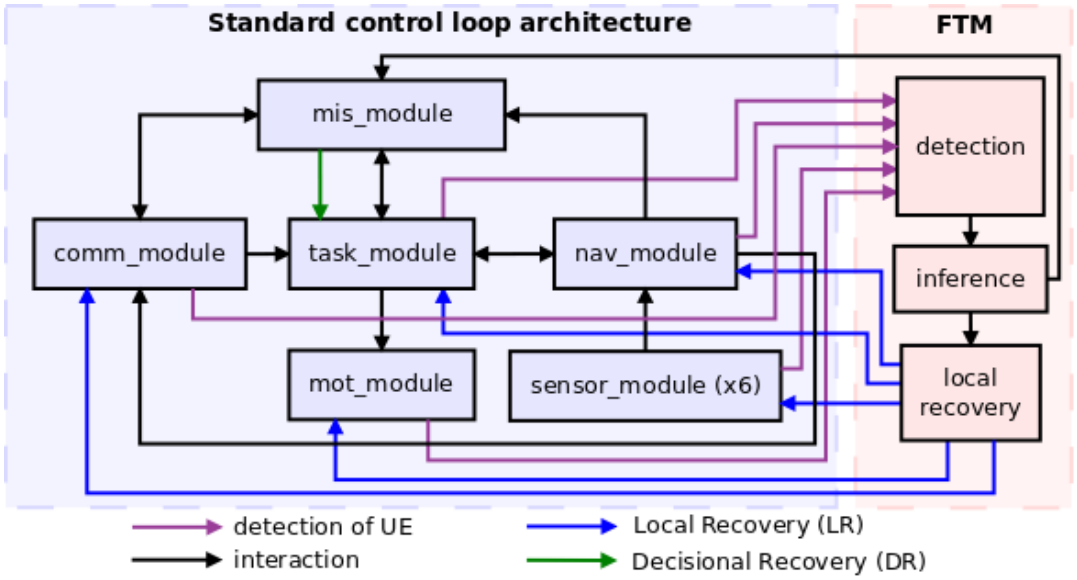


FIGURE 4.9: The control loop architecture with the FTM

From the detection and inference processes, several recoveries can then be carried out. Local recoveries can be initiated to locally correct or prevent undesired events by interacting directly with the concerned modules. A decisional recovery can also be triggered by informing the *mission_manager* when a set of undesired events triggered a top event. The *mission_manager* can then change the task to prevent a more critical undesired event to occur, or because it is impossible to complete the mission successfully.

We choose to run the FTM with a periodicity of 100 ms, the same periodicity as the nav_module and task_module. This periodicity is sufficient to react quickly to possible malfunctions. The FTM is implemented in C++, as illustrated in Fig 3.8 in the previous chapter.

The robotic system is therefore composed of a multitude of modules and communication protocols. These different hardware and software components are unfortunately susceptible to malfunction. These malfunctions can lead to undesirable consequences on the mission and safety, despite the presence of hardware redundancies of the actuators and sensors (Tab 4.2).

After the above description of the system architecture, we will focus on the description of the mission in the next section.

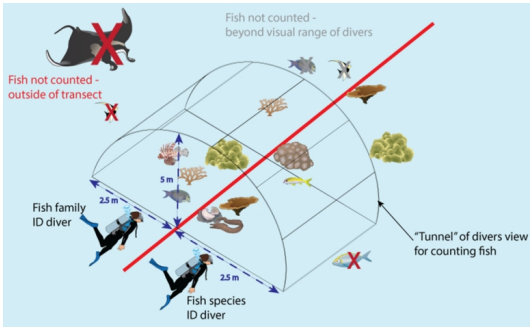
4.3 Mission description: the underwater transect

In this section, we will first start from the environmental context and the biologist needs, before addressing the description of the robotic mission that is used as a support for our case study, to finally detail the constraints related to the specifications

4.3.1 An approach to observe the marine environment

Climate change and the desire to better understand the marine ecosystem make marine observation an important issue ([115]). At a depth of less than 100 meters, a marine ecosystem can rely on the natural sunlight. Small fishes usually live in these shallow areas where coral is present owing to photosynthesis. Two main types of observation are carried out. A first observation is the study of the fish species, their analysis, size and density. A second approach is to study sessile organisms, such as coral or algae. Both observations are good indicators of the health of an ecosystem. Nevertheless, the observation of the marine environment is still a challenge for the biologists around the world ([163]) as it is necessary to have reliable hardware platforms in addition with handling a large amount of data.

There are 2 main techniques for visual census of fish species ([144]): static observation and transect. They have to be deployed in adequate area to have the best representation of species, whose number can differ from one place to an other ([28]). The ultimate purpose of these observations is to map and assess species through tools, which further support biologists in their research ([106]). Static observation is often done with an underwater camera, which films a location of interest to biologists for several hours. As opposed to static observation, transect is a dynamic observation. During a transect, one or several divers follow a straight line of several tens of meters in a specific area and evaluate each species encountered (Fig 4.10). During the course of the straight line, the species, their sizes, their weights and their behaviors are thoroughly noted in a notebook.



Source: [159]



Source: observatoire.criobe.pf

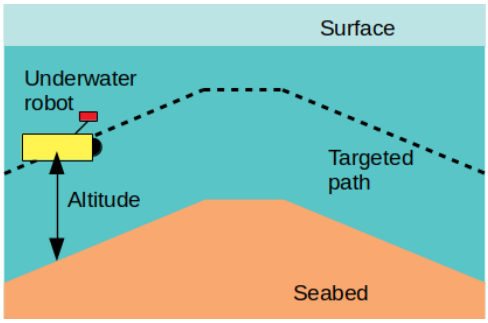
FIGURE 4.10: A transect performed by divers

Compared to a static observation, a transect has the advantage of potentially observing a larger number of species. However, a bias may be present in the observation due to the divers' motions, that may scare away shy species and attract curious ones. In addition to the danger for the divers to remain underwater for long periods at great depths, the information noted in the notebook may be misinterpreted or misobserved. For these reasons, robotics is a good candidate to perform underwater transects, which we will see in the next section.

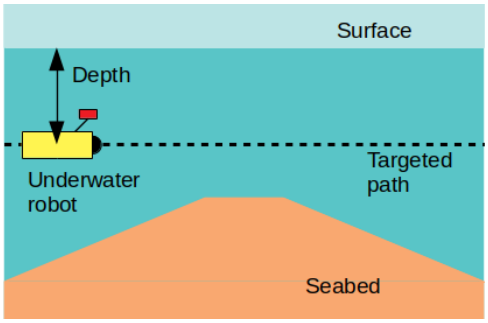
4.3.2 Overview of the robotic transect mission

Thanks to the recent development of technologies and a growing interest in marine exploration, we have addressed in the section 4.1 that it is currently possible to perform underwater robotic missions with autonomous underwater robots. The work presented in this thesis is part of the BUBOT project ([23]) which aims to develop new and innovative tools for marine biodiversity observation and their usage to evaluate anthropic impact on marine reef environment.

We start from the work of [96] to characterize our robotic mission. The authors divide the transect mission into 3 simple tasks: diving, moving along the transect line and surfacing. The transect line can be traveled at a constant depth or at a constant altitude (Fig 4.11). The surge speed of the robot and its attitude must remain constant during the transect.



(A) At constant altitude



(B) At constant depth

FIGURE 4.11: An UUV performing a transect

In order to communicate between the different stakeholders of the project (biologists, operators and programmers) and to establish the outline of the specifications, we build a UML use case diagram of the system performing a transect mission (Fig 4.12).

The final goal of the system is to allow biologists to analyze the videos of the stereo cameras associated with the logs of the robot, to determine the species encountered during the transect. Both biologists and operators can program the mission via the QGroundControl interface having previously defined the mission in a file or in real time. The mission is defined by 2 GPS positions and a target depth or target altitude defining the Transect Line (TL). The operator need to initialize the system, choose the mission, and can observe the mission in real time and interrupt it if necessary. We also require that the developer has access to detailed system logs to fix potential bugs and to improve the system.

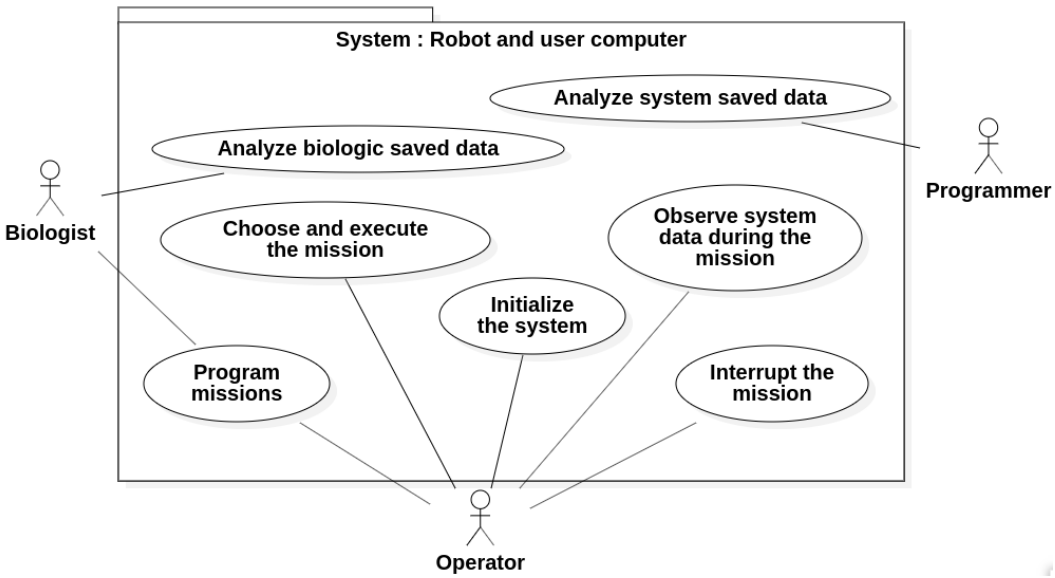


FIGURE 4.12: Use case diagram of the system

Given the initial expectations of the system, we divide the transect mission into 4 Mission Phases (MPs), performed in sequence and illustrated in Fig 4.13:

- MP0 (initialization phase): the robot verifies that it is ready to execute the mission. The robot dives to its current location to a depth of 3 m, to verify that it receives a first signal from the USBL surface beacon.
- MP1: the robot must reach the Starting Point (SP) of the transect. SP is given by a latitude, a longitude and a depth or an altitude.
- MP2: during this central task of the transect mission, the robot must follow TL from SP to the Ending Point (EP), given by a latitude, a longitude and the same depth or altitude than SP.
- MP3: for the last task of the mission, the robot must return to the surface ship, defined by a GPS point on the QGroundControl user interface.

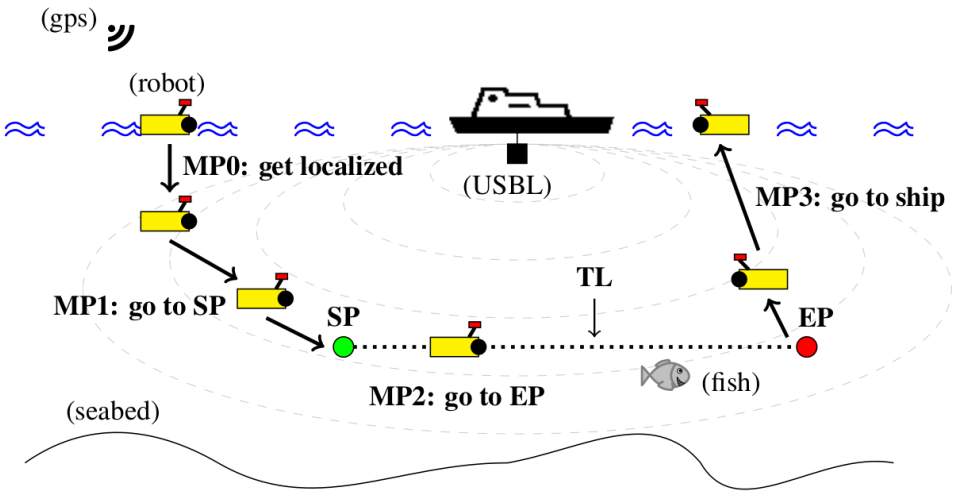


FIGURE 4.13: A transect mission performed by our underwater robot

In order to clarify these specifications, we will address in the next section that it is necessary to establish constraints related to the mission.

4.3.3 Definition of the mission constraints and mission properties

In this section, we detail the constraints and properties that the robot must respect during the 4 phases of the mission. Constraints and properties must be defined with the project actors (biologists, developers and operators). The mission properties mathematically define the constraints of the mission, partly established in the specifications. The description of the mission and the elaboration of the constraints and properties that have to be respected by the system during the mission will then allow to define the undesirable events related to the mission fault tree.

We associate each mission constraint with an ID. As addressed in section 3.2.1, we identify several classes of properties: spatial properties (S), temporal properties (T), energetic properties (E) and informational properties (I). A mission constraint is associated with one or more properties, and with a reaction from the mission manager in case of validation or violation of the associated properties. For all MPs, we note δt the time since the beginning of the current MP, δ^{TL} the distance between the robot and TL.

Proceeding by mission phase, we detail in the following sections the mission constraints and the corresponding reactions in case of their violation.

4.3.3.1 Mission constraints related to MP0

We list the constraints and properties that must be respected during MP0 in Tab 4.4. The different thresholds used in MP0 are listed in Tab 4.5.

We identify 3 mission constraints during MP0. If C0_2 is violated, the decision to surface is taken by the mission manager, because it considers that the robot will not be able to catch the USBL signal in all circumstances. The mission manager does not react if C0_3 is violated, because we did not set up the TL detour during MP0 and MP1. This property is therefore regularly violated in practice.

Id	Class	Description & associated properties	Reaction
C0_1	I	The USBL signal is caught: $USBL_rcvd = true$	C0_1 $\rightarrow$ next MP
C0_2	T	MP0 does not last too long: $\delta t < \delta t^{MP0}$	$\overline{\text{C0_2}}$ $\rightarrow$ surface
C0_3	S	The robot does not cross TL: $\delta^{TL} > \delta^{TL}_{fear_zone}$	

TABLE 4.4: Mission constraints related to MP0

Designation	Description	Value
$\delta t^{MP0\,max}$	Maximum time allowed to perform MP0	20s
$\delta^{TL}_{fear_zone}$	Minimum distance allowed to TL to avoid frightening the fish before executing MP2	3m

TABLE 4.5: Thresholds and constants used in MP0

4.3.3.2 Mission constraints related to MP1

We identify 5 mission constraints during MP1 that we list in Tab 4.6 with their associated thresholds in Tab 4.7.

Id	Class	Description & associated properties	Reaction
C1_1	S	SP is reached: $\delta^{SP}_{xy} < \delta^{Point}_{xy\,reach}$ $\delta^{SP}_{z} < \delta^{Point}_{z\,reach}$	C1_1 $\rightarrow$ next MP
C1_2	S,T	MP1 does not last too long: $\delta t < \frac{d^{MP1}}{u^{MP1}_{min}} + \delta t^{MP1}$	$\overline{\text{C1_2}}$ $\rightarrow$ go to ship
C1_3	S	The robot does not cross TL: $\delta^{TL} > \delta^{TL}_{fear_zone}$	
C1_4	S	The robot's localization is precise: $P(x) < P^{xy\,MP1}_{max}$ $P(y) < P^{xy\,MP1}_{max}$ $P(z) < P^{z\,MP1}_{max}$	$\overline{\text{C1_4}}$ $\rightarrow$ surface
C1_5	E,S	The robot has enough energy to finish MP1: $V_{PWS1} > C_v \times \delta^{SP} + V_{min}$	$\overline{\text{C1_5}}$ $\rightarrow$ go to ship

TABLE 4.6: Mission constraints related to MP1

To characterize C1_1, we note δ^{SP} the distance between the robot and SP. To characterize C1_2, we note d^{MP1} the distance to travel during MP1. As for MP0, if the robot takes too much time to perform MP1, the decision to go to the ship is taken by the mission manager because it considers that the robot will not be able to reach SP in all circumstances.

To characterize C1_4, we note $P(x)$, $P(y)$ and $P(z)$ the variance associated to x, y, and z calculated by the EKF. If the robot is not located accurately, the mission manager considers that the robot might not follow the trajectory defined for the mission and therefore takes the decision to surface to locate the robot with the embedded GPS. Finally, to characterize C1_5, we note V_{PWS1} the voltage at the terminals of PWS1. If there is not enough energy to finish the current phase, the decision to go to the ship is taken by the mission manager, so that an operator can replace the batteries.

Name	Description	Value
$\delta^{TL}_{fear_zone}$	Minimum distance allowed to TL to avoid frightening the fish before executing MP2	$3m$
$\delta^{Point}_{xy\,reach}$	Maximum distance to be arrived at a point on the horizontal plan	$1.5m$
$\delta^{Point}_{z\,reach}$	Maximum vertical distance to be arrived at a point	$0.5m$
u^{MP1}_{min}	Minimum average speed that the robot should attain during MP1	$0.5 \times u^{MP1}_{cruise}$
δt^{MP1}	Additional time allowed for maneuvering	$18s$
$P^{xy\,MP1}_{max}$	Maximum variance associated to x, y tolerated during MP1	$15m^2$
$P^{z\,MP1}_{max}$	Maximum variance associated to z tolerated during MP1	$0.5m^2$
C_v	Coefficient associated to the estimated loss of voltage per meter	10^-3
V_{min}	Minimum voltage to keep at the end of the MP	$11V$

TABLE 4.7: Thresholds and constants used in MP1

4.3.3.3 Mission constraints related to MP2

We list the constraints and properties that must be respected during MP2 in Tab 4.8 and their thresholds in Tab 4.9.

To characterize C2_1, we note δ^{EP} the distance between the robot and EP. To characterize C2_2, we note d^{MP2} the distance to travel during MP2. As for MP1, if the robot takes too much time to perform MP2, the decision to surface is taken by the mission manager, because it considers that the robot will not be able to reach EP in all circumstances. To characterize C2_3, dt^{TL}_{MP2} was added after the first tests because the robot's position estimation could be temporarily out of the acceptable range when a new USBL measurement reports the robot outside the acceptable area.

To characterize C2_9, we note R_{missed} the ratio between the volume that has not been filmed but that should have been filmed since the beginning of MP2 (or the missed volume since the beginning of MP2), on the total volume that the robot should normally film during MP2. The filmed volume is constructed by considering that the front stereo cameras film a 4 m square at a distance of 4 m. This constraint was also used in the work of [96] with $cov^{MP2}_{min} = 90\%$. At the time, this constraint

was easier to respect in estimation, because there were no absolute localization sensors underwater.

ID	Class	Description & associated properties	Reaction
C2_1	S	EP is reached: $\delta_{xy}^{EP} < \delta_{xy\,reach}^{Point}$ $\delta_{z}^{EP} < \delta_{z\,reach}^{Point}$	C2_1 → next MP
C2_2	S,T	MP2 does not last too long: $\delta t < \frac{d^{MP2}}{u_{min}^{MP2}} + \delta t^{MP2}$	$\overline{C2_2}$ → surface
C2_3	S	The robot is near TL: $\exists t' \in [t - dt_{MP2}^{TL}, t], \delta^{TL} < \delta_{MP2}^{TL}$	$\overline{C2_3}$ → restart MP2
C2_4	S	The robot attitude is adequate: $\lvert\phi\rvert < ang_{max}^{MP2}$ $\lvert\theta\rvert < ang_{max}^{MP2}$ $\lvert\psi - \psi_{targeted}^{MP2}\rvert < ang_{max}^{MP2}$	$\overline{C2_4}$ → go to ship
C2_5	S,T	The surge velocity is not too high $u < u_{max}^{MP2}$	$\overline{C2_5}$ → go to ship
C2_6	S,T	The acceleration is not too high: $\exists t' \in [t - dt_{MP2}^{acc}, t], \lvert acc_u\rvert < acc_{max}^{MP2}$ $\exists t' \in [t - dt_{MP2}^{acc}, t], \lvert acc_v\rvert < acc_{max}^{MP2}$ $\exists t' \in [t - dt_{MP2}^{acc}, t], \lvert acc_w\rvert < acc_{max}^{MP2}$	$\overline{C2_6}$ → go to ship
C2_7	S,T	The surge velocity is not too low: $\exists t' \in [t - dt_{MP2}^{u}, t], u > u_{min}^{MP2}$	$\overline{C2_7}$ → go to ship
C2_8	S,T	The robot angular speed is not too high: $\lvert p\rvert < \omega_{max}^{MP2}$ $\lvert q\rvert < \omega_{max}^{MP2}$ $\lvert r\rvert < \omega_{max}^{MP2}$	$\overline{C2_8}$ → go to ship
C2_9	S,I	The robot films the appropriate volume: $R_{missed} < 1 - cov_{min}^{MP2}$	$\overline{C2_9}$ → restart MP2
C2_10	S	The robot's localization is precise: $P(x) < P_{max}^{xy\,MP2}$ $P(y) < P_{max}^{xy\,MP2}$ $P(z) < P_{max}^{z\,MP2}$	$\overline{C2_10}$ → surface
C2_11	E,S	The robot has enough energy to finish MP2: $V_{PWS1} > C_v \times \delta^{EP} + V_{min}$	$\overline{C2_11}$ → stop

TABLE 4.8: Mission constraints related to MP2

Compliance with constraints C2_3 to C2_10 certifies the good quality of the transects for the biologists. Indeed, C2_3, C2_4, C2_9 and C2_10 specify the required position of the robot with respect to TL. C2_5 and C2_6 specify how to avoid frightening the species during MP2. C2_8 ensures a good video quality. Thus the non-respect of these constraints leads to the uselessness of the mission from a biological point of view. In that case, it is no longer necessary to pursue it.

Name	Description	Value
$\delta^{Point}_{xy\,reach}$	Maximum distance to be arrived at a point on the horizontal plan	$1.5m$
$\delta^{Point}_{z\,reach}$	Maximum vertical distance to be arrived at a point	$0.5m$
u^{MP2}_{min}	Minimum average speed that the robot should attain during MP2	$0.5 \times u^{MP2}_{cruise}$
δt^{MP2}	Additional time allowed for maneuvering	$10s$
δ^{TL}_{MP2}	Maximum distance allowed to TL	$2m$
dt^{TL}_{MP2}	Time constraint	$10s$
ang^{MP2}_{max}	Maximum roll, pitch, and yaw deviation tolerated	22.5
$\psi^{MP2}_{targeted}$	Desired yaw during MP2	$atan(\frac{EP_y - SP_y}{EP_x - SP_x})$
u^{MP2}_{max}	Maximum speed allowed during MP2	$2 \times u^{MP2}_{cruise}$
acc^{MP2}_{max}	Maximum acceleration tolerated during a given time	$1.0m/s^2$
dt^{acc}_{MP2}	Time during which the acceleration should not be too high	$0.6s$
u^{MP2}_{min}	Minimum surge velocity tolerated during a certain time	$0.3 \times u^{MP2}_{cruise}$
dt^{u}_{MP2}	Time during which the surge velocity should not be too low	$5s$
ω^{MP2}_{max}	Maximum angular velocities tolerated	$45°/s$
cov^{MP2}_{min}	Minimum ratio of filmed volume that should be attained at the end of MP2	90%
$P^{xy\,MP2}_{max}$	Maximum variance associated to x, y tolerated during MP2	$13m^2$
$P^{z\,MP2}_{max}$	Maximum variance associated to z tolerated during MP2	$0.5m^2$
C_v	Coefficient associated to the estimated loss of voltage per meter	10^-3
V_{min}	Minimum voltage to keep at the end of the MP	$11V$

TABLE 4.9: Thresholds and constants used in MP2

4.3.3.4 Mission constraints related to MP3

We list the constraints and properties that must be respected during MP3 in Tab 4.10 and their thresholds in Tab 4.11. To characterize C3_1, we note δ_{xy}^{ship} the distance between the robot and the ship on the horizontal plan.

ID	Type	Description & associated properties	Reaction
C3_1	S	The ship is reached: $\delta_{xy}^{ship} < \delta_{xy\,reach}^{Point}$ $z < \delta_{z\,reach}^{Point}$	C3_1 $\rightarrow$ stop
C3_2	S,T	MP3 does not last too long: $\delta t < \frac{d^{MP3}}{u_{min}^{MP3}} + \delta t^{MP3}$	$\overline{C3_2}$ $\rightarrow$ surface
C3_3	S,E	The robot has enough energy to finish MP3: $V_{PWS1} > C_v \times \delta ship + V_{min}$	$\overline{C3_3}$ $\rightarrow$ stop

TABLE 4.10: Mission constraints related to MP3

Name	Description	Value
$\delta_{xy\,reach}^{Point}$	Maximum distance to be arrived at a point on the horizontal plan	$1.5m$
$\delta_{z\,reach}^{Point}$	Maximum vertical distance to be arrived at a point	$0.5m$
u_{min}^{MP3}	Minimum average speed that the robot should attain during MP3	$0.5 \times u_{cruise}^{MP3}$
δt^{MP3}	Additional time allowed for maneuvering	$18s$
C_v	Coefficient associated to the estimated loss of voltage per meter	10^-3
V_{min}	Minimum voltage to keep at the end of the MP	$11V$

TABLE 4.11: Thresholds and constants used in MP3

From the description of our case study, composed of the underwater robot and the constraints of the transect mission, we will address in the next section how to build the safety, mission and control loop fault trees. We will also focus on the detection processes and recoveries associated to the undesired events identified in the fault trees.

4.4 Fault tree, detection and recovery implementation

It is necessary to have a logical representation of fault trees on which fault tolerance mechanisms can be based in real time. We use the software ArbreAnalyst ([6]) to build the 3 fault trees. This software is able to logically and graphically model classical fault trees using OR and AND gates, to perform a FTA. This simple software

also allows to provide the construction of the fault trees in the form of an eXtensible Markup Language (XML) file, also supported by other FT construction and analysis software. This XML file is then parsed by the framework during the running phase to inform the logic between the undesired events.

In this section, we will first describe how the safety fault tree is implemented, with the associated detection and recovery processes. We will then focus on the mission fault tree and finish with the control loop fault tree.

4.4.1 Implementation of the safety fault tree

From the guidelines detailed in section 3.2.2, we identified several undesired events in the safety fault tree, represented Fig 4.15 and Fig A.8. These events affect the integrity of the robot or of the environment. We also develop the detection processes related to these events and the associated recoveries.

4.4.1.1 Overheating inside the enclosures

The electronics of our underwater robot are located in a condensed space formed by two 15cm diameter sealed tubes. These tubes are made of metal to evacuate the heat from the components during their period of activity, as overheating can damage the components. Overheating is therefore an obvious event in our FT (event SAFE_3).

We verify this event by comparing the internal temperature measured by a thermometer with a critical threshold of 45C beyond which the components are not supposed to operate. If this event is detected, it leads to the deduction of the top event SAFE_0.

The mission manager associates this event with a decisional recovery (DR) and launches the *stop_task* if this event is true, to cool down the robot's enclosures as soon as possible.

4.4.1.2 Collision with an obstacle

Physical contacts or collisions between the robot and the environment can damage both sides. It is therefore recommended to take these events into account (event SAFE_1).

There are two basic ways to detect this event. The first way consists in using specific contact sensors (mechanical bumpers), as on the Pioneer robot ([88]). However, this technology remains unproven for underwater robots. A second way is to use sensors that detect a signal, assimilated to a collision. We therefore use the robot's onboard IMU to detect abnormal acceleration/deceleration. After several tests, we estimated a collision was considered as an abnormal acceleration of at least $2.5ms^{-2}$ during $0.2s$.

The mission manager associates this event with a DR and launches the *stop_task* if this event is true. It is difficult to determine which part of the robot is in collision, therefore a different reaction would be potentially dangerous.

4.4.1.3 In collision zone

Many autonomous robots have presence sensors (or digital bumpers) that indicate the position of obstacles before a potential collision, as on the Pioneer robot ([88]).

It is therefore possible to define zones to be avoided or forbidden around these obstacles, and to define undesired events specifying the presence of the robot in these zones.

With the seabed The seabed is a fixed obstacle that our robot must avoid. Our robot possesses downward pointing acoustic sensors that measure the altitude of the robot, i.e. its distance to the seabed. We therefore decide to define 3 undesired events related to 3 zones close to the seabed (Fig 4.14):

- SAFE_6: the robot is in the critical collision zone of the seabed. This event is detected as true when the estimated altitude of the robot is below 0.5 m. This threshold allows time to react before a potential collision with the seabed, without being restrictive to the mission. The mission manager associates this event with a DR and launches the *surface_task* if this event is true to move the robot away from the seabed. Then, an operator can verify if algae are stuck in the thrusters.
- SAFE_6_1: the robot is in the warning collision zone 2 of the seabed. This event is detected as true when the estimated altitude of the robot is below 0.8 m. This threshold allows time to react when the robot is in the warning zone 2 without being in the critical zone. We decide to associate a local recovery (LR) to the event SAFE_6_1 that we identify as LR9_1, and set its priority to 1. LR9_1 consists in overriding the heave command normally generated by the *task_module* by a fixed negative command (-40 Newtons) that would cause the robot to rise. This LR is not compatible with the *surface_task* because it produces a similar effect to the purpose of the robotic task. We represent the failure of LR9_1 by the event R_SAFE_6_1.
- SAFE_6_4: the desired path is in the warning collision zone 1 of the seabed. This event is detected as true when the target to be reached (VP) has an altitude inferior to 1.0 m. We decide to associate a LR (LR8_1) with a priority of 3 to the event SAFE_6_4. LR8_1 consists in overriding the VP z or $alti$ command normally generated by the *task_module*, by a depth or altitude that sets VP at the upper limit of the collision warning zone 1 of the seabed (1.0 m from the seabed). This threshold allows time to react to SAFE_6_4 and to smooth the evasive motion of the seabed. LR8_1 is not compatible with *surface_task* and LR9_1, as changing the heave command may interfere with changing VP's location. We represent the failure of LR8_1 by the event R_SAFE_6_4.

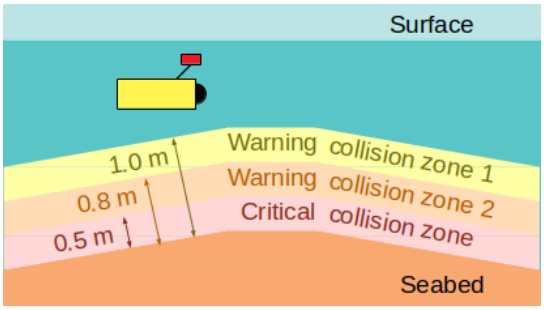


FIGURE 4.14: The danger zones near the seabed

With unforeseen obstacles We also represent the entry into the critical collision zone of an object or coral heads by SAFE_2. Unfortunately, as the robot does not have sensors to detect these elements, we decided to leave this event undeveloped. We consider that when this high criticality event occurs, the operator has to detect it with the help of the front camera and to react by manually interrupting the mission through the user interface.

4.4.1.4 Unsafe pressure on the enclosures

A robot is mechanically designed to operate under a specific maximum pressure. Excessive pressure can damage the sealed enclosure and create a leak, or damage the robot's external sensors. Unfortunately in a marine environment, the pressure increases by one bar every 10 meters. Similarly to the definition of the danger zones near the seabed, we therefore decide to define 3 undesired events related to 3 zones defined by their depth or associated pressure:

- SAFE_5: the robot is in the critical depth zone. This event is detected as true when the estimated depth of the robot is above 82 m (or the external pressure is approximately above 9.2 bars). The robot was normally designed to operate at a maximal depth of 100 m. Nevertheless, we limit this constraint because the seals guaranteeing the waterproofing of the robot's enclosures weaken with time. The mission manager associates this event with a DR and launches the *surface_task* if this event is true to reduce the depth and thus the pressure. An operator can then verify if damage is present on the robot.

- SAFE_5_1: the robot is in the warning depth zone 2. This event is detected as true when the estimated depth of the robot is above 81 m (or the external pressure is approximately above 9.1 bars). This threshold allows time to react when the robot is in the warning zone 2 without being in the critical zone. We decide to associate a LR to the event SAFE_5_1. We identify it as LR9_2 and set its priority to 1. As LR9_1, LR9_2 consists in overriding the heave command by a fixed negative command (-40 Newtons). LR9_2 is not compatible with LR9_1, LR8_1 and the *surface_task*. We represent its failure by the event R_SAFE_5_1.

- SAFE_5_4: the desired path is in the warning depth zone 1. This event is detected as true when the target to be reached (VP) has a depth superior to 80 m (or the external pressure is approximately above 9.0 bars). We decide to associate a LR (LR8_2) with a priority of 3 to the event SAFE_5_4. LR8_2 consists in overriding the VP z or $alti$ command, by a depth or altitude that sets VP at the upper limit of the depth warning zone 1 (80 m of depth). This threshold allows time to react to SAFE_5_4 and to smoothly limit the robot's progress down to the depth. LR8_2 is not compatible with LR8_1, LR9_1, LR9_2 and *surface_task*. We represent its failure by the event R_SAFE_5_4.

4.4.1.5 Water inside the enclosures

Finally, as with any underwater vehicle, water can be present inside the enclosures and damage the electronics (SAFE_4). We therefore use water leak detectors to detect SAFE_4. We note that the presence of water inside the robot does not necessarily come from a leak, but can also be induced by the initial presence of water inside the enclosures.

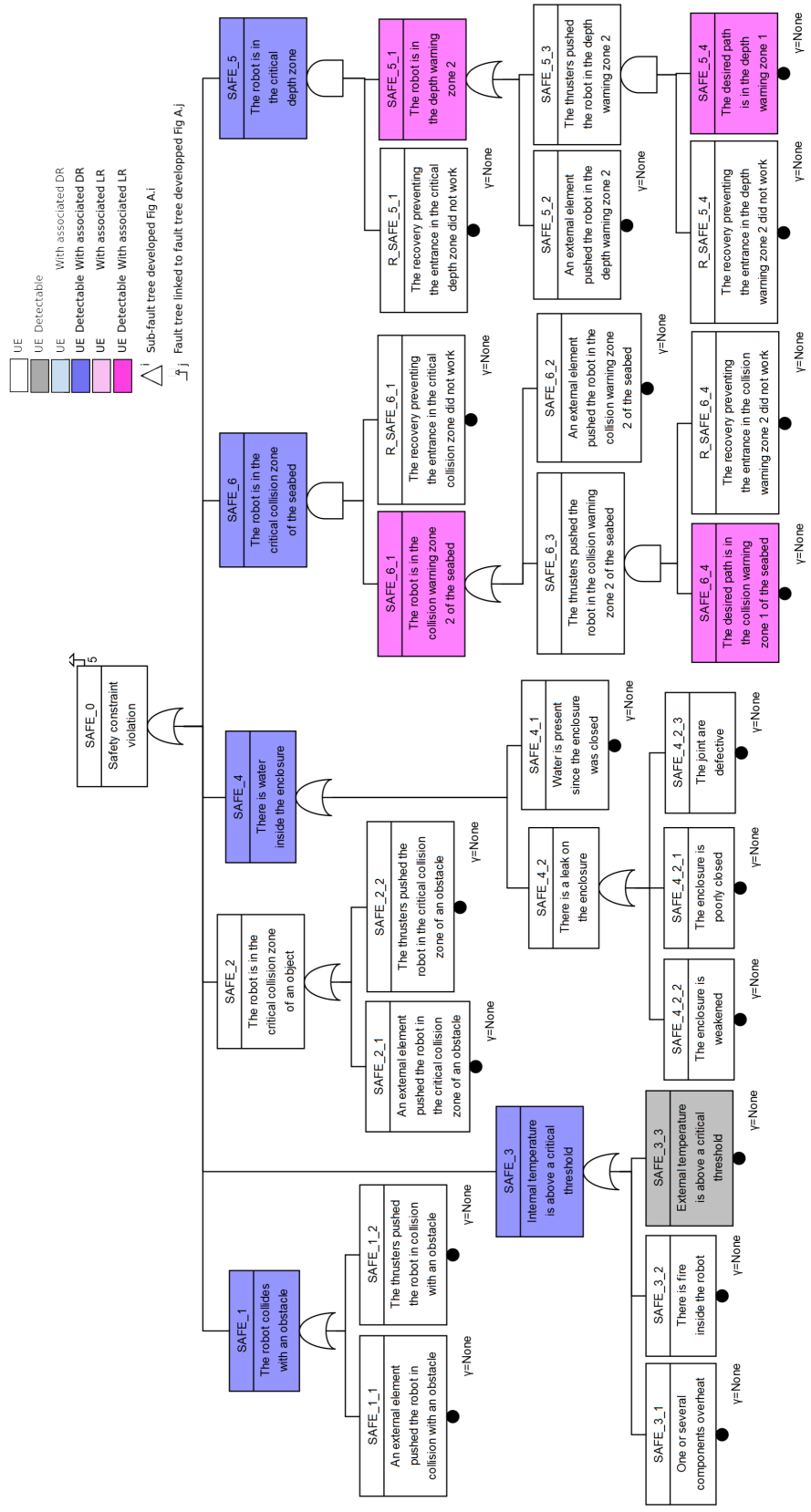


FIGURE 4.15: Safety fault tree

The mission manager associates this event with a DR and launches the *surface_task* if SAFE_4 is true, to limit the potential water entry on the surface and to be pulled out of the water by operators. Another reaction would be to turn off all electronics and let the robot surface using its positive buoyancy. Unfortunately, the robot is not equipped with an electronic master switch.

In the next section, we focus on the implementation of the mission fault tree, detection processes and recoveries of the associated undesired events.

4.4.2 Implementation of the mission FT

As stated in the chapter 3, we build the mission fault tree by listing the property violations by mission phase. The mission FT is presented Fig 4.16 and Fig A.6. Each of the property violations reflects a complex error chain. This chain is difficult to represent usually in a fault tree, because the involved errors are context and time dependent. We therefore consider initially that the property violations are basic events. The mission manager associates each of these UEs with the DR associated with the reaction mentioned in Tab 4.4, Tab 4.6, Tab 4.8 and Tab 4.10.

However, it may be possible to prevent these UEs by implementing LRs. In this thesis, we have implemented only a part of the detection and recovery processes (LRs and DRs). We focus on a few specific examples to illustrate our approach.

We have thus implemented LRs to prevent the robot from moving away from TL (MIS_3_1). Thus, we define a new UE (MIS_3_1_1), which is detected as true if the property C2_3 is violated but with more restrictive thresholds ($dt^{TL}_{MP2} = 6s$ and $\delta^{TL}_{MP2} = 1.5m$). We associate this event to 4 compatible LRs, that are initiated at the same time, named LR4, LR5, LR6 and LR7. LR4 divides the surge proportional velocity gain ($K_{spe}[0]$ introduced in section4.2.2.3) by 2 to be less restrictive on the surge speed, whereas LR5 and LR6 increase the sway and heave proportional velocity gains by 1.3 to be more restrictive on the sway and heave speeds. LR7 divides the path gain K_{path} introduced in section 4.2.2.3 by 2. Combined, these 4 LRs increase the ability of the robot to better react to MIS_3_1_1, and maybe prevent MIS_3_1.

Similarly, we consider that there may be a warning threshold before the robot's velocities and angular velocities are above critical thresholds. The UEs MIS_3_3_1_1 to MIS_3_3_3_1 and the UEs MIS_3_6_1_1 to MIS_3_6_3_1 are therefore set with thresholds of $1.5 \times u^{MP2}_{cruise}$ for velocities and $35°/s$ for angular velocities. We assign a LR to each of these UEs, which intends to reduce the proportional velocity gain according to the direction concerned. For example, the LR associated to MIS_3_3_1_1 (*Robot surge velocity is above a warning threshold*) decreases the surge proportional velocity gain to reduce excessive velocity that may be caused by abrupt control in the current context.

Finally, we also consider that the robot can react locally before the occurrence of MIS_2_1 (*MP1's duration is over a critical threshold*). We define then a new event MIS_2_1_1, which is detected similarly to MIS_2_1 but with different thresholds ($u^{MP1}_{min} = 0.62 \times u^{MP1}_{cruise}$ and $\delta t^{MP1} = 10s$). We associate this new UE with a LR (LR3) that temporally multiplies by 1.5 the surge desired velocity to attain SP in time.

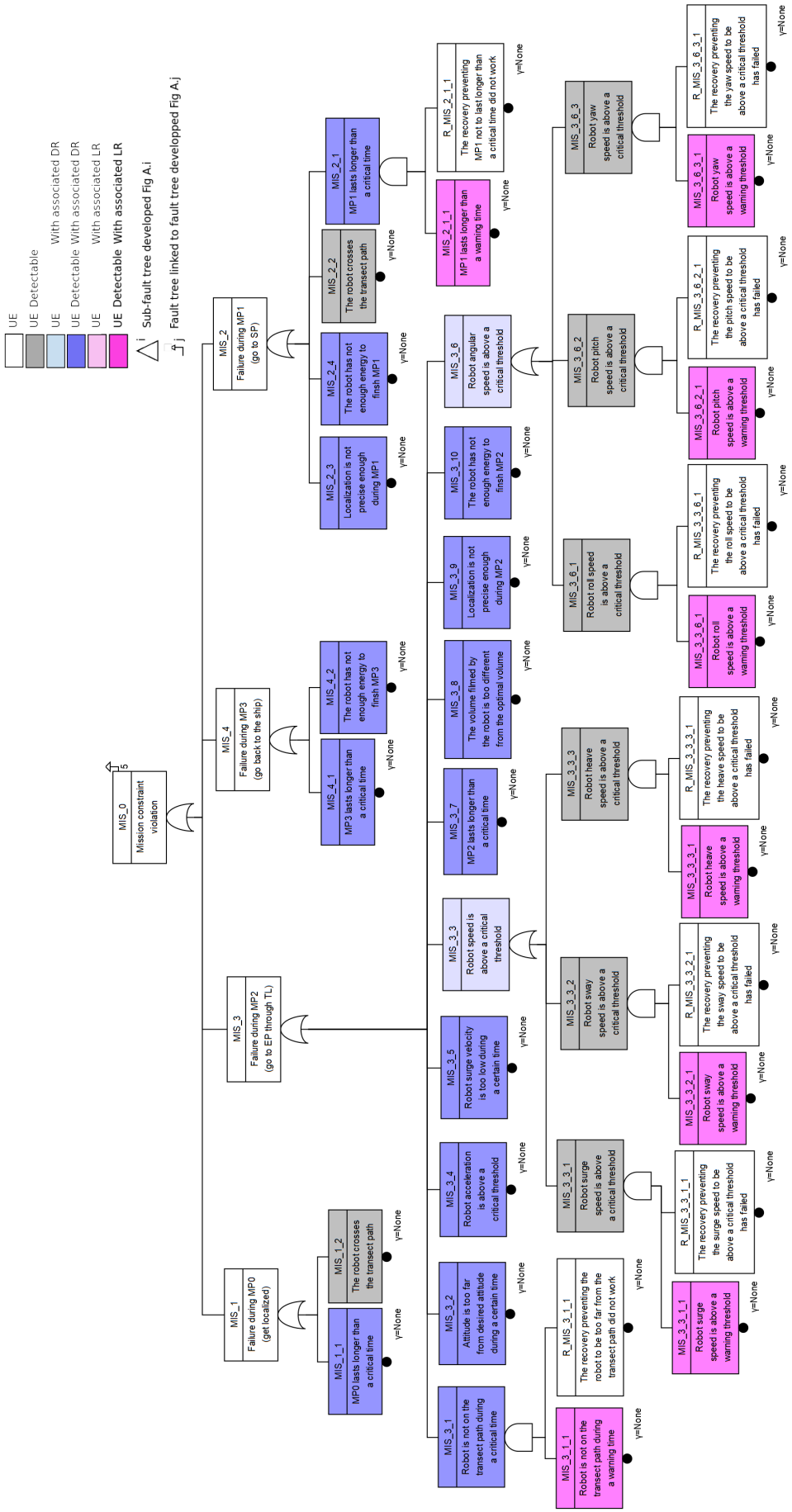


FIGURE 4.16: Mission fault tree

We will now turn our attention to the control loop fault tree which we build from the system description. We will also address the detection and recovery processes of the associated undesired events.

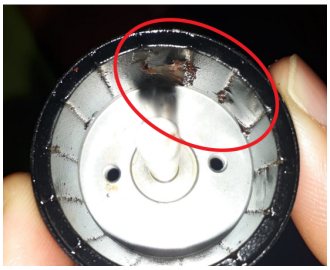
4.4.3 Implementation of the control loop fault tree

The control loop fault tree contains the different undesired events related to a malfunction of the control loop. We define the top event ACTd_PROP_0 of this tree as an invalid propulsion of the actuators (Fig 4.18 and Fig A.1), i.e. the forces generated by the actuators are incorrect. We expand this tree as explained in section 3.2.3.2 and following the architecture of the control loop Fig 4.5.

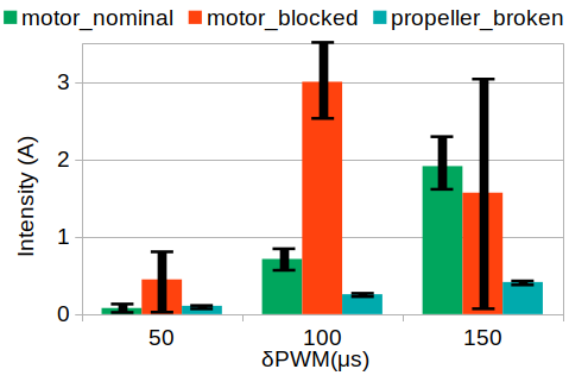
4.4.3.1 Undesired events related to powerswitches and actuators

In order to simplify the construction of the control loop fault tree, we decide to group some modules together. We consider first the UEs related to the set {powerswitches + speed controllers + actuators}. The fault tree first establishes whether the occurrence of the top event comes from an error considered as internal (ACTd_PROP_1) or external (ACTd_PROP_2) to this set.

Among the internal UEs, we have experienced several errors caused by a blockage or a strong slowdown of the engines due to advanced corrosion of the rotors or to an object (e.g. algae) stuck in the propellers (Fig 4.17a). We model a motor blocking, severe motor slowdown, or advanced propeller degradation as an out-of-service actuator (ACTd_PROP_1_1_10 to ACTd_PROP_1_1_17).



(A) The corrosion of a rotor



(B) Intensity measured at the terminal of PWS1 depending on the PWM command of M1

FIGURE 4.17: Detection of a faulty motor

These UEs cannot be detected directly, because there are no speed or torque sensors for the motors. However, powerswitches are equipped with voltmeters and ammeters that allow to measure the activity of a group of motors. Fig 4.17b represents the current (mean and standard deviation) measured at the terminal of PWS1, depending on the absolute value of the difference between the PWM command of M1 (motor1) and the neutral PWM (1500μs) with different status of M1 (nominal, blocked, propeller broken).

Since there is a difference in current measured when an actuator is free (nominal), blocked or when the propeller is broken, we can therefore detect that one or several actuators are faulty among a group of 4 actuators (ACTd_PROP_1_1_8 and

ACTd_PROP_1_1_9). By noting $I_{PWS1\,meas}(t)$ the current measured by PWS1 at a time t and $I_{PWS1\,nom}(t)$ the expected nominal current at the terminals of PWS1 at time t, we detect ACTd_PROP_1_1_8 as true at a time $t0$ if:

$$\forall t \in [t0 - \delta t_{detec}, t0], |I_{PWS1\,meas}(t) - I_{PWS1\,nom}(t)| > \delta I_{max} \tag{4.6}$$

with $\delta t_{detec} = 1.5s$ and $\delta I_{max} = 0.9A$, two thresholds set as the field experiments progressed to minimize false detection (the event is detected true whereas it is false) and non-detection (the event is detected false whereas it is true).

In the context of the current distribution presented Fig 4.17b and the fact that 2 motors can receive very similar commands during a standard motion, it is difficult to diagnose which actuator(s) are faulty during a classical robot motion. This diagnosis can be done if the robot performs the *test_actuators_task*. As a consequence, if ACTd_PROP_1_1_8 or ACTd_PROP_1_1_9 are detected as true while the current robotic task is not the *test_actuators_task*, then event ACTd_PROP_1_1_6 (*the diagnosis of the failed motor cannot be performed*) is detected. The mission manager associates then this event with a DR and launches the *test_actuators_task* with the occurrence of this event.

During this task, ACTd_PROP_1_1_10 to ACTd_PROP_1_1_17 can be detected in the same way as the detection of ACTd_PROP_1_1_8 and ACTd_PROP_1_1_9, as the motors rotate one by one. We decide to associate a LR (LR1) to each event. LR1 consists in recalculating the pseudo-inverse of the actuator configuration matrix A used by the *task_module*, having removed the faulty motors from the model (by putting 0 in the associated column). For example, the actuator configuration matrix of REMI with M2 faulty is (Eq 4.7):

$$A_{\overline{M2}} = \begin{bmatrix} 0.707 & 0 & 0 & -0.707 & 0.707 & 0 & 0 & -0.707 \\ -0.707 & 0 & 0 & -0.707 & 0.707 & 0 & 0 & 0.707 \\ 0 & 0 & 1 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -0.14 & 0 & 0 & -0.14 & -0.14 & 0 \\ 0 & 0 & -0.19 & 0 & 0 & -0.19 & 0.19 & 0 \\ -0.382 & 0 & 0 & -0.382 & -0.382 & 0 & 0 & -0.382 \end{bmatrix} \tag{4.7}$$

If too many motors are detected as faulty, then the A matrix may no longer be of rank 6 and the motions can no longer be executed independently (event ACTd_PROP_1_1_5). This event is detected as true if the determinant of AA^T is inferior to a threshold set to 10^{-5}. This threshold causes the UE to be detected as true if at least 2 horizontal or 2 vertical motors fail. The mission manager associates then ACTd_PROP_1_1_5 with a DR and launches the *stop_task* when this event triggers the deduction of the top event.

Among the external UEs of the set {powerswitches + speed controllers + actuators}, there are two main categories: UEs related to the power supply of the powerswitches (ACTd_2_1) and UEs related to the modules and connections that interact with the powerswitches (ACTd_PROP_2_2). If there is not enough energy to operate the robot (ACTd_2_1_2), the mission manager shuts down the motors to conserve the remaining energy and the robot surfaces with its positive buoyancy.

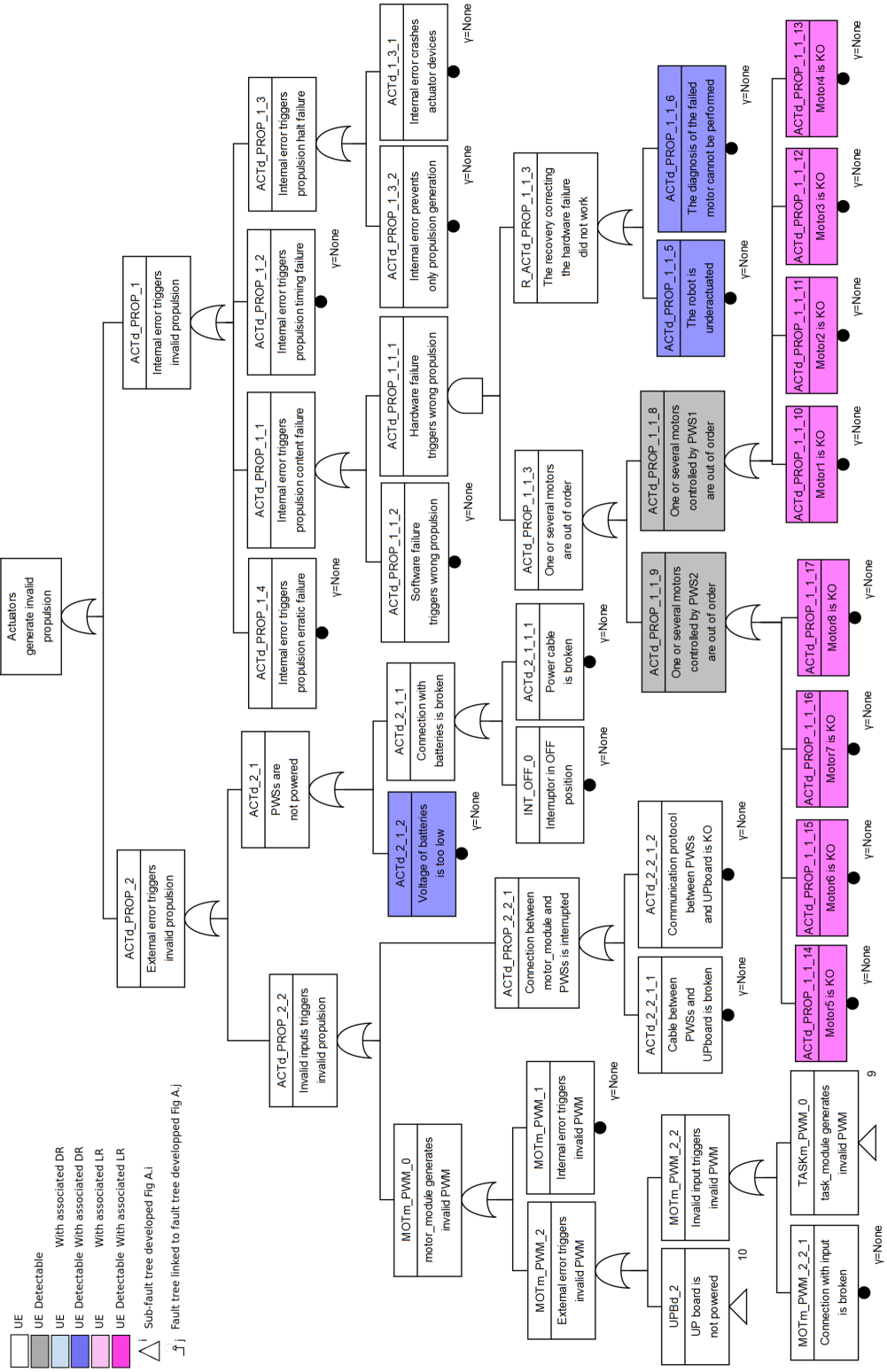


FIGURE 4.18: Control loop fault tree - Actuators module

4.4.3.2 Undesired events related to command

A faulty connection between the *motor_driver* and the powerswitches (event ACTd_PROP_2_2_1 in Fig A.1) is significant, because it causes the robot to become uncontrollable. It can be detected by using a watchdog monitoring a heartbeat in the communication protocol. We could then associate a LR which would restart the *motor_driver* to reestablish the connection. We did not implement this detection for logistic reasons.

The forces generated by the actuators may also be incorrect due to an invalid command sent by the *motor_driver*. We then establish that the error comes either from the driver itself (event MOTm_PWM_1) or from external UEs, and particularly UEs occurring in the *task_module* which calculates the command (events TASKm_PWM_0). The command can be considered as invalid because of an internal error of the *task_module* (event TASKm_PWM_1 in Fig A.9) or because of wrong inputs (event TASKm_PWM_2). We then establish if the error comes from an invalid command from the surface PC. In that case, we identify events UPBd_PC_2_2_1_1 and UPBd_PC_2_2_1 (Fig A.2) as respectfully a warning and critical interruption of the communication between the surface PC and the UPboard controller. These events are detected by using a watchdog monitoring a heartbeat in the communication protocol. The warning threshold is set to 4s and the critical threshold is set to 10s. UPBd_PC_2_2_1_1 is associated to LR10, which restarts the *comm_module*.

We also establish if the error comes from a wrong estimation of the navigation state of the robot (event NAVm_STA_0 in Fig A.9 and Fig A.7), which will be developed in the next section.

4.4.3.3 Undesired events related to navigation

The output of the *nav_module* is the estimated state and the estimated altitude (χ and $alti$) of the robot. We first differentiate between errors considered as internal (NAVm_STA_1 in Fig A.7) and those considered as external (NAVm_STA_2). Among the external errors, we differentiate errors coming from the connection to other modules (NAVm_STA_2_2_1) or related to the information exchange protocol between modules, from errors coming from the other modules themselves (event NAVm_STA_2_2_2).

We divide the UEs that can be the cause of NAVm_STA_2_2_2 into 7 sets of UEs, according to the type of sensor driver outputs (events NAVm_STA_2_2_2_1 to NAVm_STA_2_2_2_7). We detect these UEs as true if the variances of the estimated state, that are the values of the diagonal of the covariance matrix of the estimated state, are above a threshold. This reflects that the estimated state is not accurate, and therefore potentially incorrect. For example, noting $P(x)$ the variance associated to x and $P(y)$ the variance associated to y, NAVm_STA_2_2_2_1 is detected as true if:

$$P(x) > P_{xy\,max} \text{ or } P(y) > P_{xy\,max} \tag{4.8}$$

with $P_{xy\,max} = 16m^2$ the maximal variance tolerated associated to x and y. This threshold has been set from experience in the field. Similarly, we set $P_{alti\,max} = 1m^2$ the maximal variance tolerated associated to $alti$.

The mission manager associates the poor location accuracy in *xy* and *alti* with a DR and launches the *surface_task* if these events are true. The estimated state and the covariance matrix produced by the EKF (Appendix C) are a function of: the occurrence of new sensor measurements, the values of these measurements, the noise associated with these measurements (i.e. the confidence associated with the measurements), the current estimated state and the current covariance matrix.

In order to simplify the representation of the relations between UEs in our fault tree, we consider that NAVm_STA_2_2_2_1 to NAVm_STA_2_2_2_7 are the consequences of the joint invalidity of the drivers' outputs. Under these UEs, we thus represent the invalidity of the sensor drivers outputs, grouped under AND gates by their output types mentioned in Tab 4.2.

4.4.3.4 Undesired events related to sensors and drivers

The output of a driver can be invalid due to an error considered as internal (event MODULEm_OUTPUT_1) or external (MODULEm_OUTPUT_2). In this manuscript, we only represent the UEs related to *dvl_module* in Fig A.3 and to *echo_module* in Fig A.4) for simplification purpose.

Internal errors are defined as mentioned in the section 3.2.3. LRs are set up to locally manage internal UEs. A content failure (event MODULEm_OUTPUT_1_1_1) is handled by a LR that increases the estimated standard deviation associated to the noise of the sensor measurement, or in other terms that decreases the confidence of the sensor measurement.

For example, for the echosounder driver, if ECHOm_ALTI_1_1_1 is detected as true, LR2 is initiated to multiply by 10 the standard deviation of the noise of the echosounder measurement. In our configuration, we use the FilterA (section 4.2.2.1) type to prevent the echosounder driver from providing an altitude value whose estimated noise would be higher than a threshold, set to 0.35 m. Since the minimum standard deviation of the echosounder measurement noise is 0.1m by default, the launching of LR2 causes the blocking of the echosounder driver output.

A timing failure (event MODULEm_OUTPUT_1_2_1) is handled by a LR that increases the frequency of the associated module to attempt to meet the time constraints. A halt failure (event MODULEm_OUTPUT_1_3_1) is handled by a LR that restarts the associated module, in the expectation of obtaining a new output after the restart. We consider that this LR is failed (event R_MODULEm_OUTPUT_1_3_1) if several successive reconnection attempts have failed.

We also define LRs reacting to the connection break between the drivers and the sensors (events UPBd_SENSOR_2_2_1_1). These LRs also aim at restarting the modules managing the drivers to re-establish the lost connection. We consider that these LRs are failed (events R_UPBd_SENSOR_2_2_1_1) if several successive reconnection attempts have failed. We also consider the connection is not re-established after a critical time (events UPBd_SENSOR_2_2_1).

4.5 Conclusion

In this chapter, we first introduced notions about underwater robotics. We then presented our experimental robotic platform. Our hybrid remotely operated vehicle (HROV) is capable of conducting an autonomous mission, or of being controlled by an operator. Our robot has several sensors and motors, managed by drivers on a control architecture. An Extended Kalman Filter allows to estimate the pose and the speed of the robot and to provide this information to the *task_module*. The latter calculates the command to send to the motors to accomplish the task selected by the *mission_manager*.

The fault tolerant manager is implemented on the same controller in parallel to the standard control architecture. It allows to detect and infer undesired events on the one hand, and to launch local recoveries or to inform the *mission_manager* that critical undesired events occur on the other hand. In that last case, the *mission_manager* can initiate a decisional recovery.

We also presented the mission, a dynamic biological observation called transect. We elaborated the specifications in partnership with the biologists, to obtain a set of constraints or properties that the robot must respect during its mission.

We detailed the construction of the 3 fault trees used to implement the fault tolerance mechanisms during the transect mission: the safety fault tree, the mission fault tree and the control loop fault tree. The definition of mechanisms for the detection and recoveries of undesired events is the result of long experimental work through numerous campaigns at sea, but also of valuable exchanges with marine biologists.

In the next chapter, we address the validation of the proposed FTM implemented on our robot for the transect mission.

Chapter 5

Fault tolerance framework: experimental validation

Contents

5.1 Validation of our framework: test objectives 95
5.1.1 System under test and mission context 96
5.1.2 Context of the tests in the field 96
5.1.3 Context of the software in the Loop 97
5.2 Calculating an estimation of the recovery coverage 98
5.3 Unit testing of the FTM functionalities 99
5.3.1 UEs and recoveries of interest 100
5.3.2 Results of the unit testing 103
5.4 Functional and non-regression testing of the system 109
5.4.1 Test cases: mission and environmental context 109
5.4.2 Results of the functional and non-regression testing 111
5.5 Conclusion and discussion 116
5.5.1 Respect to the test objectives 116
5.5.2 Limitations and threats to validity 117

In this chapter, we detail the validation steps of our fault-tolerant framework described in chapter 3, implemented on our underwater robot presented in chapter 4. We first introduce the validation process, we then present the tests performed in simulation and in the field. We finally conclude on the efficiency of our framework and we present the limits of the experimental validation.

5.1 Validation of our framework: test objectives

In this section, we tackle the problematic of validating our fault tolerant mechanism, which heavily relies on testing the system. As seen in section 2.2.4, testing is widely used to remove faults from system during the development phase, but it is also and especially used to validate the behavior of a system.

Our test objectives are:

- to first demonstrate the theoretical improvement of the recovery coverage with a static verification. This work is addressed in section 5.2.
- to perform a series of unit tests to validate the detections of a limited set of Undesired Events (UEs), and to verify that the inference and the associated recoveries are properly initiated. This work is addressed in section 5.3.

- to finally perform a series of functional and non-regression tests on multiple runs with different initial conditions, to verify that the overall system behavior is improved with the Fault Tolerance Manager (FTM). This work is addressed in section 5.4.

It is first necessary to define the system under test and the test context before continuing.

5.1.1 System under test and mission context

The System Under Test (SUT) is constituted of the robot (section 4.2) with the FTM as described in Fig 5.1. We will address in the relevant sections that different faults can be injected in the system without the FTM, depending on the type of test.

We also consider that inputs may eventually cause violations of mission or safety constraints, or failures within the robot. For example, during a constant depth transect, an incompatibility between maintaining the robot at constant depth and not approaching the seabed may occur.

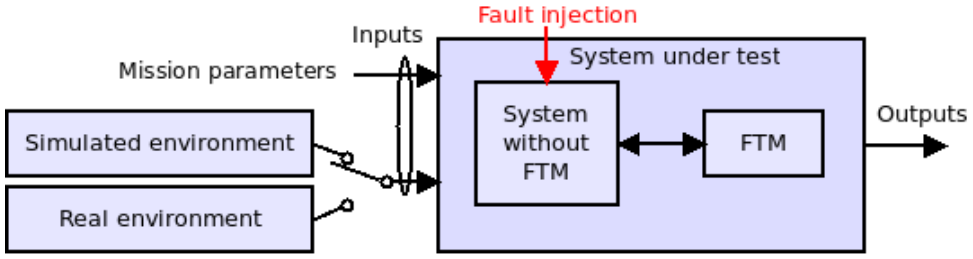


FIGURE 5.1: Testing our robotic system

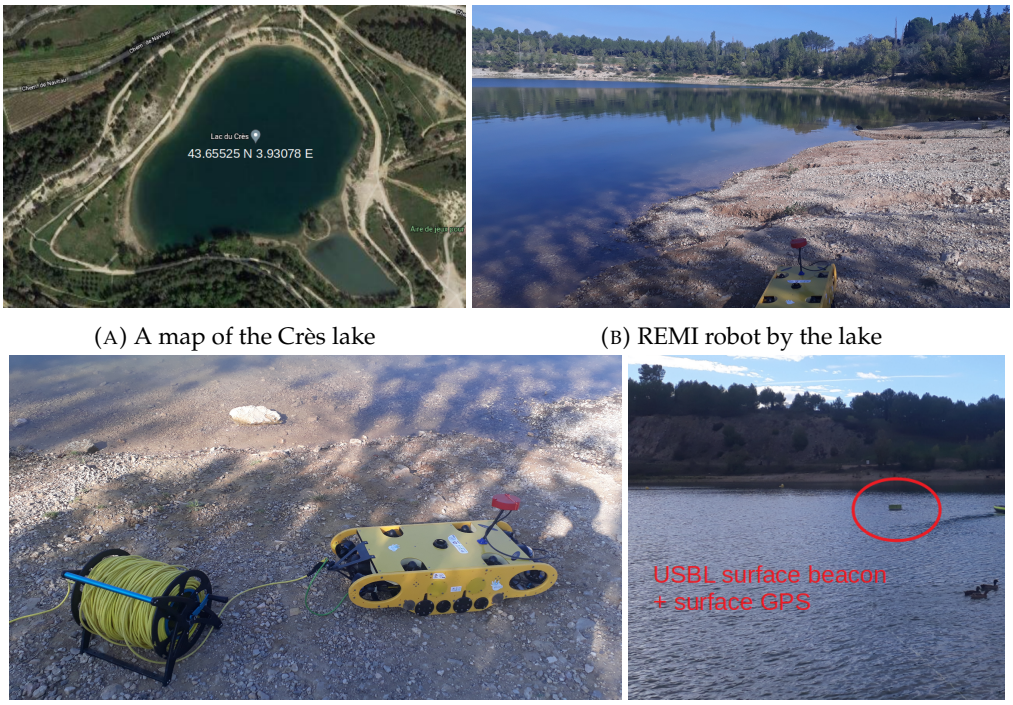
During the various tests, the robot performs transects, divided into 4 phases as described in section 4.3. The mission parameters will be detailed for each type of test in the associated sections. During MP0, the robot dives to catch the USBL signal from the USBL surface beacon. During MP1, the robot navigates to reach the Starting Point (SP), at a specific latitude, longitude and depth/altitude. During MP2, the core process of the transect, the robot navigates to reach the End Point (EP) through the virtual Transect Line (TL). SP and EP are set via the QGroundCOntrol user interface on the user PC. Finally, during MP3, the robot navigates to reach the ship, modeled by a surface point on the QGroundControl interface.

The outputs of the tests are made of the logs saved by the robot during the transect missions. These logs are saved cyclically every 100ms for physical quantities (e.g. pose, speed, forces deployed on the actuators) and in an event-driven manner for events (e.g. change of robotic task, undesired event detected, local recovery initiated).

The tests are carried out on the real robot in the field (TiF) or on the real software running on a simulation PC (SiL), which will be presented now.

5.1.2 Context of the tests in the field

The TiF are performed in the Crès lake near Montpellier. The lake forms a disc on the surface with a radius of approximately 100m (Fig 5.2a) and has a maximum depth (or water column) of 10m, which is sufficient to perform transects.



(A) A map of the Crès lake

(B) REMI robot by the lake

(C) REMI robot and the cable reel

(D) The surface beacons

FIGURE 5.2: The validation test at the Crès lake

During the TiF, the weather conditions must be ideal: wind speed inferior to 5knots, no sea current and no choppy water (Fig 5.2b). We are stationed at the edge of the lake with the user PC and the cable reel (Fig 5.2c), sending mission parameters and instructions to the robot in the lake. We placed the USBL surface beacon on a buoy in the middle of the lake (Fig 5.2d).

5.1.3 Context of the software in the Loop

The SiL are performed with the real software running on a PC with an Intel i7-4930MX processor. The environment is simulated.

The environment simulator is made of additional modules launched directly within the architecture of the robot (Fig 5.3). From the real pose, speed and forces applied on the robot, the *sim_env_module* runs a dynamic model similar to the *nav_module* (Appendix C) to calculate the real state of the robot (pose and speed), but with different physical parameters than those used in Appendix B to reflect that the dynamic model established in the *nav_module* is not perfect.

For example, the gravity is set as 295.0N and the buoyancy as 296.0N in the simulator, whereas it is respectively estimated as 289.1N and 295.0N by the *nav_module*. We also consider that the terms in the hydrodynamic matrix $D(\nu)$ are higher in *sim_env_module*. Depending on the tests, the *sim_env_module* can also integrate a marine current changing with time, a force exerted on the cable depending on the speed and the pose of the robot, and a permanent deviation of the behavior of the thrusters.

During tests in SiL, the seabed is simulated by a surface bounded between 5m and 35m, and following the 2nd degree (Eq 5.1):

$$z = a \times x^2 + b \times x + c \times y^2 + d \times y + e \tag{5.1}$$

with z the water column associated to the x, y coordinates, and a, b, c, d, e random bounded constants.

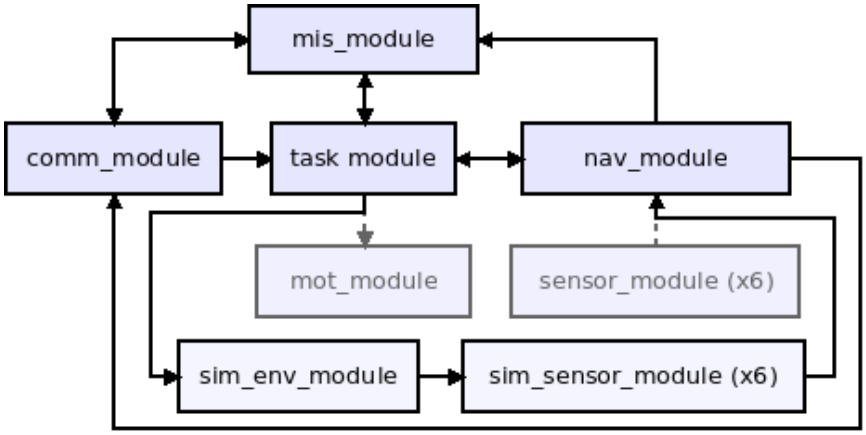


FIGURE 5.3: Implementation of the simulator within the architecture

The sensor drivers are simulated. From the real state of the robot provided by the *sim_env_module*, the 6 *sim_sens_module* generate a simulated measurement for each sensor. To simplify the simulation, we assume the simulated measurements do not have offset errors, contrary to what could possibly occur in reality. However, we consider the measurements have noise following a Gaussian distribution. The standard deviations of the measurement noise depend on the sensors in question and may vary over time.

We further validate that the *sim_env_module* and the *sim_sens_module* do not have a negative impact on the software during the test, since the CPU load of the total process remains below 5% on the simulation PC.

In the next section, we will discuss the theoretical study surrounding the improvement of the coverage of recoveries.

5.2 Calculating an estimation of the recovery coverage

A first estimation of the recovery coverage can be done using theoretical fault tree concepts. The mission, safety and control loop Undesired Events (UEs) need to be addressed by the FTM. Since we do not conduct a quantitative analysis with fault trees, we cannot evaluate a decrease of the probability of top events.

However, it is still possible to observe a decrease of the number of minimum cut sets (MCSs) due to incorporation of Local Recoveries (LRs) after the implementation of the FTM. The first-order MCS can lead to a failure by their mere occurrence, this is then potentially the most dangerous set of undesired events. UEs that do not belong to the first-order minimum cut sets prior to FTM implementation are covered by internal system redundancy.

We define the **local recovery coverage** as the ratio of all UEs covered by at least one LR (i.e. all downlinked UEs to an UE associated with a LR) over the total set of UEs.

Similarly, we define the **decisional recovery coverage** as the ratio of all UEs covered by at least one DR over the total set of UEs. Calculation of both local and decisional recovery coverage are represented in Table 5.1. The columns give respectively the results without the FTM ($\overline{FTM}$) and with the FTM that has been implemented for the tests (FTM).

	$\overline{FTM}$	FTM
Nb of UEs	301	443
Nb of basic events	172	243
Nb of first-order MCSs	88	57
Nb of UEs associated with LRs	0	77
Nb of UEs covered by LRs	0	83
Nb of UEs associated with DRs	0	28
Nb of UEs covered by DRs	0	174
LR coverage	0.0%	18.7%
DR coverage	0.0%	39.3%

TABLE 5.1: A static analysis of the FTM

The results indicate that despite the increasing number of UEs by the implementation of the FTM and the introduction of new faults, the number of first-order MCSs decreases. The LR coverage is 18.7%, which may appear low but we have not implemented all the LRs just as we have not developed all the UEs. This rate is therefore to be put into perspective with a future deepening of this work. The DR coverage is 39.3%, which is obviously higher than the LR coverage, because the UEs associated with DRs are potentially higher in the fault trees.

This static analysis is nevertheless insufficient to fully qualify the effectiveness of the FTM. It is necessary to validate its behavior through simulation and field testing which is addressed in the next sections.

5.3 Unit testing of the FTM functionalities

In this section, we validate the detection, inference and recovery functionalities. We verify that these 3 steps of the fault tolerant process are performed properly:

- First, we validate that a detectable UE which is true is effectively detected as true by the FTM in the next iteration of detection.
- Second, this detection leads to a inference of other UEs according to the cases presented in the section 3.4.2, right after as the detection. We only present the inference from events detected as true, since events detected as false do not impact the system behavior.
- Third, the recoveries are initiated as planned for each configuration of recoveries mentioned in section 3.5 (a LR alone, compatible LRs, partially incompatible LRs and DRs).

These 3 steps are validated on a limited set of UEs and recoveries presented in the following section.

5.3.1 UEs and recoveries of interest

Of all the events and recoveries we may encounter, we choose to focus on a limited representative set for the validation process to ensure that the basic principles operate correctly. We focus on the undesired events listed in both table 5.3 and table 5.4. Distributed in the 3 FTs, the selected UEs are representative of undesired events occurring in a mobile/underwater robot, or have demonstrated a high rate of occurrence during our tests in Mayotte ([65]). We indicate in the "associated recovery" column the recoveries that are launched if the corresponding event is detected or deduced as true. The recoveries of interest are listed in table 5.2.

Through the choice of UEs of interest, we made sure to illustrate several combinations of recovery execution: LR alone, compatible LRs, partially incompatible LRs and DR.

As seen in section 3.5, a LR is launched by the FTM if it is compatible with the current task and if it belongs to the group of compatible LRs with the highest priority. A DR is launched by the mission manager if an UE with an associated DR ("DRi" in table 5.2) is detected or inferred as true and lead to the deduction of a top event.

Recovery	**Description**	**Incompatibilities with LRs**
DR1	Stop the robot	LR8 & LR9
DR2	Surface	
DR3	Redo MP2	
DR4	Go to ship	
DR5	Test the motors	LR8 & LR9
LR1	Change configuration matrix	LR1
LR2	Decrease echo confidence	LR2
LR3	Increase surge speed	LR3
LR4	Decrease surge speed gain	LR4
LR5	Increase sway speed gain	LR5
LR6	Increase heave speed gain	LR6
LR7	Decrease path gain	LR7
LR8	Raise Z target	LR8, LR9
LR9	Apply negative heave command	LR9, LR8
LR10	Restart comm_module	LR10

TABLE 5.2: Recoveries of interest for the validation process

Event	Description	Detection process	Associated Recovery
SAFE_0	Safety constraint violation (Top event)	Bottom-up inference (deduction)	
SAFE_4	Water inside robot	Sponge sensor detecting high level of humidity	DR2
SAFE_6	In critical collision zone with seabed	Estimated altitude compared to a critical threshold	DR2
SAFE_6_1	In warning collision zone 2 with seabed	Estimated altitude compared to a warning threshold	LR9
R_SAFE_6_1	LR8 fails to prevent SAFE_6	Top-down inference (diagnosis)	
SAFE_6_4	In warning collision zone 1 with seabed	Target altitude compared to a warning threshold	LR8
SAFE_1	Collision with obstacle	Acceleration compared to a threshold during a time period	DR1
MIS_0	Mission constraint violation (Top event)	Bottom-up inference (deduction)	
MIS_3_1	In critical distance from TL	Distance to TL compared to a critical threshold during a time period	DR3
MIS_3_1_1	In warning distance from TL	Distance to TL compared to a warning threshold during a time period	LR4 & LR5 & LR6 & LR7
R_MIS_3_1_1	LRs fail to prevent MIS_3_1	Top-down inference (diagnosis)	
MIS_3_9	Localization not precise during MP2	xy variance compared to a threshold during a time period	DR2
MIS_2_1	SP not reached in critical time	Time since the beginning of MP1 compared to a critical threshold	DR4
MIS_2_1_1	SP not reached in warning time	Time since the beginning of MP1 compared to a warning threshold	LR3
R_MIS_2_1_1	LR3 fails to prevent MIS_2_1	Top-down inference (diagnosis)	
MIS_3_8	Filmed volume not concordant	Ratio of volume effectively missed out of total volume to be filmed compared to a threshold	DR3
MIS_2_4	Not enough energy to finish MP1	Battery voltage compared to a threshold proportional to the distance to travel	DR4

TABLE 5.3: Safety and mission related events of interest for the validation process

Event	Description	Detection process	Associated Recovery
ACTd_PROP_0	Actuator generates invalid propulsion (Top event)	Bottom-up inference (deduction)	
ECHOm_ALTI_1_1	Permanent echosounder content failure	ECHOm_ALTI_1_1_1 true during a critical time	
ECHOm_ALTI_1_1_1	Echosounder content failure	Freeze check during a time period, measured altitude noise compared to a threshold during a certain time and residual with EKF compared with a threshold	LR2
R_ECHOm_ALTI_1_1_1	LR2 fails to prevent ECHOm_ALTI_1	Top-down inference (diagnosis)	
ACTd_PROP_1_1_5	The robot is under-actuated	Determinant of AA^T compared to a threshold (A=motor configuration matrix)	DR1
ACTd_PROP_1_1_6	Diagnosis of failed motor impossible	ACTd_PROP_1_1_8 or ACTd_PROP_1_1_9 while not in *test_actuators_task*	DR5
ACTd_PROP_1_1_8	Invalid PWS1 intensity	Difference of modeled and measured intensities during a time period compared to a threshold	
ACTd_PROP_1_1_9	Invalid PWS2 intensity	Difference of modeled and measured intensities during a time period compared to a threshold	
ACTd_PROP_1_1_11	Actuator 2 KO	Difference of modeled and measured intensities during a time period compared to a threshold (during motor test)	LR1
ACTd_PROP_1_1_15	Actuator 6 KO	Difference of modeled and measured intensities during a time period compared to a threshold (during motor test)	LR1
UPBd_PC_2_2_1	Connection with PC is definitively KO	UPBd_PC_2_2_1_1 during a time period	DR2
UPBd_PC_2_2_1_1	Connection with PC is KO	Watchdog of user joystick during a time period	LR10
R_UPBd_PC_2_2_1_1	LR9 fails to prevent UPBd_PC_2_2_1	More than 2 failed re-connection attempts	
ACTd_2_1_2	Not enough energy for actuation	Battery voltage compared to a threshold	DR1

TABLE 5.4: Control loop related events of interest for the validation process

In the next section, we summarize the main results of the unit testing and give some examples. The precise description of the events and recoveries, as well as the details of the unit testing are given in Appendix D.

5.3.2 Results of the unit testing

The synthesis of the unit testing results for mission, safety and control loop related undesired events are represented in Tab 5.5 and Tab 5.6. The tables represent the 15 unit tests (T1 to T15) with the injected faults, the UEs detected as true, the UEs inferred as true, the recovery performed and the verdict of each unit test. We develop the validation of detection, inference and recovery steps and illustrate them with examples.

5.3.2.1 Detection process validation

The injected faults trigger the UE either directly (such as SAFE_4) or indirectly (such as SAFE_6). In the latter case, in order to establish whether the FTM detects the UE in time, it is not relevant to measure the difference between the time of fault injection and the time of UE detection. In order to estimate the adequate time of detection of the events, we refer to the difference between the moment when the UE to be detected is actually present and the moment when the UE is detected, which is noted t_{detect}. For example during a transect, we inject the fault *Force bit of water_inside_enclosure* (test T1) at t=80.000s and the event SAFE_4 is detected at t=80.082s.

After setting the different thresholds of the detection processes, the UEs are detected in line with expectations. Considering all the UEs of interest, we measure systematically $t_{detect} < 100ms$, which is a satisfactory result considering the looping rate of the FTM is 100ms.

5.3.2.2 Inference process validation

We only initiate recoveries associated to the UEs that are estimated as true. Furthermore, we prioritize the inference of UEs estimated as true over UEs estimated as false during conflicts between detection and inference (see section 3.4.2.1). We thus restrict our inference process validation to the UEs inferred as true (following Inference Rules (IR) 1, 3 and 5 introduced in section 3.4.2.1):

- IR1: bottom-up inference (deduction) though OR gates → T1, T2, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14 and T15
- IR3: bottom-up inference (deduction) though AND gates → T12 and T13
- IR7: top-down inference (diagnosis) though AND gates → T6, T8, T11 and T14

The inference proceeds as expected in the same iteration of the FTM as the detection process. We also establish that the propagation time of the inference algorithm in the trees is negligible ($<$ 1 ms).

Example of deduction through OR gates After forcing the battery voltage under a critical threshold at t=50.00s, ACTd_2_1_2 *Voltage of batteries is too low* is detected at t=50.07s (Fig 5.4). The inference algorithm determines then that ACTd_2_1, ACTd_PROP_2 and ACTd_PROP_0 (top event) are inferred as true at t=50.07s.

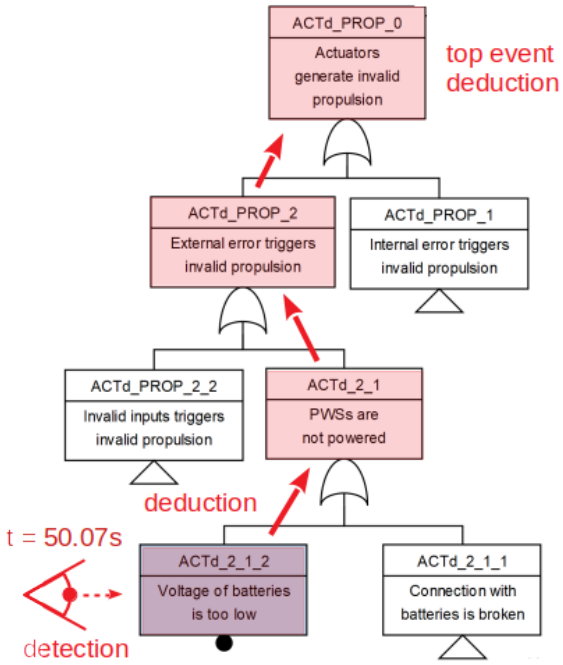


FIGURE 5.4: The inference process during T15

Example of deduction through AND gates During T12, we remove the transmission of the motor2 command. Thus, the motor 2 is actually inoperative whereas the robot believes that the motor2 is operational. During MP0, the Fault Tolerance Manager first detects ACTd_PROP_1_1_8 *Invalid PWS1 intensity* at t=24.55s (step 1 in Fig 5.6). There is no LR handling this UE and no diagnosis can be made as the FTM do not know which motor was inoperative. ACTd_PROP_1_1_6 (*Diagnosis of failed motor impossible*) is then also detected as true. Following the detection of these 2 UEs, a deduction is performed trough AND and OR gates, and the top event of the control loop fault tree is deduced (step 2).

Example of diagnosis through AND gates MIS_2_1_1 (*SP not reached in warning time*) is first detected as true at t=119.3s (Fig 5.5), because we restrain the robot by its cable to prevent it from moving forward. LR3 is then initiated to increase the surge velocity of the robot so that it reaches SP before a critical time. Unfortunately, this time is over during the test and MIS_2_1 (*SP not reached in critical time*) is detected as true at t=137.90s. As a consequence, there is a diagnosis of LR3 failure (R_MIS_2_1_1) which did not prevent MIS_2_1 at t=137.90s (and deductions of MIS_3 and MIS_0). This diagnosis is not frequent because it requires a redundancy (AND gate) whereas in most cases, fault trees are composed of OR gates.

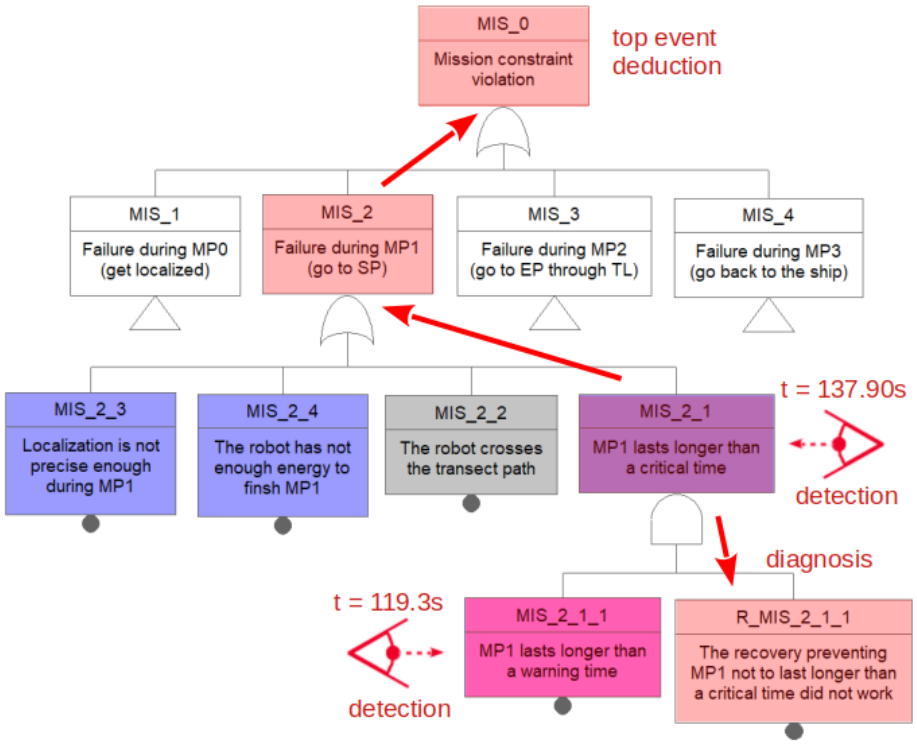


FIGURE 5.5: The deduction process during T8

5.3.2.3 Recoveries process validation

During the 15 unit tests performed, the recoveries are initiated as expected following the detection and inference steps. The recoveries are initiated in less than 100ms after the detection or inference of UEs, which is acceptable considering that the dynamics of our underwater robot is slow. During unit testing, different configurations of recovery are treated:

- a single DR is initiated → T1, T2, T5, T7, T9, T10, T15
- a single LR is initiated → T3, T11
- LR(s) and DR are initiated → T6, T8, T12, T13, T14
- several compatible LRs are executed at the same time → T6
- several incompatible LRs are executed alternately → T4

In some cases, recoveries allow to avoid more catastrophic UEs. In other cases, the injected fault is too severe to be corrected or treated by the recoveries in use. It is also possible that the recoveries are not adapted to the injected fault. For example, during T14 (where we inject the fault: *unplug the user PC from the robot*), LR10 restarts the *comm_module* to reestablish the communication with the surface, which is however impossible to recover in a software way, because the injected fault is material (manual disconnection of the ethernet cable between the robot and the PC).

T12 and T13 are among the most complex test scenarios of recovery. For example, during T12 (where we inject the fault: *prevent motor2 from operating*), both LR and DR are performed (Fig 5.6). Following the deduction of the top event of the control loop fault tree (step 2), the mission manager decides to launch DR5 (step 3). This robotic task allows to make detectable the UEs ACTd_PROP_1_1_10 to ACTd_PROP_1_1_17. Finally, ACTd_PROP_1_1_11 is detected as true, as motor 2 is inoperative (step 4). As a consequence, LR1 is initiated by the FTM to recalculate the configuration matrix without the motor2 (step 5).

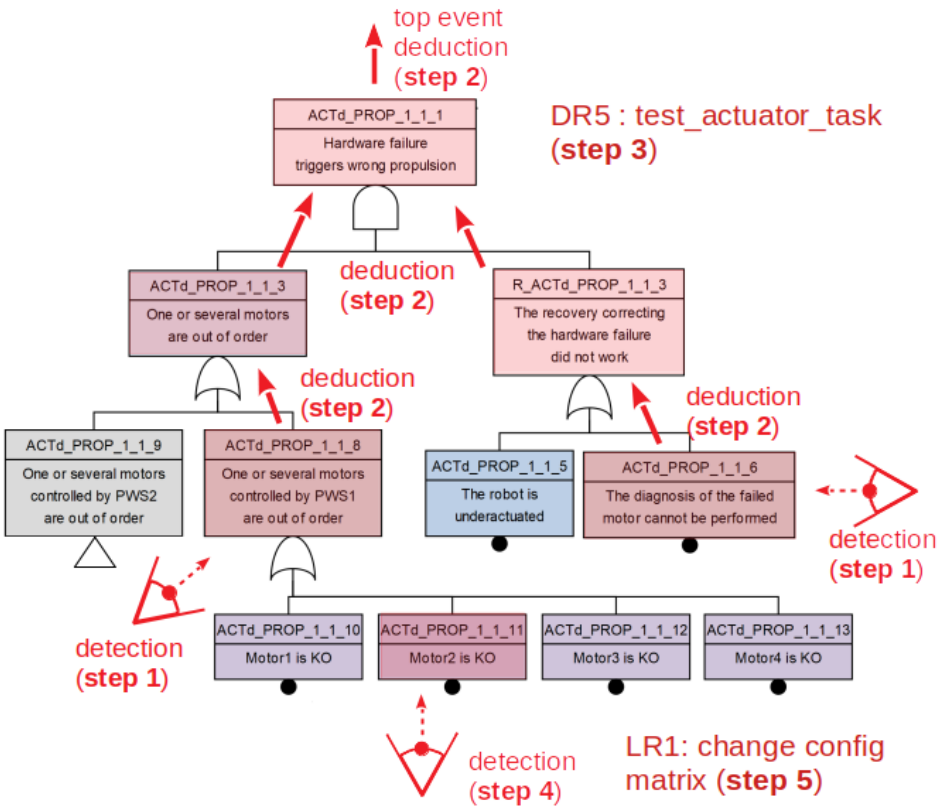


FIGURE 5.6: Detection, inference and recovery during T12

Test ID	Injection description	Detection	Inference	Recoveries	Verdict
T1	Force the water_inside_enclosure bit (TiF)	SAFE_4	SAFE_0	DR2	OK
T2	Perform transect close to the seabed while deactivating LR8 & LR9 (SiL)	SAFE_6	SAFE_0	DR2	OK
T3	Perform transect close to the seabed (TiF) while deactivating LR8	SAFE_6_1		LR9	OK (prevent SAFE_6)
T4	Perform transect close to the seabed (TiF)	SAFE_6_4		LR8	OK (prevent SAFE_6)
		SAFE_6_4 & SAFE_6_1		LR9	
T5	Pull instantly robot cable (TiF)	SAFE_1	SAFE_0	DR1	OK
T6	Simulate strong lateral sea current (SiL) while deactivating MIS_3_8 detection	MIS_3_1_1		LR4, LR5, LR6, LR7	Behavior OK but LRs not sufficient
		MIS_3_1_1 & MIS_3_1	R_MIS_3_1_1 & MIS_0	DR3	
T7	Switch off USBL surface beacon (TiF)	MIS_3_9	MIS_0	DR2	OK
T8	Retain the robot by its cable (TiF)	MIS_2_1_1		LR3	Behavior OK but LR not sufficient
		MIS_2_1_1 & MIS_2_1	R_MIS_2_1_1 & MIS_0	DR4	
T9	Simulate strong lateral sea current (SiL) while deactivating MIS_3_1 detection	MIS_3_8	MIS_0	DR3	Behavior OK but sea current too strong
T10	Set SP very far from robot position (TiF)	MIS_2_4	MIS_0	DR4	OK

TABLE 5.5: Synthesis of the unit testing for safety and mission related events

Test ID	Injection description	Detection	Inference	Recoveries	Verdict
T11	Force echosounder measured altitude to a constant value (TiF)	ECHOm_ALTI_1_1_1		LR2	Behavior OK but LR not sufficient
		ECHOm_ALTI_1_1_1	R_ECHOm_ALTI_1_1_1 & ECHOm_ALTI_0	LR2	
T12	Prevent motor 2 from operating (TiF & SiL)	ACTd_PROP_1_8 & ACTd_PROP_1_1_6	ACTd_PROP_0	DR5	OK
		ACTd_PROP_1_8 & ACTd_PROP_1_11		LR1	
T13	Prevent motor 2 & 6 from operating (TiF)	ACTd_PROP_1_8 & ACTd_PROP_1_1_6	ACTd_PROP_0	DR5	OK
		ACTd_PROP_1_1_8 & ACTd_PROP_1_1_9 & ACTd_PROP_1_1_11 & ACTd_PROP_1_1_15		LR1	
		ACTd_PROP_1_1_11 & ACTd_PROP_1_1_15 & ACTd_PROP_1_1_5	ACTd_PROP_0	DR1	
T14	Unplug the user PC from the robot (TiF)	UPBd_PC_2_2_1_1		LR10	Behavior OK but LR not sufficient
		UPBd_PC_2_2_1_1 & UPBd_PC_2_2_1	R_UPBd_PC_2_2_1_1 & ACTd_PROP_0	LR10 & DR2	
T15	Force battery voltage under threshold (SiL)	ACTd_2_1_2	ACTd_PROP_0	DR1	OK

TABLE 5.6: Synthesis of the unit testing for control loop related events

The unit testing presented in this section allows to validate the behavior of detection, inference and recovery, but it is insufficient to qualify the effectiveness of the proposed FTM. To demonstrate that there is a real improvement in the system's dependability, it is necessary to perform a large number of functional tests. In that perspective, we address the functional testing in the next section.

5.4 Functional and non-regression testing of the system

In this section, we develop the functional testing to validate several properties related to safety and mission. Test cases are generated, tests are performed in simulation (SiL) and logs are produced by the system. We build an oracle to check properties and we present the results. The logs are analysed by the oracle which provides a verdict by analysing the violation rate of the properties.

5.4.1 Test cases: mission and environmental context

We performed three series of functional tests. We performed the first series of transect missions (107 runs) in the field (TiF) in 2019 in Mayotte ([65]) to test a first version of our system without the FTM. This test version is noted $TiF_{\overline{FTM}}$. We logged the data of interest to run an oracle test and check the percentage of violation of mission and safety properties over all the tests. The results revealed that a lot of properties were violated in the first place.

We then intended to replicate the same tests in 2021 with and without the FTM to illustrate the positive FTM contribution on the updated system. Unfortunately, during the beginning of the mission in Mayotte, the DVL and USBL sensors ceased to function, which compromised the optimal progress of the test campaign.

Thus, to validate how the properties are impacted by the FTM, we perform tests only in simulation without and with the FTM. The test cases (mission parameters, environment parameters and injected faults) are summarized in the Tab 5.7. We perform 300 runs without the FTM ($SiL_{\overline{FTM}}$) and 300 runs with the FTM (SiL_{FTM}) to have a representative number of runs while varying the mission parameters, to validate our results on a representative set close to the reality. We simulate a variation of the environmental context and of the characteristic of the robot performance. Among the runs, the mission and the environmental context are not necessarily compatible (e.g. TL can cross the seabed). In this case, there will necessarily be violations of mission or safety constraints. We also inject specific faults into the system, at random times, which may occur during the tests in the field.

Context type	**Parameter**	**Ranges**
Mission	SP and EP xy position	random but satisfying $\delta SP_{init} < 50m$, $\delta EP_{init} < 50m$, $TL > 30m$
Mission	Transect type	50% of chance that TL is at constant depth 50% of chance that TL is at constant altitude
Mission	SP and EP depth	random between $WC_{init} - 2m$ and $WC_{init} - 5m$ (same for SP and EP)
Mission	SP and EP altitude	random between 1m and 3m (same for SP and EP)
Mission	u_{cruise}^{MP1} and u_{cruise}^{MP3}	random between 0.3m/s and 0.7m/s
Mission	u_{cruise}^{MP2}	random between 0.2m/s and 0.5m/s
Environment	Seabed	random between 5m and 50m ($slope < 58$ deg)
Environment	Sea current	random between 0m/s and 0.3m/s in any direction, evolving every 30s
Environment	Sensor noise	random standard deviations depending on the sensor and evolving in time
Environment	Actuator effectiveness	random between 80% and 100% for each actuator (fixed in time)
Environment	External cable force	from 0N to 20N depending on the robot speed and pose
Injected fault	Absence of USBL signal	20% of chance every 4s (period of reception). If not received in the previous iteration, 67% of chance in the current iteration
Injected fault	User PC disconnection	0.67% of chance every 12s
Injected fault	Sensor module crash	0.67% of chance every 12s for each sensor module

TABLE 5.7: Synthesis of the functional test cases

In the next section, we present the results of the functional and non-regression testing performed on the basis of the test cases.

5.4.2 Results of the functional and non-regression testing

The results of the functional and non-regression testing reveal that:

- there is a significant decrease in the property violation rates, particularly for safety properties. The system's estimate of the average violation rate of the properties decreases by 4.1%. In reality (with ground truth), the average violation rate of the properties decreases by 10.2%.
- the FTM is active during the runs as a large number of UEs are detected or inferred. 4.9 LRs are initiated per run and 0.5 DRs are initiated per run during the tests.
- without the FTM, a run lasts approximately 6.65 minutes while it lasts 6.18 minutes with the FTM. The difference is caused by the FTM interrupting the mission by a DR if a significant event is detected, knowing that the robot is endangered or that the transect data will be of little interest for biologists.
- the FTM does not have a significant negative impact on the system in terms of CPU load and module timing constraints since the CPU load remains inferior to 5% in the UP-board controller.

We will detail the results in the next sections.

5.4.2.1 Evaluation of the violation rate of properties

The results of our functional tests are presented in Tab 5.8 and the property violations in simulation (SiL) are represented graphically Fig 5.7. The mission and safety properties mentioned in Tab 5.8 have been defined in 2019. These properties may be different from the properties defined in the chapter 4, especially in regard to the thresholds.

We have posted the property violation rate obtained from the tests in the field in 2019 for informative purposes. Since then, the robotic system without the Fault Tolerance Manager has evolved, as additional sensors were introduced (GPS, DVL). It is thus impossible to validate the efficiency of the FTM in the system from these data, especially since the recent functional tests are performed in simulation (SiL).

We distinguish P1 to P8 as the mission properties and P9 to P11 as the safety properties. For each run, a property is evaluated if the conditions necessary for its evaluation are met. For example, P3 (*Distance to TL is inferior to 3m*) is not evaluated if the robot does not start MP2. Thus, P1, P9, P10 and P11 are always evaluated and P2 is evaluated if the robot has performed MP0. P3, P4, P5, P6 are evaluated if the robot has performed MP1. P7 and P8 are evaluated is the robot has performed MP2.

For the tests performed in simulation, we indicate if the properties have been violated with respect to the estimate made by the system or with respect to the ground truth (provided by the simulator). This process cannot be done in the field, since we do not possess accurate external sensors. We consider that the ground truth and the estimate are in accordance for P1, P8, P10 and P11.

Id	Property	Property violation rate				
		$TiF_{\overline{FTM}}$ (2019)	$SiL_{\overline{FTM}}$ (2021)		SiL_{FTM} (2021)	
		Estimate	Estimate	Ground truth	Estimate	Ground truth
P1	USBL signal is received during MP0	8.3%	0.7%		1.3%	
P2	SP is reached	51.5%	12.3%	29.3%	11.0%	12.0%
P3	Distance to TL is inferior to $3m$	72.9%	11.4%	21.3%	12.0%	11.6%
P4	Speed is inferior to $0.75m/s$	27.1%	4.9%	1.9%	3.7%	3.7%
P5	Angular speed is inferior to $22.5/s$	81.2%	24.0%	18.6%	23.2%	12.0%
P6	EP is reached	25.0%	15.2%	52.5%	16.9%	40.4%
P7	Coverage rate of the filmed volume is superior to 70%	44.4%	0.0%	12.6%	0.0%	0.0%
P8	EP is reached in time with a minimal speed of $0.1m/s$	13.9%	2.2%		1.4%	
P9	Estimated altitude is superior to 0.75m	3.7%	24.3%	24.7%	16.3%	5.0%
P10	A signal is received from user PC for less than 3s	37.0%	16.3%		0.3%	
P11	The variance over xy is inferior to $16m^2$	?	20.0%		0.0%	

TABLE 5.8: Property violation rates over all runs

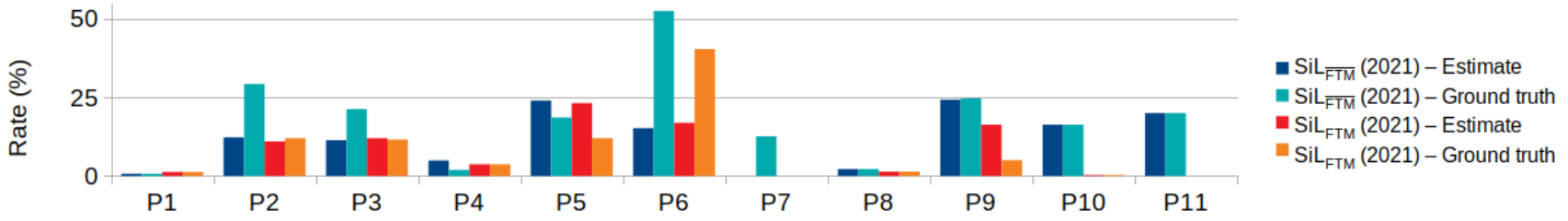


FIGURE 5.7: Rate of property violation with and without the FTM

We first notice that the introduction of the FTM provides a closer match between the results of the ground truth and the estimate by the system. Indeed, for P2, P3, P4, P5, P6, P7, P9, there is 12.2% of mean difference between data without the FTM and only 6.8% of mean difference between data with the FTM. This is a result of the individual sensor modules being restarted by the FTM when they fail, which improves the accuracy of the localization. As a result, P11 is violated less often with the FTM. The system's estimate of the average violation rate of the properties decreases by 4.1% and the ground truth of the average violation rate of the properties decreases by 10.2% with the FTM.

We then notice that the FTM has no particular effect on the USBL signal reception during MP0 (P1), on the respect to the maximal speed during MP2 (P4) or on reaching EP in time (P8) (average ground truth difference of 1.1%). We believe that the recoveries performed, which normally help the system to respect these properties, are not particularly effective in the testing context. However, with the FTM, the other mission property violation rates tend to decrease, especially with regard to the ground truth (average ground truth difference of 11.7% for the other mission properties). This is related to the association of the enhanced localization of the robot and the local recoveries implemented.

We also notice that the properties related to safety are significantly better met with the FTM (average ground truth difference of 18.6%). P9 is better respected with the FTM, due to the effectiveness of the LRs that prevent the robot from getting too close to the seabed. It should be noted that P9 is violated in 5.0% of the cases, which occurs in particular when the seabed rises too sharply. Finally, P10 is better respected because the LR restarts the communication module when it crashes which prevents P10.

In the next section, we will examine the undesired events, LRs and DRs occurring during the functional testing.

5.4.2.2 Occurrence of undesired events, local recoveries and decisional recoveries

During the functional testing, the FTM has allowed:

- to detect or infer 70.3 UEs per run, (683 per hour). We count an UE as detected or inferred when there is a rising edge of its estimate by the FTM, at least 1 second after the falling edge of this UE. Approximately 30% of the events are related to excessive acceleration during MP2 (MIS_3_4), which may scare the fishes during the transect. However, we believe that this event is not common in the real environment, because it arises mainly from inadequate sensor measurement noises during the simulation combined with the overestimation of the real acceleration by the simulated environment. For this reason, we deactivate the association of the DR associated to MIS_3_4 before the runs in simulation. Around 15% of the events are related to the excessive surge speed during MP2 (MIS_3_3_1_1), mainly on transect where the desired surge speed is low (0.2m/s). It is indeed easy to exceed the speed limit, which is set at 1.5 times the desired speed. The other detected events represent a proportion lower than 5% each.

- to launch 4.9 LRs per runs (48 per hour). The main LRs initiated (around 20% of the LRs) are *change of u speed gain* (LR4) to react to the excessive surge speed (MIS_3_3_1_1), *restart usbl_module* to react to the usbl halt output and *change heave force* (LR9) to react to the proximity with the seabed. *Change of z target* (LR8) and *restart echo_module* are also launched frequently (around 8% of the LRs).
- to launch 0.54 DRs per runs (5.2 per hour). The main DR launched are to stop the motion (DR1), to surface (DR2) or to perform a motor test (DR5).

The local and decisional recoveries are various and allow, directly or indirectly, to limit the constraints violations as illustrated in the previous section. The DRs also prevent unnecessary mission continuation when a severe event occurs. Thus, the average time of a transect decreases from 6.65 minutes without the FTM to 6.18 minutes with the FTM.

In the following section, we detail the non-regression aspects of functional testing.

5.4.2.3 Non-regression testing of the system

We have observed that the introduction of the FTM does not worsen the violation of the properties established in section 5.4.2.1 in simulation. However, during the tests in the field, overheads may also be present as a result of inappropriate reactions of the FTM, e.g. if the FTM detects UEs that are not actually present (false positive UEs). The FTM may also induce a heavy computational cost that may disrupt the system.

Thus, we decide to conduct 2 series of tests in the field while the robot was performing the same transect, with a desired depth of 3m and a TL length of 25m. During the first series of tests, we perform 2 runs without the Fault Tolerance Manager, that we consider as the golden runs. During the second series of tests, we perform 2 runs with the Fault Tolerance Manager. We do not inject faults during the tests.

In the next paragraphs, we will develop that reactions may disrupt the system slightly or totally in certain cases, mainly due to false detections of undesired events, incorrectly set thresholds or unexpected context. However, the FTM does not have a significant negative impact on the rest of the system with respect to execution time.

False positive undesired events and inappropriate reactions Even if we did not inject faults during the non-regression test, several UEs were detected during the runs with the FTM.

For example, the communication with the user PC was interrupted for a few seconds (UPBd_PC_2_2_1_1) during a loading of the mission, so the FTM detected a communication break and restarted the *comm_module* to re-establish the connection (LR10). This reaction was not necessary in this context, because the absence of signal was due to the large amount of information to be sent between the surface PC and the robot during a limited period. However, the reaction of the FTM did not significantly alter the operation of the system, since the module restarted after the mission was loaded.

Similarly, the USBL was not detected during a warning time (event USBLm_XYusbl_1_3_1). As a consequence, the FTM restarted the module several

times before the robot picked up the USBL signal again. This reaction was not necessary in this context, because the lack of USBL signal was most often due to a bad environmental context (transient fault). The signal was recovered automatically when the robot get closer to the vertical axis of the USBL surface beacon.

More importantly, the water leak sensor detected the presence of water inside the enclosure. As a consequence, the FTM detected SAFE_4 (Fig 5.8). The two possible immediate causes of this undesired events are *There is a leak on the enclosure* or *Water is present since the enclosure was closed*. We checked there was no leak on the enclosure and we concluded that SAFE_4 was caused by condensation inside the rear enclosure that humidified the sensor and triggered the false signal.

When this SAFE_4 was detected, the mission manager immediately stopped the transect, which could not even be started. Knowing there was no water that can potentially damage the electronics inside the enclosure, we bypassed the water leak sensor.

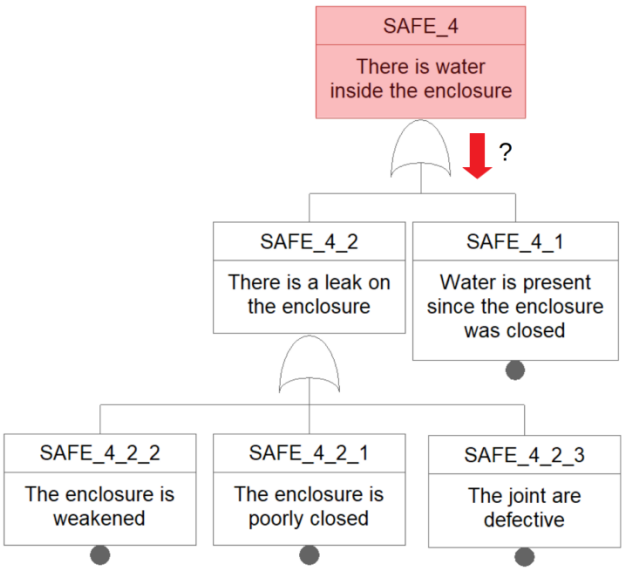


FIGURE 5.8: Extract of the safety fault tree around SAFE_4

Respect to the module timing constraints We compare software characteristics such as the mean CPU load and the mean module latency to observe the impact of the FTM on the system between the 2 series of tests. The results, detailed in table 5.9, reveal that the FTM increases the overall CPU activity by 10% of the initial activity (4.5% $\rightarrow$ 4.9%). With an average CPU load of 4.9%, the impact of the FTM on the CPU load is insignificant for the system.

Moreover, the FTM does not induce latency on the modules since the average latency for each module is very similar with and without the FTM. With $RMMP(i)$ the Real Mean Module Period and $DMP(i)$ the Dedired Module Period of the module i, we simply calculate the Mean Module Latency (MML) of the N modules as:

$$MML = \frac{1}{N}\sum_{i=1}^{N}(RMMP(i) - DMP(i)) \tag{5.2}$$

	$TiF_{\overline{FTM}}$	TiF_{FTM}
Number of modules	10	11
Mean total CPU used (%)	4.5	4.9
Mean module latency (μs)	537	521

TABLE 5.9: Software activity with and without FTM

We conclude that the Fault Tolerance Manager does not have a significant impact on the system in terms of CPU load or module latency.

5.5 Conclusion and discussion

5.5.1 Respect to the test objectives

In this chapter, we experimentally validate the framework described chapter 3, on our experimental robotic platform described chapter 4. We first review the tests objectives, the system under test, and describe the context of the tests in the field and in simulation.

There is an improvement from a theoretical perspective since the Fault Tolerance Manager covers many undesired events locally, which decreases the first-order minimal cut sets in the fault trees built. We have calculated that with our current implementation, 18.7% of the undesired events are covered by at least one local recovery. Decisional recoveries can also be initiated if there is no local solution available, further reducing the probability of a catastrophic event. We have calculated that with our current implementation, 39.3% of the undesired events are covered by at least one decisional recovery.

The unit testing of the FTM validates the detection and inference of several undesired events related to safety, mission and control loop aspects. The detection and inference of one or more events is performed as planned, and indicated in the chapter 3. Compatible local recoveries are performed together and incompatible local recoveries are initiated based on their priority. Decisional recoveries are initiated when the FTM informs the mission manager of the presence of a top event and the associated reaction prevents a more catastrophic event from occurring.

We extend the functional testing of the FTM to the system, to validate its behavior on a large number of tests. We perform more than 300 transects with and without the FTM in simulation and we observe that the FTM decreases the violation rate of mission and safety properties by 10.2% on average (from 18.2% to 8.0%). The average time of a transect is reduced by 7% with the FTM (6.18 minutes with the FTM instead of 6.65 minutes without it). Thus, more missions are performed for the same total duration.

Non-regression testing demonstrates that the FTM has no negative impact on the system as the CPU load remains low on the UPboard controller. Furthermore, the FTM does not induce latency on the system modules. In addition, recoveries initiated by the FTM do not disrupt the system, except for failed and non-redundant sensors indicating false positive catastrophic events.

We can conclude that our test objectives are validated and therefore our fault tolerance framework improves dependability by managing undesired events at multiple levels. However, limitations and threats to validity are present, which we will address in the next section.

5.5.2 Limitations and threats to validity

The question that drives this section is: did we perform the tests correctly and do they really validate the effectiveness of our FTM?

The literature mentions two types of validity for a procedure ([151]). Internal validity aims to reinforce confidence in the rigor of the test to control every aspect of it. External validity establishes confidence in the generalization of the procedure to other contexts. In our validation process, both aspects have been addressed. Section 5.3 allowed to validate the functioning of the detection, inference and recoveries of the undesired events whereas section 5.4 allowed to validate the behavior of the system on a large number of tests.

However, we believe it is necessary to discuss several points: the thoroughness of the tests, the relevance of the injected faults and the context, the reaction to unexpected events and the consideration of the ground truth in the validation.

5.5.2.1 Thoroughness of the tests

The number of tests to be performed is at the discretion of the tester. In our case, we performed 15 runs in the field or in simulation dedicated to unit testing, 300x2 runs in simulation dedicated to functional tests and 2x2 runs in the field dedicated to non-regression tests. We believe that these numbers are sufficient to give a representative indication of the performance of the fault tolerance framework.

We performed the tests on our proposed framework implemented in the architecture of an underwater robot. However, we have implemented only a subset amount of the detections and recoveries, as it would require a substantial amount of time and resources. It would be appropriate to continue the implementation of the framework to perform a more complete validation of the framework combined with the system. We also believe that it would make sense to extend the properties of the functional testing to have a more complete representation of the performance of the system associated with the fault tolerance framework.

5.5.2.2 Relevance of the injected faults

We inject faults that we consider to be representative of the real faults occurring in the robot. We estimate these faults are likely to occur. However, we have not tested the system sufficiently in real conditions to trace all significant faults and, more broadly, the undesired events, as this would be more time consuming. We believe that having modeled real-world aspects in the simulator such as cable strength or marine current, enhances the validity of the test. However, a test in simulation does not have the same value as a test in the field, because the controlled environment cannot trigger all faults in the simulated system.

In addition, the way some faults are injected does not necessarily cause the same system reaction if the faults are really present. For example, let's consider the injected fault of T1 (*Force the water_inside_enclosure bit*). In this example, if the fault

were real, it could initially damage the robot controller before the sensor detects the presence of water. This scenario must therefore be considered at the design stage of the system to place the water detectors at strategic locations to give the system the time to react.

We have also observed that the injected faults (or the real environmental context) are sometimes too severe and that the system cannot fight against the resulting undesired events with LRs. For example, the strong lateral constant sea current simulated during the unit test T6, or maintaining the robot by the cable during the unit test T8, could only be countered by reactions that exceed the physical capabilities of the robot. It is therefore important to identify the mission contexts in which the system associated with the Fault Tolerance Manager can and cannot react.

5.5.2.3 Reaction to unplanned undesired events

Even a well-designed framework cannot cover all undesired events, because it is impossible to predict all faults, especially in an open environment where the context is not fully known. Thus, we wonder: How our framework reacts to undesired events that are not expressed in the fault trees (UEs not in $UE_{express}$)?

We believe that there are two main scenarios. In the first case, the unknown undesired event can be assimilated to another undesired event. For example, if a giant octopus comes too close to the robot from below, the UE *In critical zone with the seabed* will most likely be detected. Thus the decisional recovery will bring the robot to the surface, which may seem appropriate in this context.

In a second case, the unknown event is not assimilated to another event. It is not detected and therefore does not cause any reaction from the robot. In this case, one can only hope that the criticality of the undesired event in question remains very low, i.e. it is very unlikely to occur and that it does not cause significant damage.

5.5.2.4 Ground truth vs estimate in undesired event detection

In the light of the functional testing conducted in simulation, it appears that the difference between the ground truth and the robot's estimate can vary widely. More generally, the low cost of the robot hardware allows only a small number of sensors with relatively low accuracy, which can cause bad estimates, and false detections or non-detections of UEs.

For example, during tests in the field in the Mediterranean sea in June 2021, the echosounder occasionally indicated erroneous readings, which puts the robot in dangerous situation when it was close to the seabed. Indeed, the sensor indicated that the robot was far from the seabed, which prevented the recoveries from moving the robot away from the seabed.

To verify the accuracy of the localization in our validation campaign (Mayotte 2019), we materialized the transect line with a rope placed on the seabed ([64]). By filming the robot with external cameras, we checked the relative position of the robot on the rope and compared it to the estimation by the system. We concluded that there were also large differences between the two positions. The issue of ground truth remains open, especially in underwater robotics where the logistical implementation of traditional solutions to obtain ground truth (instrumentation of the environment or use of precise sensors external to the control loop) is difficult and very expensive.

In conclusion, the presence of a fault tolerant software framework alone is not enough to guarantee the dependability of an autonomous system. There is always a trade-off between resources, money, skills, infrastructure, time and reliability. In the next and final chapter, we conclude on our work and provide elements of improvement for the continuation of our research.

Chapter 6

Conclusion

Contents

6.1 Background and problematic 121
6.2 Synthesis of the main results 121
6.3 Limitations 124
6.4 Improvement 125
6.5 Future works 126

Our study focused on the problems raised by the consideration of the dependability of low cost autonomous mobile robots. After reviewing the context of our work, we will first summarize the main results that constitute the important and original points of our work. Secondly, we will discuss its limitations and we will examine the works that could complete the present results. Finally, we will draw some lessons about the contributions of this study and discuss the interest that the proposed framework could have for other robotic missions carried out in autonomy.

6.1 Background and problematic

Mobile autonomous systems can perform a variety of missions, ranging from exploration to interaction with the environment. Mobile systems performing autonomous missions have a requirement for reliability to accomplish their missions while preserving their integrity and the integrity of the environment. The work presented in this manuscript aims to enhance the dependability of autonomous mobile systems.

The dependability is usually the result of a compromise with the involved costs. Nevertheless, it appears that in some situations, a better dependability would have generated lower costs than the consequences of the system accident. We therefore raise the question of how to improve the dependability of an autonomous mobile robot, while minimizing the final cost of its development.

In next section, we will summarize the main results reported in this manuscript.

6.2 Synthesis of the main results

Faults and failures can occur in mobile systems, during the development phase or the operational phase. Many works try to improve the dependability of these systems through various methods. Fault prevention, through the different methods

of the engineer, allow to reduce the introduction of faults. Fault forecasting allows to estimate the number of faults and their consequences through the use of risk analysis or logic models that link faults and failures, such as fault trees. We also noted that through the various forms of testing, fault elimination can reduce the number of faults and their severity. Unfortunately, these solutions are not sufficient to prevent faults from occurring in real time. It is therefore necessary to proceed with fault tolerance to prevent failures despite the presence of faults.

We have presented the FDIR (Fault Detection Isolation and Recovery) approach for fault tolerance based on the use of redundancy. The work around fault tolerance is effective in responding to specific and identified errors, but it only deals with a limited set of errors at once. Context-related undesired events, i.e. violations of mission or safety constraints, and control loop-related undesired events, i.e. faults, errors and failures in the control loop of the system, are often treated separately and differently. In addition, since the methods do not necessarily handle multilevel reactions, the lack of compatibility analysis between recovery mechanisms at different levels may have consequences on the robot's safety and the performance of the mission. The dependability is therefore not granted since there is no generic method to manage the different undesired events at multiple abstraction levels. That is the issue we address in this work.

In order to address the shortcomings presented in the state of the art, we have developed an original and versatile multilevel fault tolerance framework, composed of a Fault Tolerance Manager and a classic hybrid control loop architecture. This framework is derived from the basic principles of fault tolerance methods and aims to react to undesired events at several levels to prevent more serious undesired events from occurring. Through the Fault Tolerance Manager and the mission manager, the system can react to the possible causes and consequences of the mission and security constraints violations, established after a thorough study of the specifications, the mission context and the system. Meanwhile, it can also react to undesired events related to failures in the control loop of the system.

To develop this fault tolerance framework, in the design phase, fault trees are first built from undesired events such as mission or safety constraint violations, or invalid actuator propulsion. We propose a guideline to build more easily these fault trees. The causes of these undesired events are described and logically linked in the fault trees until the initial faults are obtained. The weaknesses of the system are then analyzed by an expert, according to the present redundancy and the criticality of the undesired events. Finally, local and decisional recoveries are implemented to improve the reliability of the system.

In the running phase, a Fault Tolerance Manager scans the fault trees in real time and evaluates a predefined set of undesired events. An inference step is then performed by following the logic of the fault trees to diagnose the logic causes and deduce the logic consequences. Finally, if undesired events are detected or inferred as true, local recoveries or decisional recoveries may be initiated during the recovery step. Local recoveries consist in manipulating the local resources at disposal without modifying the robotic task to limit or eliminate an undesired event. Decisional recoveries are task changes decided by the mission manager when a top event is deduced, thus reflecting the absence or failure of potential redundant processes.

We have implemented the proposed fault tolerance framework on a low-cost underwater robot. This underwater robot is a hybrid remotely operated vehicle capable

of performing autonomous missions or being remotely operated by a user in all directions. The underwater robot has been jointly developed between the LIRMM and marine biologists from Marbec and CUFR of Mayotte. This work is more broadly included in the Bubot project ([23]), which aims to provide tools for the observation of the marine species. The mission is an underwater biological protocol called transect, which consists in following a straight line close to the seabed to film the underwater species. The specifications are also precisely defined, as well as the different constraints that the robot must fulfill during its mission.

We also specify the implementation of the 3 fault trees describing the undesired events related to the mission, safety and control loop. The mission fault tree is built from the mission constraint violations and divided according to the mission phases. The safety fault tree is established from the safety constraint violations. Finally, the control loop fault tree is built from the failure of the propulsion and using a modular representation of the control loop. This dependability approach according to these different points of view is new. It allows to guide the designers during the fault tree construction process.

We have tested the effectiveness of the proposed framework associated with the system. First, we compute the coverage of the recoveries. This coverage is theoretically estimated by calculating, from the fault trees, the coverage of the undesired events by the local and decisional recoveries. We calculate that for our implementation, these coverages are 18.7% for the local recoveries and 39.3% for the decisional recoveries. These rates are relatively low and therefore do not treat all undesired events, but it should be considered that the implementation of our framework is a first draft and proof of concept.

Second, we have tested our framework in simulation and in the field in a lake. It appears that the detection, inference and recoveries are well performed according to expectations and on time. However, depending on the test cases, some undesired events that should be prevented may still occur. This may call into question some local recoveries that would not be efficient enough, at least for the context of the tests.

Third, a functional testing was performed in simulation with 300 runs to observe the consequences of the introduction of our framework on the violation of mission or safety constraints. This testing has revealed that the introduction of the Fault Tolerance Manager causes a reduction of mission constraint violations, mainly because of the enhancement of the robot's localization, and a reduction of safety constraint violations. Indeed, many undesired events were detected during these tests and many recoveries were also initiated. Furthermore, it appears that the introduction of the fault tolerance framework did not disrupt the system, besides false detections due to non-redundant and faulty sensors. The framework also reduces the average mission time, as the mission is aborted if a critical undesired event is detected by the framework, which prevents the mission from continuing unnecessarily.

This effectiveness analysis allows to conclude that the implementation of the fault tolerance framework we propose is effective in preventing most preventable critical undesired events. Nevertheless, it also suffers from several main limitations that we will address in the next section.

6.3 Limitations

This thesis work is a proof of concept of a fault-tolerance generic framework that detects undesired events and initiates recoveries to limit or correct their consequences. Immediate directions for its improvement are to consider an increasing number of events and recoveries to enhance the dependability of the system. However, there is a trade-off between the cost of developing an autonomous robotic system and its dependability, which intervenes during the development phases of the system until the testing of the system. The development cost of a system can be temporal, financial or logistical. This trade-off usually consists in agreeing on the allocation of the different resources to achieve the expected dependability results. It must be clearly clarified at the beginning and during any project with the stakeholders.

During our last field test campaign in Mayotte, we lost two major sensors (DVL and USBL). These two expensive sensors were not redundant enough with the other sensors, so we could not fully perform our test campaign. The faults leading to the failure of the sensors were respectively overheating and leakage. We had no means set up with our fault tolerance framework to detect and react in order to prevent these failures.

In the absence of sufficient redundancy, it may be impossible to guarantee that the robot will function properly given a harsh and unknown context. Hence, it is necessary to minimize the probability of occurrence of plausible catastrophic events. It is therefore important to ensure a given level of hardware and software functions redundancy.

For any mobile robot, and even more widely for any autonomous system, it is necessary to rely on an expert in charge of the dependability of the system. Our framework is not an exception to this rule. This dependence on the expert may become a limitation to dependability, in case of misinterpretation or shortcomings from the latter.

An expert is first needed to identify constraint violations that could affect the mission or safety of the system. He must also design the fault trees, which can be relatively complex depending on the system. The expert must then analyze the weaknesses of the system and decide to which events the fault tolerance manager must react. He decides the implementation of local recoveries and the deployment of the detection mechanisms in the fault trees, depending on the available mechanisms, the criticality of the associated events, and the recovery coverage to attain. Then, to manage potential conflicts between local recoveries, the expert must prioritize recoveries associated with more critical events. Finally, the expert must establish the trade-off between priorities of local recoveries and their number in case of incompatibilities (α and β parameters in section 3.5).

We consider that the undesirable events are represented under 3 categories that are mission, safety and control loop. Mission-related undesired events directly affect system reliability and availability attributes. Safety-related undesired events directly affect system safety and integrity attributes. Control loop-related events indirectly affect all attributes. The 3 fault trees built led us to identify undesirable events related to these categories. However, the formalism of the fault trees to deduce consequences and diagnose faults has limitations, which would probably not facilitate the integration to more complex systems.

For a classical use of fault trees and to obtain the probability of occurrence of the top events, a probabilistic analysis is performed based on the probabilities of occurrence of basic events. However, a probabilistic analysis of each event is almost impossible. This would require exhaustive testing under real conditions for undesired events related to a component failure. This analysis is also complex for undesired events related to a constraint violation, as they are often context dependent which is difficult to represent in fault trees. Despite that fault trees are a graphical tool easy to interpret, the binary logic between events also makes them more difficult to integrate and to use them into a real time system. Thus, as seen in this manuscript, it is not possible to link the mission and safety fault trees with the control loop fault tree due to timing and context issues.

In the following section, we discuss possible improvements to be made based on our work described in this manuscript.

6.4 Improvement

We intend to further test our framework by conducting more tests in the field. This point is especially relevant as the functional tests carried out in this thesis were mostly preformed out in simulation. While work is underway to improve the simulation to facilitate debugging, it is necessary to validate the system in the field under real conditions, experimenting with varying weather conditions and types of environments to maximize the variety of adverse events encountered. We also plan to vary the mission and the system, for example by applying our framework on different underwater robots. All these variations would strengthen the external validity of our framework, generalize the validation and prove its versatility.

We could also consider placing the Fault Tolerance Manager on a separate controller from the primary controller, to improve the dependability of the robot. Indeed, in case of failure of the primary controller, the Fault Tolerance Manager would still be able to react to undesired events. However, this requires a major redesign of our implementation, as this would imply to build an architecture that can be fully decentralized in the secondary controller.

We can also plan an improvement of the real-time operation of the Fault Tolerance Manager. In this manuscript, we describe the Fault Tolerance Manager as a classical module executed synchronously with a period of 100 ms. Currently, for an iteration of the Fault Tolerance Manager, inference is based on a fixed set of detected undesired events, which makes it easier to implement and probably requires less computing resources.

An asynchronous implementation of the Fault Tolerance Manager would allow to be more reactive, because the detection, inference and recovery processes would be triggered instantaneously after the occurrence of an undesirable event. Asynchronism can be achieved by using dedicated middleware, such as the real-time framework ContrACT developed in the LIRMM ([126]).

In addition to extend the fault trees by identifying the causes of undeveloped undesired events, it also appears necessary to increase the means and efficiency of the detection mechanisms and the local recoveries.

Improving the means of event detection can be achieved by adding hardware to cover more events, or by refining the detection algorithms to ensure they identify the occurrence or non-occurrence of the events as closely as possible. Obstacle detection is for example an important area of improvement for our system. It would be interesting to combine obstacle detection by acoustic methods and camera image methods (as in [37]).

As stated in the previous chapter, the Fault Tolerance Manager is not always efficient to recover from undesired events, especially in a particular context of occurrence of these events. We believe that some recoveries need to be refined to best eliminate or prevent adverse events. In the example of obstacle avoidance, studies in underwater robotics allow to avoid obstacles more efficiently with dedicated algorithms ([135],[48]).

Finally, work is currently underway to improve the mission manager to support more complex missions ([166]). In this manuscript, we have considered the mission manager as a simple black box specifically reacting to a set of undesired events when a top event was deduced as true. A more sophisticated mission manager would then be able to more effectively manage conflicts between different reactions based on rules established by the user.

In the next and last section, we will focus more broadly on future works likely to use our work.

6.5 Future works

In our case study, we optimally consider that all resources should be used to maximize localization and motion performance. Current work uses a hardware and software resource manager such as PANORAMA ([88]) to respect safety, mission duration, location or energy constraints. It would be interesting to combine it with our work. This would result in a redesign of the fault trees which would become dynamic to avoid considering the components as operational that are not needed at a given time. This dynamic modification of the trees during the mission would thus be based on the resources needed to perform the robotic tasks.

Mobile robotics projects and applications are numerous: marine resource observation, environmental exploration, structural inspections, underwater operations... These projects require efficient and reliable methods to avoid compromising the integrity of the system and impacting the project cost.

Karst is a landscape formed by sinkholes, caves and underground drainage systems where rainwater can be stored ([31]). Current projects led by the LIRMM aim to use autonomous underwater robots to explore these inaccessible areas full of plate water resource. In this particular harsh context, the localization of the robot and its integrity are essential to avoid generating additional costs. As the robot is expected to evolve into a fully autonomous system without user supervision, our fault tolerance framework would then provide a solid basis to perform this type of mission as safely as possible.

It is clear that to achieve full autonomy for long-duration robotic missions, the development of new efficient approaches for fault tolerance could be a key issue in the next decades.

Appendix A

Representation of the fault trees

In this appendix, we represent the fault trees that have been implemented during this thesis.

		Detectable	With associated LR	With associated DR
	UE			
	UE	X		
	UE			X
	UE	X		X
	UE		X	
	UE	X	X	

R_XX Failure of the LR associated to undesired event XX

i Sub-fault tree developed Fig A.i

j Fault tree linked to fault tree developped Fig A.j

Legend of the fault trees

The fault trees are displayed on 10 pages. Fig A.6 represents the mission fault tree and Fig A.8 represents the safety fault tree. Fig A.5 represents the concatenation of the 3 FTs for simplification purpose. The remaining figures represent sub-fault trees of the control loop fault tree.

We have developed the sub-fault trees related to a failure of the altitude output by the *dvl_module* (Fig A.3) and *echo_module* (Fig A.4). We do not represent the sub-trees related to the other driver outputs which follow the same logic, also presented in section 4.4.3.4.

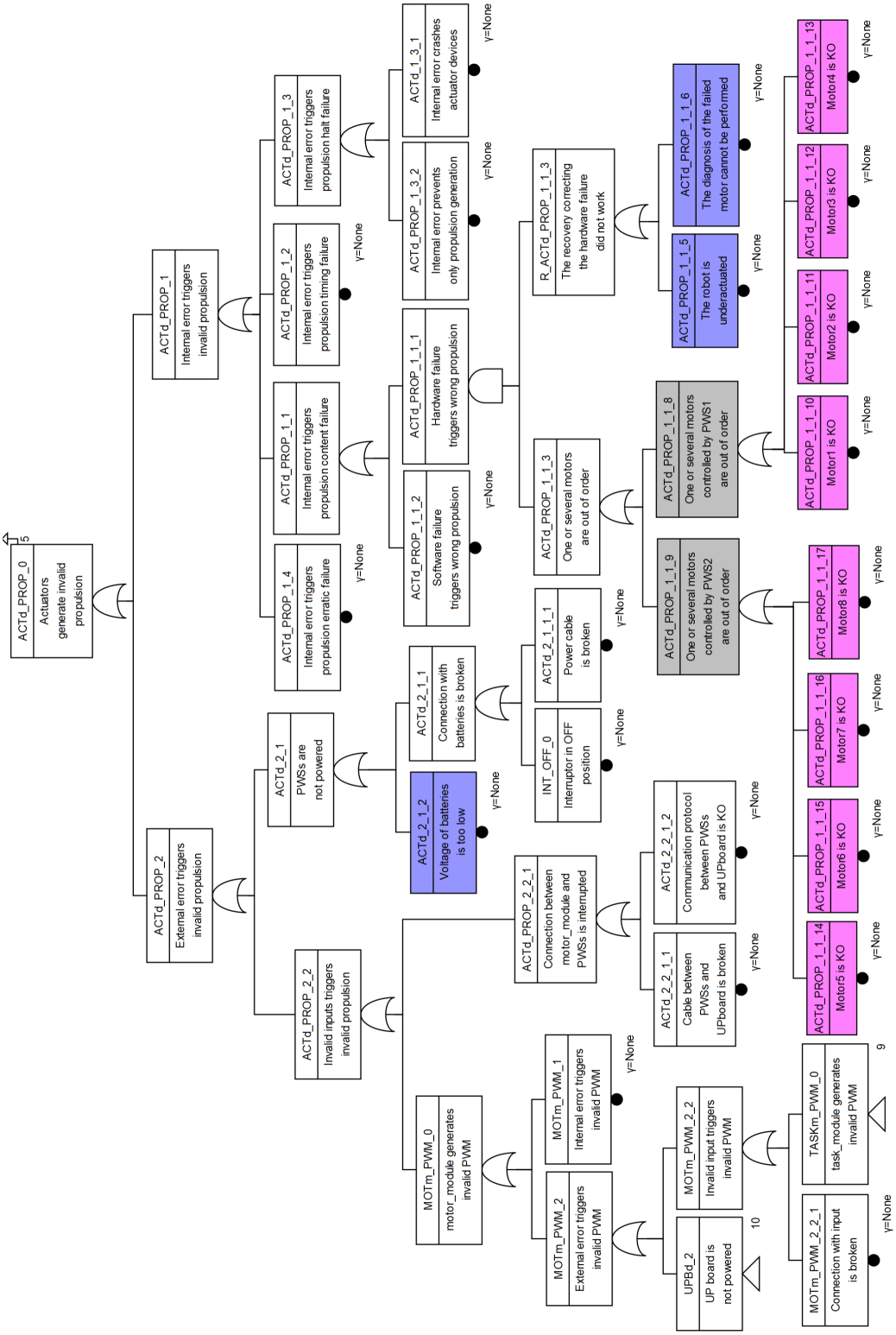


FIGURE A.1: Control loop fault tree

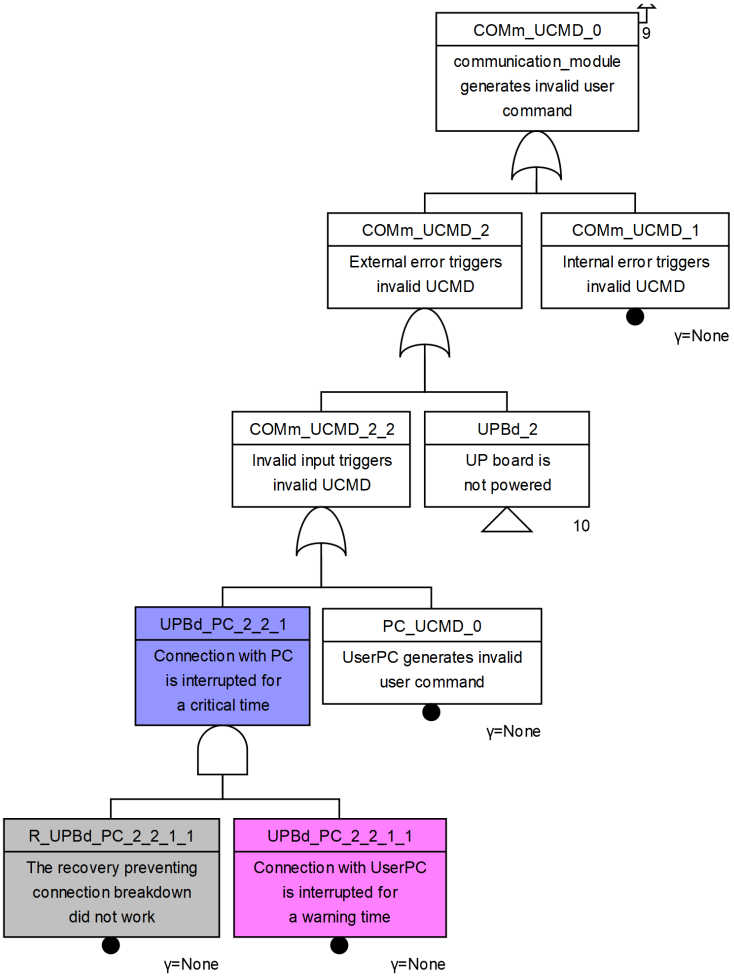


FIGURE A.2: Comm-PC module sub-fault tree

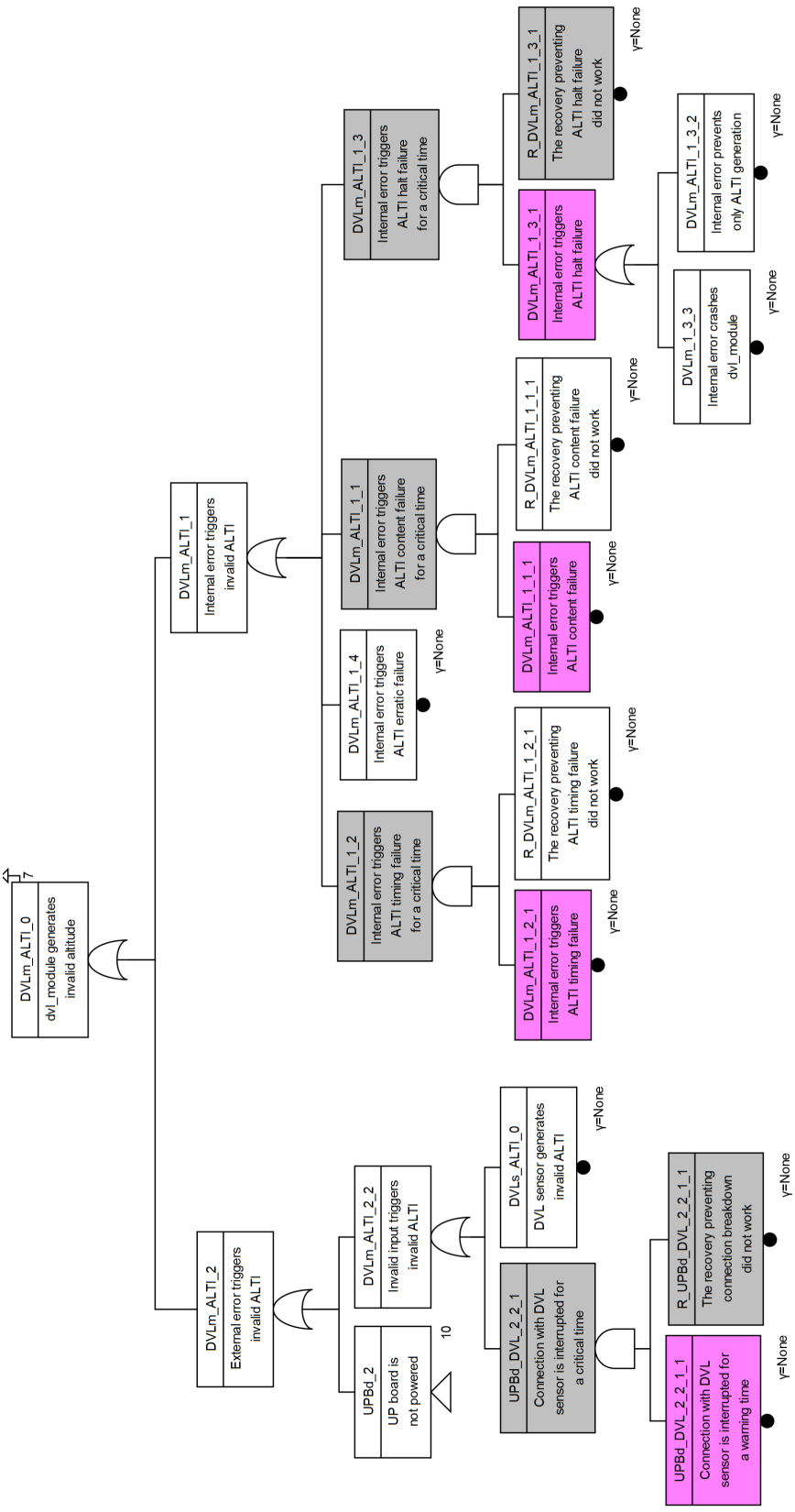


FIGURE A.3: DVL module sub-fault tree

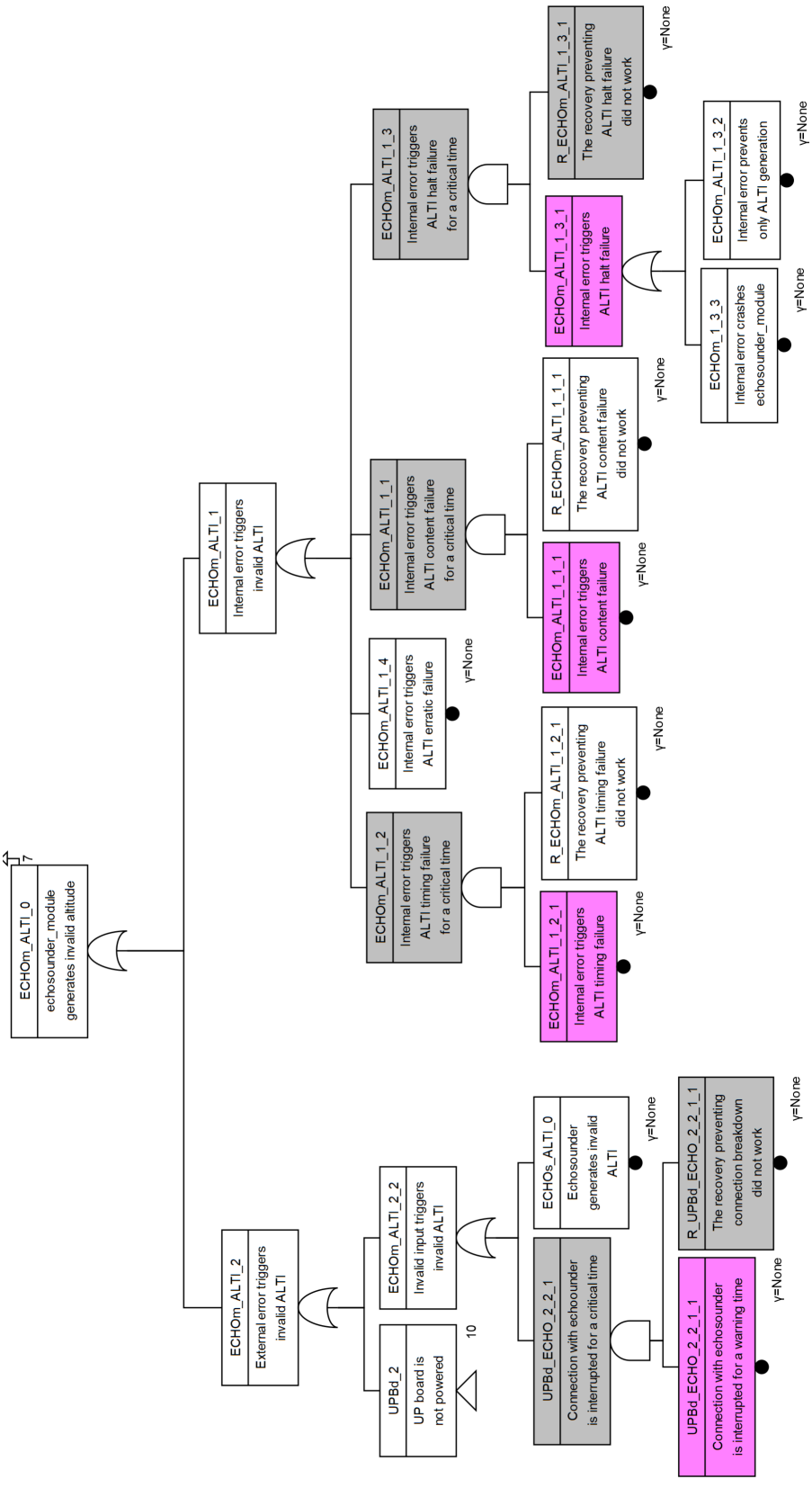


FIGURE A.4: Echosounder module sub-fault tree

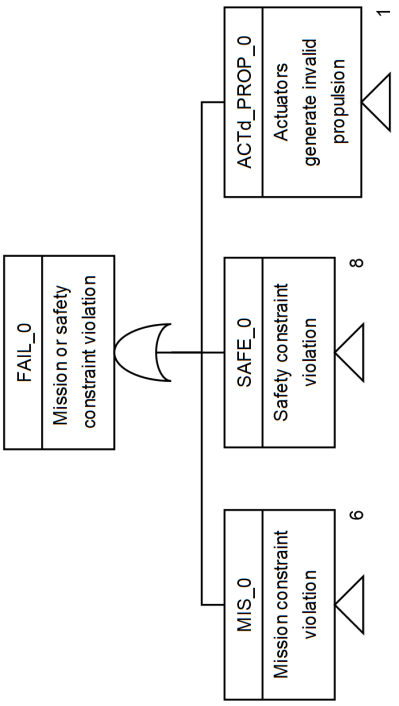


FIGURE A.5: Concatenation of all FTs

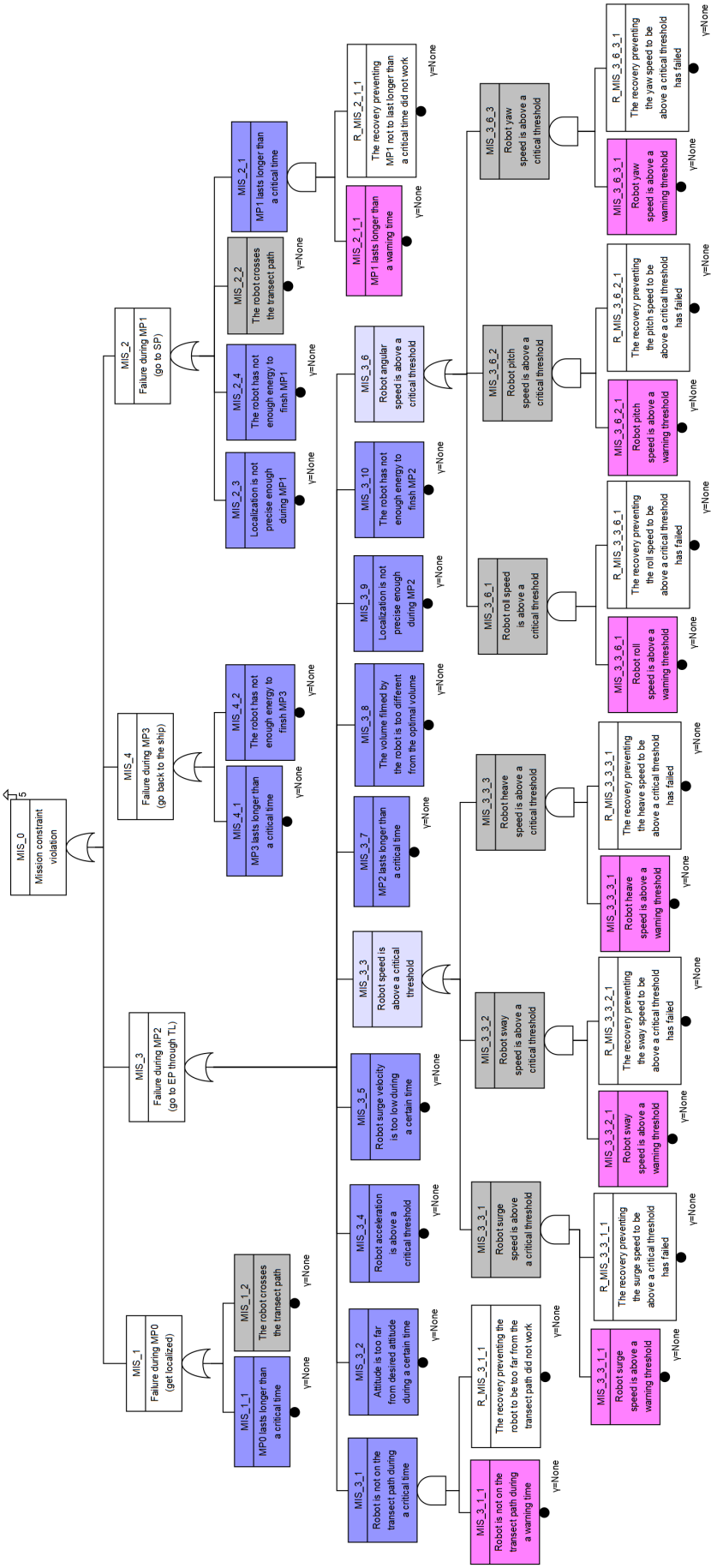


FIGURE A.6: Mission fault tree

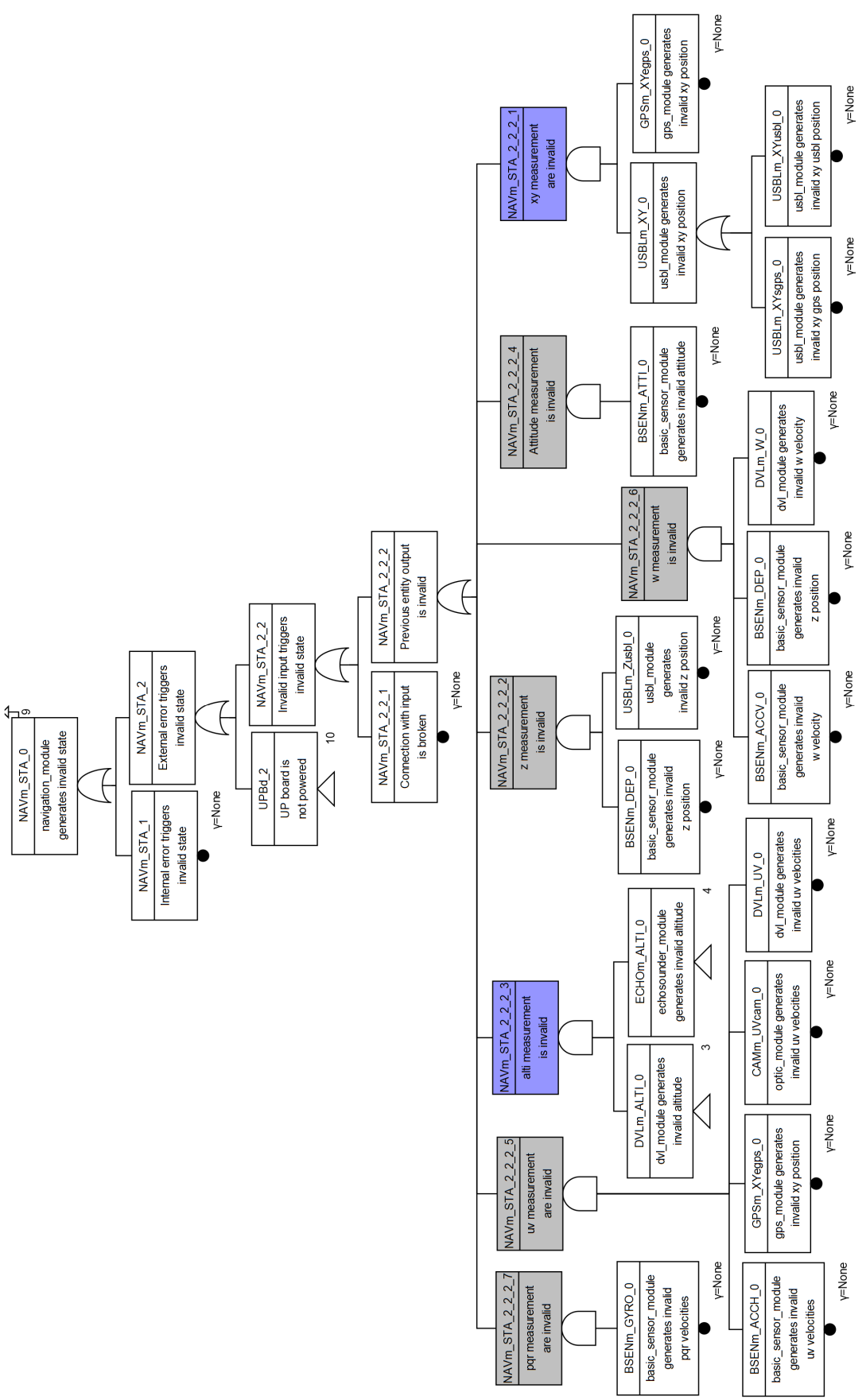


FIGURE A.7: NAV module sub-fault tree

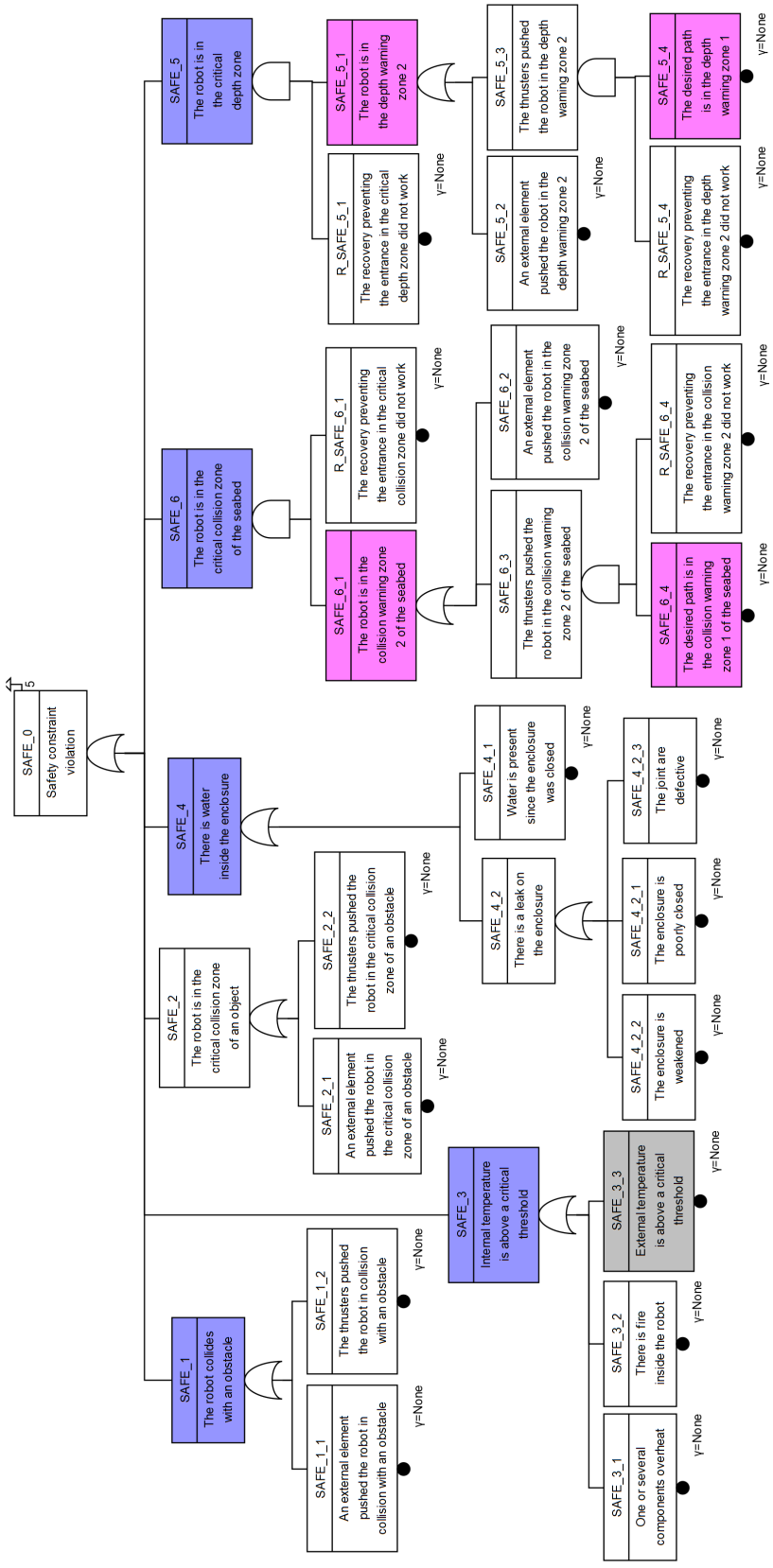


FIGURE A.8: Safety fault tree

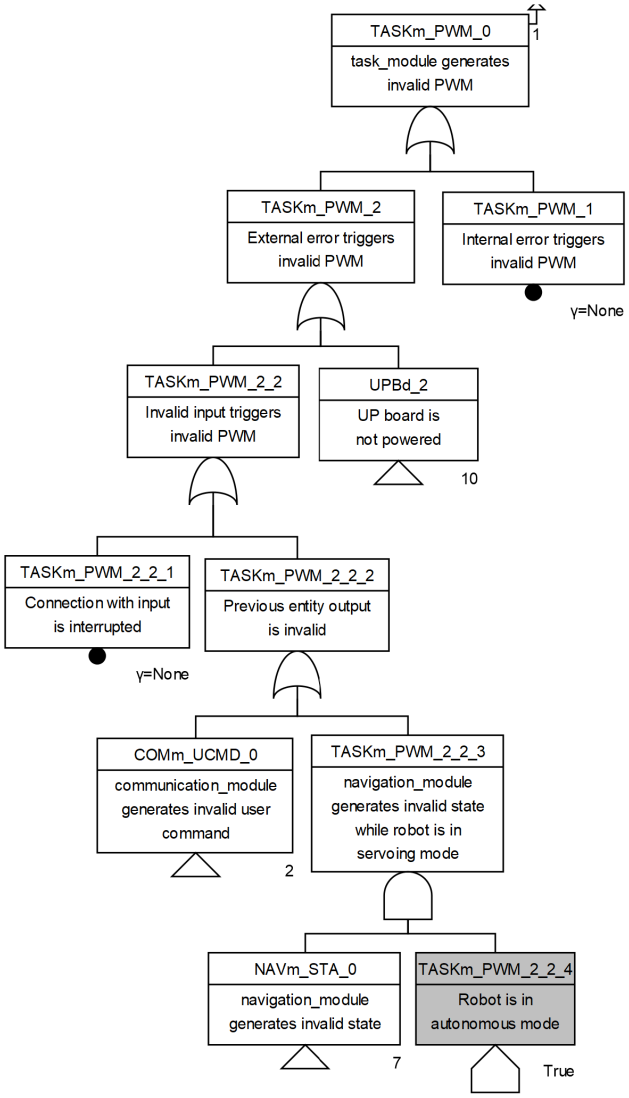


FIGURE A.9: Task module sub-fault tree

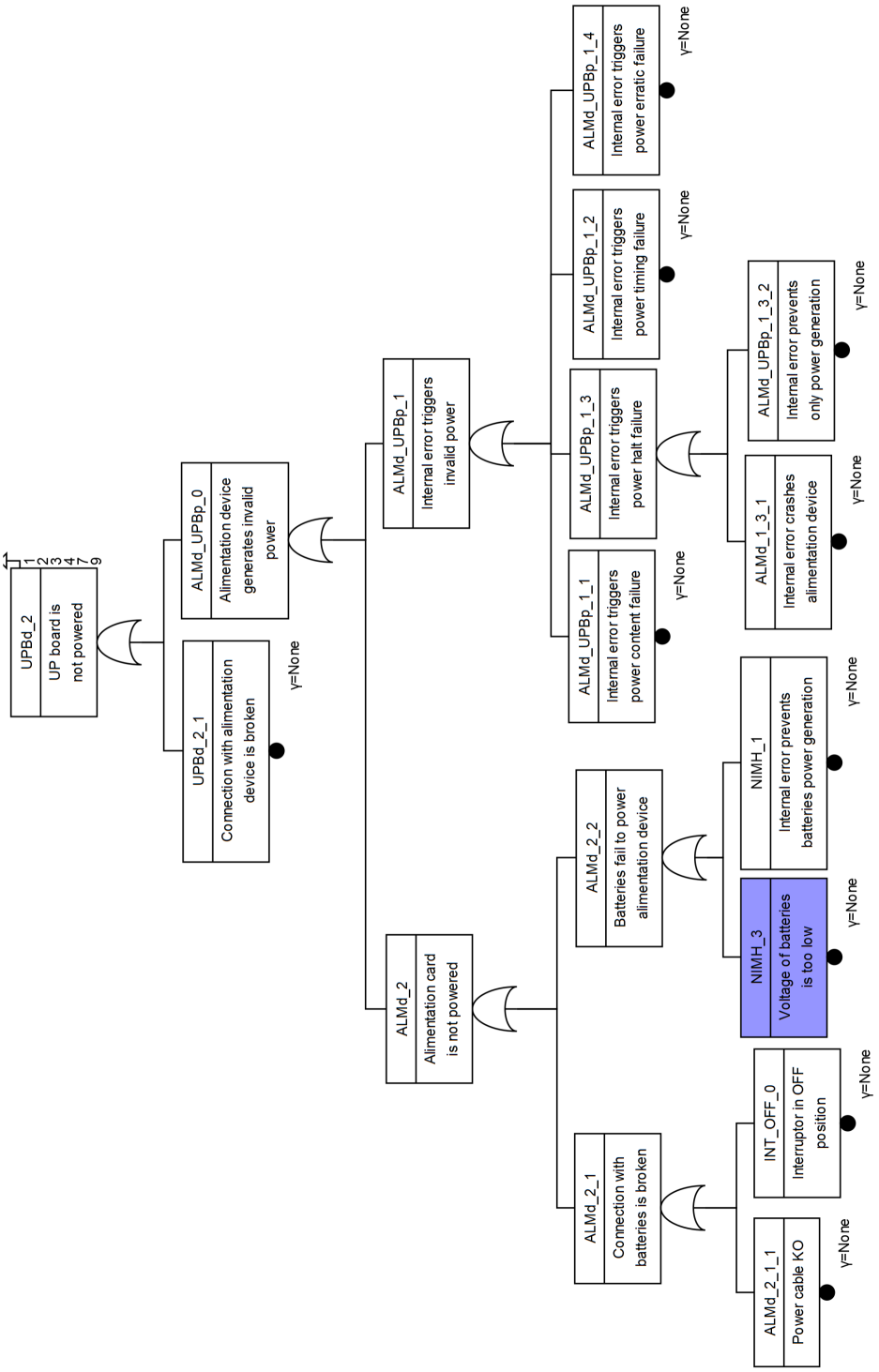


FIGURE A.10: UPB Power sub-fault tree

Appendix B

Estimating the physical parameters

This appendix describes the physical parameters of the robot along with their values.

B.1 Dynamic equation

The dynamic equation of the robot is given by:

$$M\dot{\nu} + C(\nu)\nu + D(\nu)\nu + g(\eta) = \tau + \tau_{ext} \tag{B.1}$$

with η the robot position and attitude, ν the robot speed and speed rotation, M, C, D, g, τ and τ_{ext} respectively the mass matrix, Coriolis matrix, damping matrix, gravity effects, actuators and external forces and moments on the robot in its body frame.

B.2 Mass matrix

The mass matrix contains the terms of the added mass corresponding to the inertia of the water around the robot. They are noted $X_{\dot{u}}$ to $N_{\dot{r}}$ and are given by abacus:

$$M = \begin{bmatrix} m + X_{\dot{u}} & 0 & 0 & 0 & 0 & 0 \\ 0 & m + Y_{\dot{v}} & 0 & 0 & 0 & 0 \\ 0 & 0 & m + Z_{\dot{w}} & 0 & 0 & 0 \\ 0 & 0 & 0 & I_x + K_{\dot{p}} & 0 & 0 \\ 0 & 0 & 0 & 0 & I_y + M_{\dot{q}} & 0 \\ 0 & 0 & 0 & 0 & 0 & I_z + N_{\dot{r}} \end{bmatrix} \tag{B.2}$$

m is the dry mass of the robot is estimated while weighing the robot. I_x, I_y and I_z are the principal inertial moment. The general inertial moment along x axis is expressed as:

$$I_x = \int \sqrt{y^2 + z^2}\rho dx dy dz \tag{B.3}$$

From equation B.3, assuming an approximation of the robot with a uniformly dense parallepiped, we obtain the equation:

$$I_x = m\frac{{l_y}^2 + {l_z}^2}{3} \tag{B.4}$$

Given the physical parameters, we express the mass matrix:

$$M = \begin{bmatrix} 32.5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 34.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 39.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.48 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.65 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.94 \end{bmatrix} \tag{B.5}$$

B.3 Coriolis and centripetal effects

We define S a skew-symmetric matrix as:

$$S(\lambda) = \begin{bmatrix} 0 & -\lambda_3 & \lambda_2 \\ \lambda_3 & 0 & -\lambda_1 \\ -\lambda_2 & \lambda_1 & 0 \end{bmatrix} \tag{B.6}$$

with $\lambda = [\lambda_1, \lambda_2, \lambda_3]^T$. Writing the mass matrix as an assembly of 3x3 matrices:

$$M = \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix} \tag{B.7}$$

The Coriolis and Centripetal can be expressed as:

$$C(\nu) = \begin{bmatrix} 0_{3x3} & -S(M_{11}\nu_1 + M_{12}\nu_2) \\ -S(M_{11}\nu_1 + M_{12}\nu_2) & -S(M_{21}\nu_1 + M_{22}\nu_2) \end{bmatrix} \tag{B.8}$$

with $\nu_1 = [u, v, w]^T$ and $\nu_2 = [p, q, r]^T$. The Coriolis effects are neglected due to the small distances to travel and small speeds of the robot.

B.4 Hydrodynamic matrix

We assume the damping terms are diagonal, with a linear and turbid effects. We simplify the model and assume that the robot has 2 planes of symmetry along the x and y axes. Finally, we add a lift term due to the shape of the robot. The hydrodynamic matrix takes the form:

$$D(\nu) = \begin{bmatrix} D_{11}(u) & 0 & 0 & 0 & 0 & 0 \\ 0 & D_{22}(v) & 0 & 0 & 0 & 0 \\ L_{l33}sign(u) + L_{t33}u & 0 & D_{33}(w) & 0 & 0 & 0 \\ 0 & 0 & 0 & D_{44}(p) & 0 & 0 \\ 0 & 0 & 0 & 0 & D_{55}(q) & 0 \\ 0 & 0 & 0 & 0 & 0 & D_{66}(r) \end{bmatrix} \tag{B.9}$$

with $D_{ii}(\alpha) = D_{lii} + D_{tii}|\alpha|$. We used the software ANSYS ([5]) to determine the different value of the coefficients. We varied the speed of the fluid for each simulations and launched a solver of the software to estimate the drag and lift effects. We adapted the result with test in the field, as most of the coefficients were badly estimated during the simulation:

$$D(\nu) = \begin{bmatrix} 10+40|u| & 0 & 0 & 0 & 0 & 0 \\ 0 & 28+100|v| & 0 & 0 & 0 & 0 \\ 5sign(u)+5u & 0 & 35+240|w| & 0 & 0 & 0 \\ 0 & 0 & 0 & 2+3|p| & 0 & 0 \\ 0 & 0 & 0 & 0 & 5+9|q| & 0 \\ 0 & 0 & 0 & 0 & 0 & 4+5|r| \end{bmatrix} \tag{B.10}$$

B.5 Hydrostatic vector

We define the cross product operator $\times$ as

$$\lambda \times a = S(\lambda)a \tag{B.11}$$

with S the skew-symmetric matrix define in section B.3. We note W the gravity on the robot and B the buoyancy total effect. The hydrostatics of a marine vehicle is expressed as:

$$g(\eta) = \begin{bmatrix} R^T({f_g}^n + {f_b}^n) \\ {r_g}^b \times R^T {f_g}^n + {r_b}^b \times R^T {f_b}^n \end{bmatrix} \tag{B.12}$$

with R the rotation matrix expressing the attitude of the robot in the NED frame, ${f_g}^n = [0,0,-W]^T$, ${f_b}^n = [0,0,B]^T$, ${r_g}^b = [x_g, y_g, z_g]^T$ the center of gravity and ${r_b}^b = [x_b, y_b, z_b]^T$ the center of buoyancy. Supposing the robot is not far from the surface (<100m), the gravity W is calculated by:

$$W = m * 9.8 = 289.1N \tag{B.13}$$

The buoyancy B is estimated by maintaining the robot with $\phi = 0$ and $\theta = 0$ at constant depth. From the mean estimated force to counter the heave motion of the robot, we can deduce the buoyancy:

$$B = W + \frac{\int_0^T Z dt}{T} = 295.1N \tag{B.14}$$

The center of the robot is defined to be at the middle between the center of gravity and the center of buoyancy. There is a possibility to balance the weight of the robot on 4 points to adjust the center of gravity and buoyancy. We adjust both centers on the z axis of the robot ($x_b = y_b = x_g = y_g = 0$) and we determine z_g and z_b by maintaining the robot with $\phi = 0$ and $\theta = \pi/2$:

$$z_g = -z_b = \frac{\int_0^T K}{T(W+B)} = 0.011m \tag{B.15}$$

The final simplified model is expressed as:

$$M\dot{\nu} + D(\nu)\nu + g(\eta) = \tau \tag{B.16}$$

Appendix C

Implementation of the extended Kalman filter

This appendix describes the implementation of the Extended Kalman Filter (EKF) in the robot.

C.1 A generic implementation of the extended Kalman filter

The extended Kalman filter has been implemented as an independent PID package ([130]) for a general mobile robot using sensors updates. The package is heavily based on the Eigen C++ library to perform mathematical calculations ([41]). An extended Kalman filter is used to estimate the state of a system with nonlinearities. During one iteration, it consists in two steps: prediction phase and innovation phase.

After the discretization of the state, during the prediction phase, a predicted state $\chi_{k|k-1}$ is estimated depending on the previous state:

$$\hat{\chi}_{k|k-1} = A_k \hat{\chi}_{k-1|k-1} + B_k u_k \tag{C.1}$$

with $\hat{\chi}_{k-1|k-1}$ the previous estimated state, A_k the transition matrix, B_k the control-input matrix and u_k the control vector. The predicted covariance matrix is also calculated with:

$$P_{k|k-1} = A_k P_{k-1|k-1} {A_k}^T + Q_k \tag{C.2}$$

with $P_{k-1|k-1}$ the previous covariance and Q_k the vector of the process noise covariance.

During the innovation phase, the sensors are used to correct the predicted state. The package provides a method to declare the sensors, along with their size and the element of the state vector to which they contribute. The measurement pre-residual y_k is calculated with:

$$y_k = z_k - H_k \hat{\chi}_{k|k-1} \tag{C.3}$$

with z_k the measurement of the sensors and H_k the observation model established with the method previously cited. We define the optimal Kalman gain K_k as:

$$K_k = P_{k|k-1} {H_k}^T (H_k P_{k|k-1} {H_k}^T + R_k)^{-1} \tag{C.4}$$

with R_k the vector of the sensor noise covariance. The estimated state at iteration k $\hat{\chi}_{k|k}$ is then given by:

$$\hat{\chi}_{k|k} = \hat{\chi}_{k|k-1} + K_k y_k \tag{C.5}$$

and the updated covariance at iteration k $P_{k|k}$ is given by:

$$P_{k|k} = (I_{6x6} - K_k H_k) P_{k|k-1} \tag{C.6}$$

During one iteration, the estim_state() method of the class Kalman runs the following methods in this order:

- init_prediction(): a virtual method used to implement the matrices A_k, B_k and the vector u_k.
- predict_state(): performs the calculations describe in equations C.1 and C.2
- arrange_prediction(): a virtual method used to arrange the predicted state if constraints need to be verify before the innovation phase.
- add_sensors(): a virtual method used to inform the sensors data that have been updated and their correspondence with the state elements. A method add_sensor_data() taking the value of the sensor, its noise covariance, its correspondence with the state as argument and filling the adequate matrices is provided in Kalman class.
- innovate_state(): performs the calculations describe in equations C.3, C.4, C.5 and C.6
- arrange_innovation(): a virtual method used to arrange the estimated state if constraints need to be verify.

From the described package, we will explain how we adapt it to REMI underwater robot in the next section.

C.2 Implementation of the extended Kalman filter on REMI

We run the EKF on a independent module where an iteration is performed every $dt = 100ms$. We express the Kalman state as $\chi = [\nu, \eta_q]^T$. We use the quaternion in the Kalman state to avoid the gimbal lock.

To express the dynamic model with the form of the equation C.1, we choose to discretize the dynamic model expressed in the continuous form B.16:

$$M\frac{\hat{\nu}_{k|k-1} - \hat{\nu}_{k-1|k-1}}{dt} + D(\hat{\nu}_{k-1|k-1})\hat{\nu}_{k-1|k-1} + g(\hat{\eta}_{k-1|k-1}) = \tau_{k-1} \tag{C.7}$$

We have then:

$$\hat{\nu}_{k|k-1} = (I_{6x6} - dtM^{-1}D(\hat{\nu}_{k-1|k-1}))\hat{\nu}_{k-1|k-1} + dtM^{-1}(\tau_{k-1} - g(\hat{\eta}_{k-1|k-1})) \tag{C.8}$$

On the other way, the pose of the robot is linked with its speed and rotational speed by:

$$\hat{\eta_q}_{k|k-1} = \hat{\eta_q}_{k-1|k-1} + dtR_{7x6}(\eta)\hat{\nu}_{k-1|k-1} \tag{C.9}$$

with

$$R_{7x6}(\eta) = \begin{bmatrix} R & 0_{3x3} \\ 0_{4x3} & T_Q \end{bmatrix} \tag{C.10}$$

where T_q is the transition matrix so that the derivative of the attitude expressed in quaternion:

$$T_Q = \frac{1}{2} \begin{bmatrix} -q_x & -q_y & -q_z \\ q_w & -q_z & q_y \\ q_z & q_w & -q_x \\ -q_y & q_x & q_w \end{bmatrix} \tag{C.11}$$

so that with $\omega = [p, q, r]^T$:

$$\dot{Q} = T_Q \omega \tag{C.12}$$

We can deduce that:

$$\begin{bmatrix} \hat{v}_{k|k-1} \\ \hat{\eta}_{q_{k|k-1}} \end{bmatrix} = A_k \begin{bmatrix} \hat{v}_{k-1|k-1} \\ \hat{\eta}_{q_{k-1|k-1}} \end{bmatrix} + B_k u_k \tag{C.13}$$

with

$$A_k = \begin{bmatrix} I_{6x6} - dtM^{-1}D(\hat{v}_{k-1|k-1}) & 0_{6x7} \\ dtR_{7x6}(\eta) & I_{7x7} \end{bmatrix} \tag{C.14}$$

$$B_k = \begin{bmatrix} -dtM^{-1} \\ 0_{7x6} \end{bmatrix} \tag{C.15}$$

and

$$u_k = \tau_{k-1} - g(\hat{\eta}_{k-1|k-1}) \tag{C.16}$$

It is also possible to add the sea current effect or the strength of the cable in the u_k vector (if there is a sensor capable of providing this information). During the arrange_prediction() and arrange_innovation(), we normalized the predicted and estimated quaternion so that its euclidean norm stays equal to 1. We also limit the robot z position to a positive value because it cannot be above the water.

Appendix D

Unit testing measurements

This appendix details the unit testing introduced in section 5.3. The 15 unit tests results are synthesized Tab 5.5 and Tab 5.6.

D.1 T1: Handling water inside enclosure

During T1, we verify the reaction of the robot to SAFE_4: *There is water inside the enclosure* (Fig A.8). During this test, the robot performs a transect at 4m depth in the field (Crès lake). We inject in the software the fault *Force the water_inside_enclosure bit* at t=80s.

At t=80.082s, the fault tolerance manager indicates the events SAFE_4 and SAFE_0 are true. It informs the mission manager that a top event has been deduced (SAFE_0) and the causes of this event (SAFE_4). At t=80.094s, the transect is interrupted and the *surface_task* is launched by the mission manager (Fig D.1). At t=99.374s, the robot reaches the surface and the mission manager stops the robot.

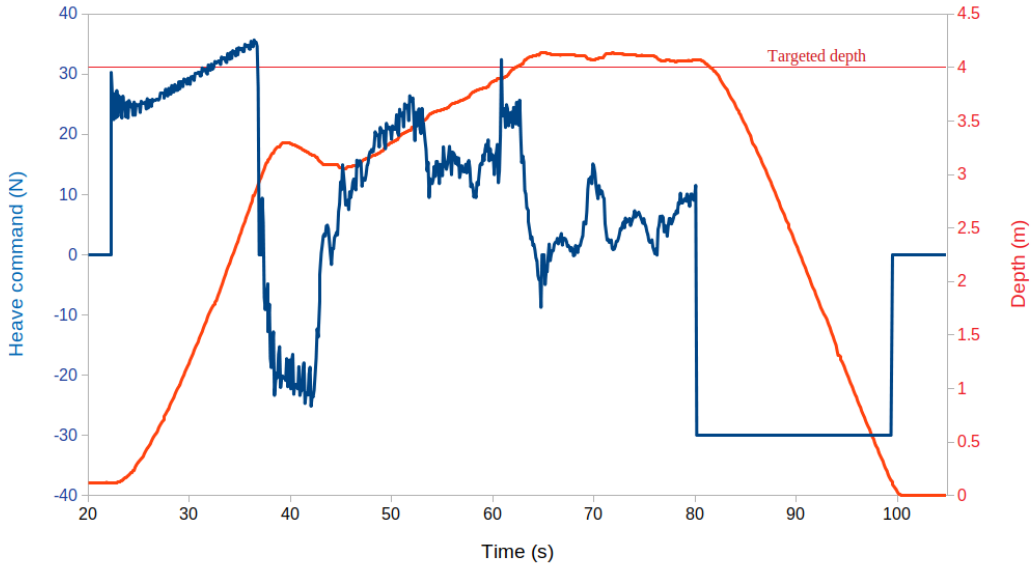


FIGURE D.1: Depth and heave command of the robot during T1

D.2 T2: Handling robot in the critical collision zone of the seabed

During T2, we verify the reaction of the robot to SAFE_6: *The robot is in the critical collision zone of the seabed* (Fig A.8). The robot is in the critical collision zone of the seabed if its altitude is below 0.5m. During this test, the robot performs a transect at 3m depth in simulation, whereas the seabed drops from 5m depth to 2m depth (water column) on the desired trajectory of the robot.

At t=132.235s, the fault tolerance manager indicates the events SAFE_6 and SAFE_0 are true. It informs the mission manager that a top event has been deduced (SAFE_0) and the causes of this event (SAFE_6). At t=132.314s, the transect is interrupted and the *surface_task* is launched by the mission manager (Fig D.2). At t=143.3s, the robot reaches the surface and the mission manager stops the robot.

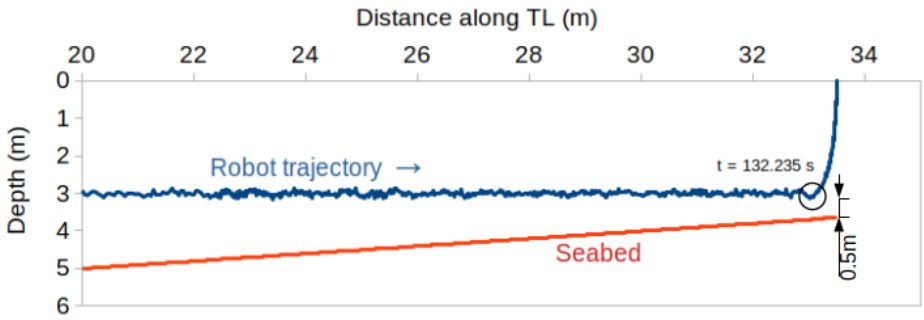


FIGURE D.2: Depth of the robot during T2

D.3 T3: Handling robot in the warning collision zone 2 of the seabed

During T3, we verify the reaction of the robot to SAFE_6_1: *The robot is in the collision warning zone 2 of the seabed* (Fig A.8). The robot is in the collision warning zone 2 of the seabed if its altitude is below 0.8m. For that we decide to perform a transect in the field at 4.5m depth taking SP with an important water column ($> 6m$) and EP with a little water column ($< 4m$) (Fig D.3). During T3, we deactivate the local recovery (LR8) associated with the entry of the target position in the warning collision zone 1 of the seabed. The local recovery associated to SAFE_6_1 (LR9) consists in overriding the actual heave command by -40 N, so that the robot moves away from the seabed promptly.

During its transect, the robot oscillates around 0.8m with an amplitude of 0.5m. Each time the robot's altitude is below the threshold, the LR changed the heave command to -40N and thus the robot raises (Fig D.4). SAFE_6 is never detected and thus the robot continued its missions until another event indicates the robot is too far from the transect line.

As a remark, the estimated altitude is uncertain around 0.5m because the altitude range of the echosounder is between 30m and 0.5m.

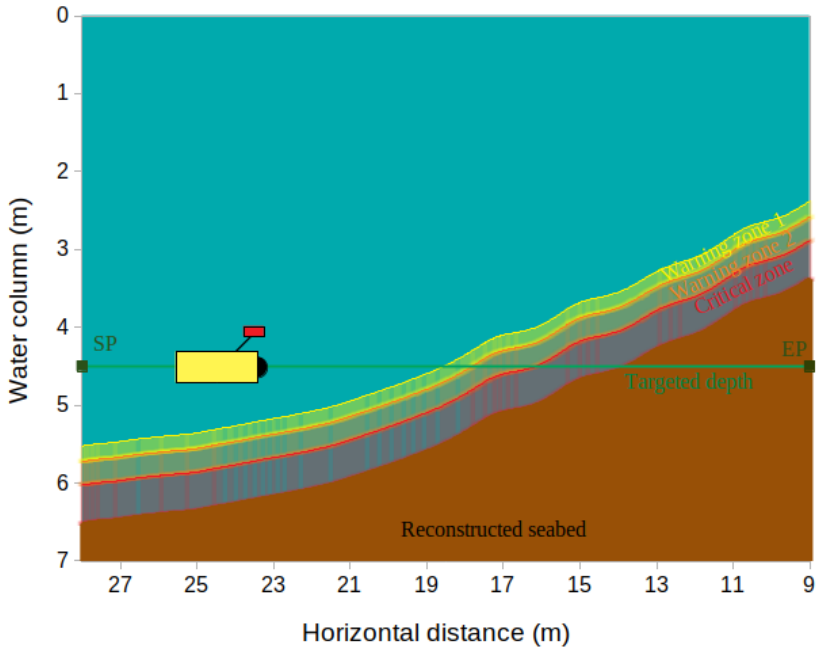


FIGURE D.3: Context of test during T3 and T4

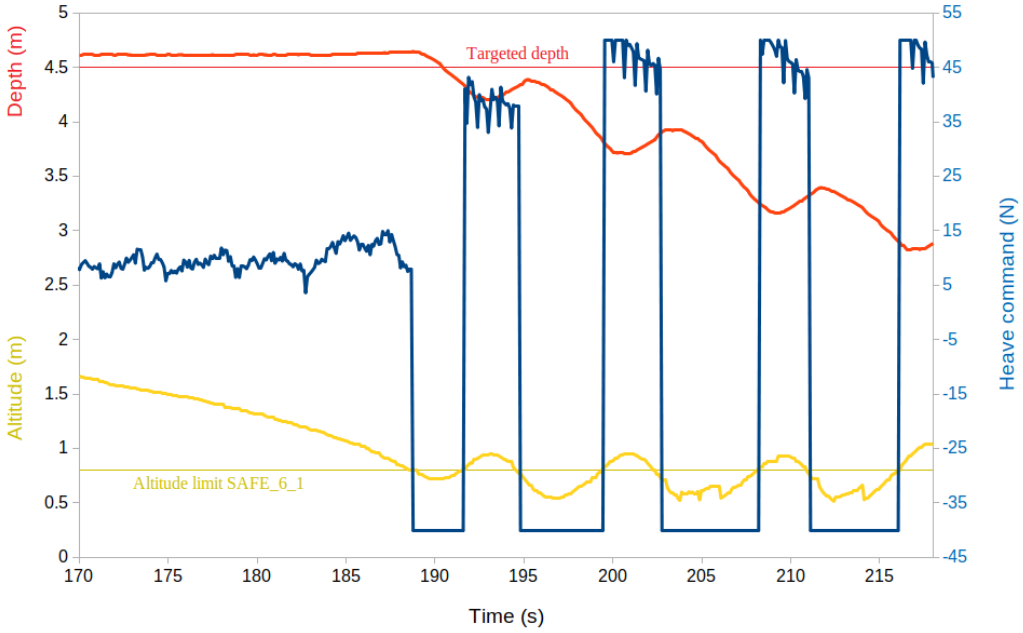


FIGURE D.4: Depth, altitude and reaction of the robot during T3

D.4 T4: Handling robot in the warning collision zones of the seabed

During T4, we verify the reaction of the robot to SAFE_6_1 (*The robot is in the collision warning zone 2 of the seabed*) and SAFE_6_4 (*The desired path is in the collision warning zone 1 of the seabed*) (Fig A.8). The robot is in the collision warning zone 2 of the seabed if its altitude is below 0.8m. The desired path is in the collision warning zone 1 of the seabed if its altitude is below 1.0m. As for T3, we decide to perform the same transect in the field at 4.5m depth taking SP with an important water column ($> 6m$) and EP with a little water column ($< 4m$) (Fig D.3).

The local recovery associated to SAFE_6_1 (LR9) consists in overriding the actual heave command by -40 N, so that the robot moves away from the seabed promptly. The local recovery associated to SAFE_6_4 (LR8) consists in overriding the target altitude path by 1.0m, so that the robot moves away from the seabed softly. LR8 and LR9 are not compatible and we set LR9 with more priority, as its associated event is more severe.

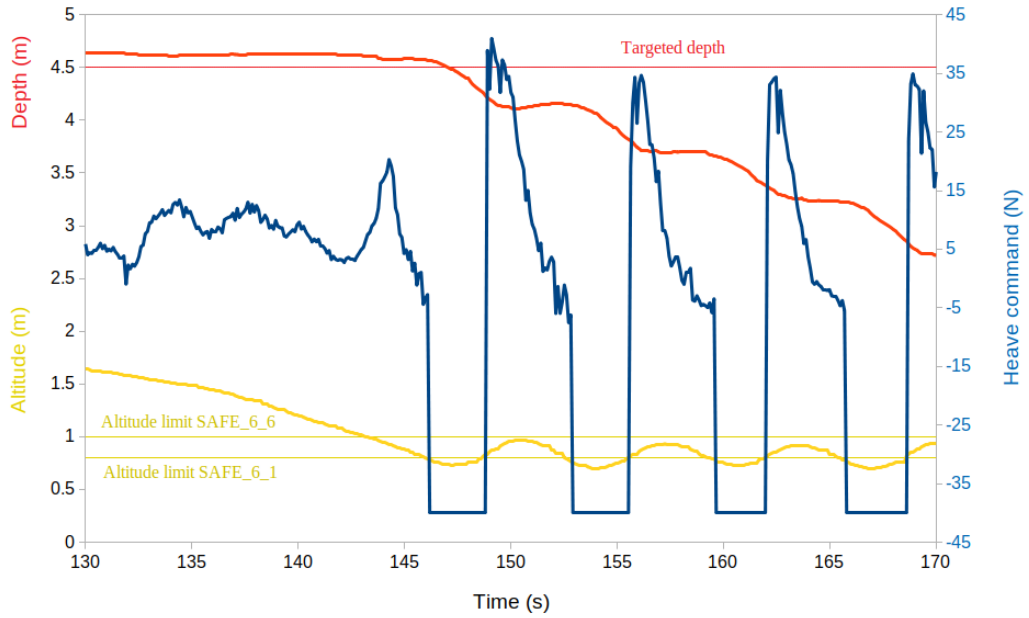


FIGURE D.5: Depth, altitude and reaction of the robot during T4

When the robot's target enters the warning zone 1 (SAFE_6_4), LR8 tends to keep the target position at the upper limit of the warning zone 1. As a consequence, the heave command decreases progressively to maintain the robot altitude at 1m. Nevertheless, the seabed is too steep and the robot enters the warning zone 2 (SAFE_6_1). From there, both events SAFE_6_4 and SAFE_6_1 are detected as true. Nevertheless, since their local recoveries are not compatible, the highest priority local recovery is then triggered alone (LR9) (Fig D.5).

As a consequence, the robot oscillates around 0.85m with an amplitude of 0.3m. The overall behavior of the robot is therefore better in the configuration of T4 with the 2 local recoveries activated than in T3 with only LR9 activated. SAFE_6 is never detected and thus the robot continued its missions until another event indicates the robot is too far from the transect line.

D.5 T5: Handling collision with an obstacle

During T5, we verify the reaction of the robot to SAFE_1: *The robot collides with an obstacle* (Fig A.8). The detection of SAFE_1 involves 2 thresholds: the absolute value of acceleration along any axis should be superior to a threshold ($2.5m.s^{-2}$) during more than a certain amount of time (0.2s). These thresholds have been set experimentally to limit false detection and non-detection. During T5, the robot performs a transect in the field at 3.0m depth.

Around t=140s, we pull violently the robot cable to simulate a collision. The fault tolerance manager verifies the acceleration of the robot (Fig D.6), and at t=144.119s, SAFE_1 is detected. The transect is interrupted by the mission manager at t=144.167s. The robot stops its actuators and surfaces with its positive buoyancy.

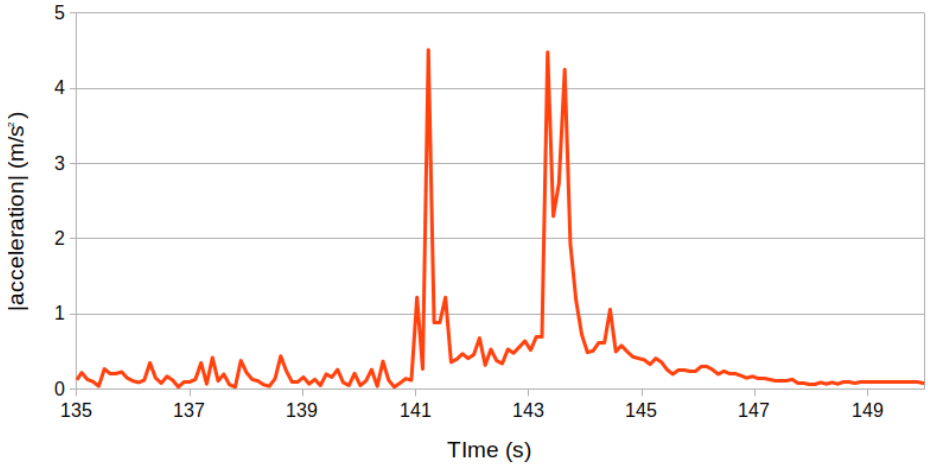


FIGURE D.6: Acceleration measurement during T5

D.6 T6: Handling robot not on the transect line

During T6, we verify the reaction of the robot to MIS_3_1_1 (*the robot is not on the transect path during a warning time*) and MIS_3_1 (*the robot is not on the transect path during a critical time*) (Fig A.6). MIS_3_1_1 is detected as true if the distance between the robot and the transect line is greater than 1.5m for more than 6s. MIS_3_1 is detected as true if the distance between the robot and the transect line is greater than 2.0m for more than 10s.

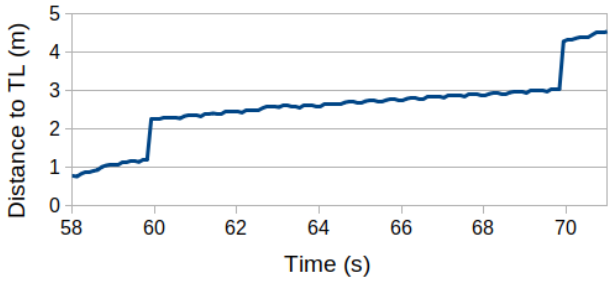


FIGURE D.7: Distance between the robot and TL during T6

During T6, the robot performs a transect in simulation at 3.0m depth. We simulate a side sea current of 0.63m/s to move the robot away from the transect line during MP2. MP2 starts at t=58.4s and the robot deviates from the trajectory (Fig D.7). At t=65.8s, MIS_3_1_1 is detected. As a consequence, compatible local recoveries are attempted (LR4, LR5, LR6, LR7) to get closer to TL. Unfortunately, the sea current is too strong and the robot cannot get closer to TL. At t=69.8s, MIS_3_1 is detected and the transect is stopped by the mission manager, which decides to reattempt the transect once.

As a remark, the distance leaps present on Fig D.7 at t=60s and t=70s are caused by an update of the simulated UBSL sensor measurement.

D.7 T7: Handling poor localization during MP2

During T7, we verify the reaction of the robot to MIS_3_9: *localization during MP2 is too poor* (Fig A.6). This event is detected as true if the variance along x or y axis calculated by the extended Kalman filter is superior to 13.0m^2 (equivalent to $\sigma_{x,y} > 3.6m$, with σ_x, y the standard deviation along x or y axis).

To illustrate the difference between an acceptable transect and a non-acceptable transect from the localization point of view, we decide to perform 2 transects in the field (Fig D.8).

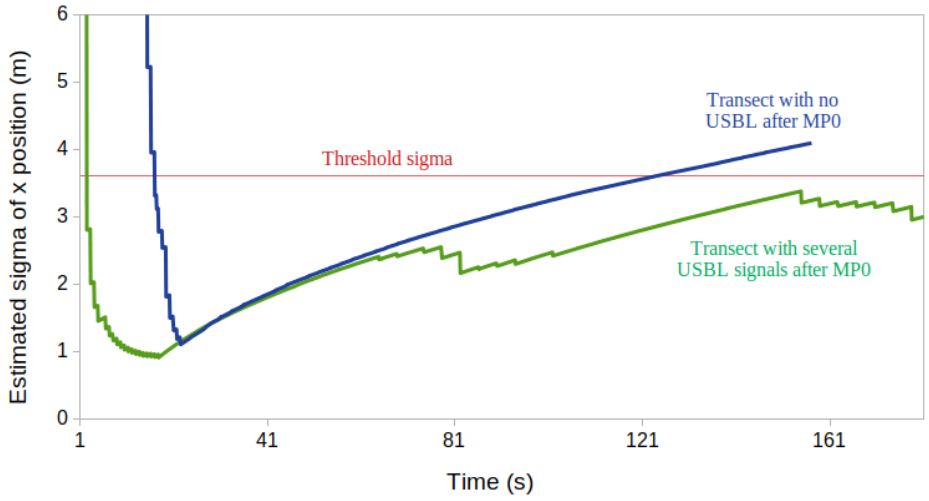


FIGURE D.8: Estimated standard deviation of the x position during 2 transects

During the first transect (green curve), we keep the USBL surface beacon in place. At the early stage of MP0, the robot is at the surface and the GPS signal is received frequently. The standard deviation (sigma) of the x position is then very small (around 1m). After the robot starts diving, the USBL signal is received once during MP0 and several times during MP1 and MP2. The standard deviation of the x position globally increases with time as the robot is in the water. However, each time the USBL signal is received, the standard deviation of the x position decreases according to the estimated precision of the received signal. Therefore, the standard deviation of the x position does not exceed 3.6m and the transect is considered valid for this point. We also note that the USBL signal is not received all the time, as there is no USBL signal between 100s and 150s.

During the second transect (blue curve), we remove the surface USBL beacon after the first signal is received. At t=42s, a restart of the *usbl_module* is attempted and at t=82s, the definitive halt failure of USBL driver output is detected. At t=124s, the standard deviation of the estimated x position rises above the threshold of 3.6m and MIS_3_9 is detected. Thus, the mission manager triggers the *surface_task* and the robot surfaces at t=129.1s.

D.8 T8: Handling MP1 lasts too long

During T8, we verify the reaction of the robot to MIS_2_1_1 (*MP1 lasts longer than a warning time*) and MIS_2_1 (*MP1 lasts longer than a critical time*) (Fig A.6). For that, we perform a classic transect in the field at 4.5m of depth. For this particular test, knowing that $u_{cruise}^{MP1} = 0.6m/s$ and $d^{MP1} = 15.66m$ (see section 4.3.3.2), MIS_2_1_1 is detected as true when MP1 lasts more than 51.76s and MIS_2_1 is detected as true when MP1 lasts more than 70.2s.

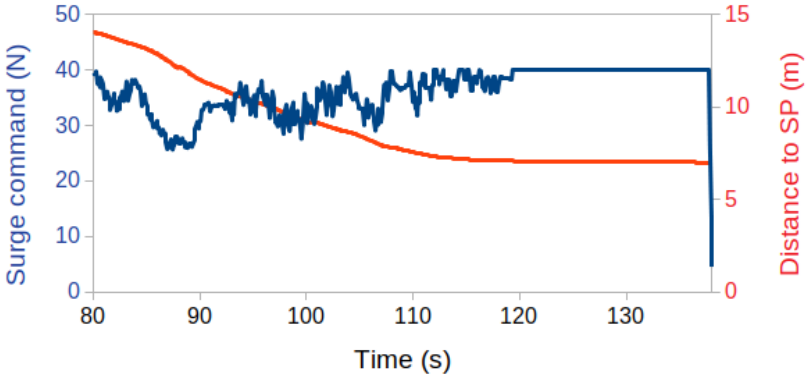


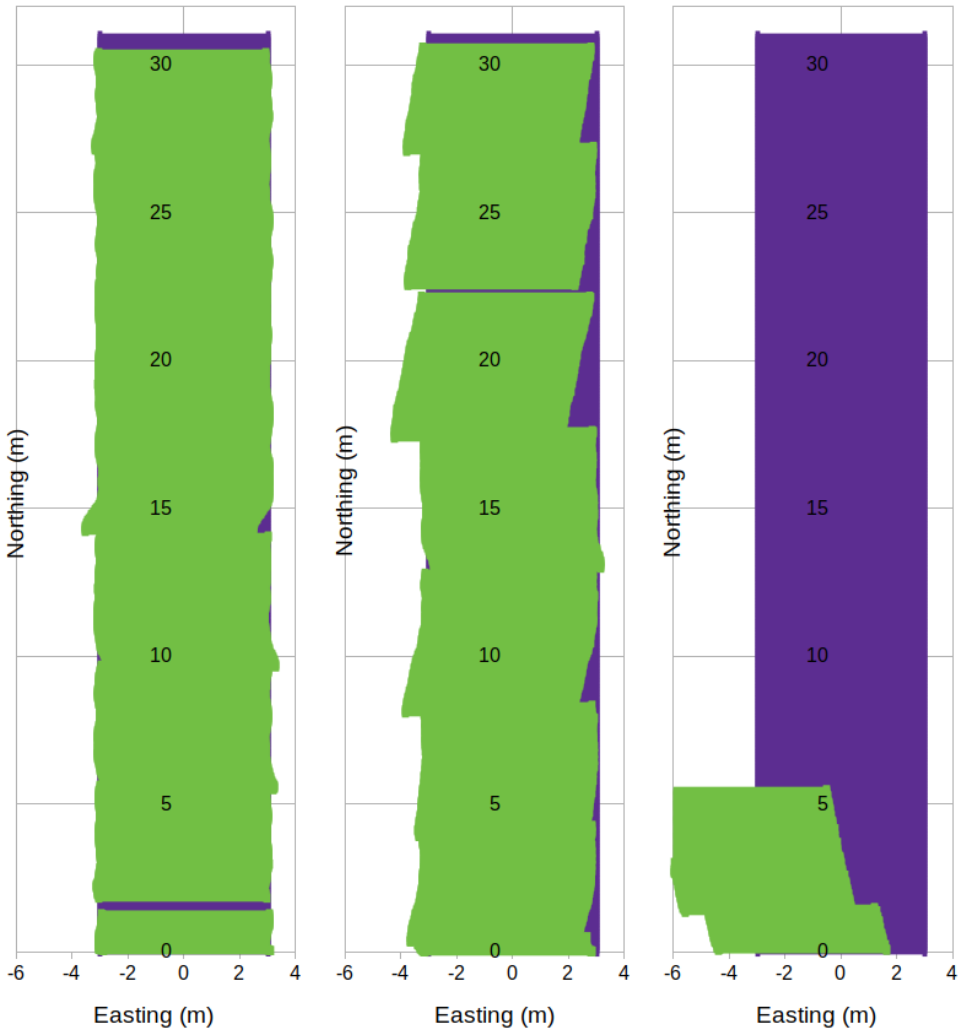
FIGURE D.9: Surge command and distance to SP during T8

During T8, MP1 starts at t=67.50s. We then softly pull the cable of the robot at t=100s, so it prevents the robot from moving forward (Fig D.9). At t=119.30s, event MIS_2_1_1 is detected. Its associated LR (LR3) consists in increasing the surge speed by 50%. As a consequence, the surge command increases to the maximum value of 40N. Despite the action of LR3 and since we continue to pull the cable, MIS_2_1 is detected at t=137.90s. The mission manager decides then to stop the mission.

D.9 T9: Handling camera coverage too low

During T9, we verify the reaction of the robot to MIS_3_8: *the volume filmed by the robot is too different from the optimal volume* (Fig A.6). We performs this test for 3 same transects in simulation while varying the lateral sea current. We also prevent the decisional recovery associated to MIS_3_1. In real time, the robot calculates the ratio of the volume that has been missed by the camera before its current position out of the total theoretical volume to be filmed. MIS_3_8 is detected as true if this ratio is superior to 0.1 (there must be a final coverage rate of at least 90% between the filmed volume and the theoretical volume).

We first perform a transect of 31m without sea current and we observe a final coverage rate of 98.3% (Fig D.10a). We then perform the same transect with a lateral sea current of 0.45m/s. The estimated final coverage is 91.8% (Fig D.10b). The green discontinuous zones represent a new measurement of the USBL sensor (leap of the position of the robot). Finally we perform the same transect with 0.67m/s of lateral sea current (Fig D.10c). At t=95.4s, MP2 starts and as expected, the coverage rate decreases below 90% at t=111.1s. As expected, the fault tolerant manager informs the mission manager that decides to stop and relaunch once the same transect.



(A) No current (B) 0.45m/s of lateral current (C) 0.67m/s of lateral current

FIGURE D.10: Coverage of the filmed zone during T9.
The purple area is the theoretical zone to film whereas the green area is the zone effectively filmed

D.10 T10: Handling not enough energy to perform MP1

During T10, we verify the reaction of the robot to MIS_2_4: *the robot has not enough energy to finish MP1* (Fig A.6). This event is detected as true if the voltage of the battery is inferior to a threshold that depends on the distance to travel. We set $C_v =$

10^{-3}, which means that we estimate the robot needs 1V to travel 1000m. The robot must have a minimum voltage of 11V at the end of MP1.

We program a new transect mission on the user interface and set the starting point of the transect at 5km from the current position of the robot. The initial voltage of the robot is 12.6V when it starts its mission in the field. At t=39.4s, MP1 is initiated and MIS_2_4 is detected 0.1s later. The transect is then interrupted by the mission manager, which decides to return to the ship.

D.11 T11: Handling freeze of echosounder driver

During T11, we verify the reaction of the robot to ECHOm_ALTI_1_1_1: *internal error triggers altitude content failure* (Fig A.4). We perform this test in the field while freezing the value of the echosounder. ECHOm_ALTI_1_1_1 detected as true when the output of the echosounder driver freezes during more than 2s. In that case, a local recovery (LR2) is initiated to lower the confidence of the measurement, so this measurement is not taken into account in the estimation.

During the run, the Fault Tolerance Module starts the detection of events at 5s, a necessary time to initialize the software and hardware components. At t=7.07s, ECHOm_ALTI_1_1_1 is detected as true and LR2 is initiated to decrease the confidence of the echosounder. However, at t=15.19s, ECHOm_ALTI_1_1 (*internal error triggers altitude content failure for a critical time*) is detected as true because the altitude output freezes during more than 10s. As a consequence, the ECHOm_ALTI_0 is deduced as true and ECHOm_ALTI_1_1_1 is diagnosed as true (Fig A.4). Since the altitude of the robot is measured by the echosounder and the DVL sensors, the redundancy provided by the operational DVL sensor prevents the deduction of further undesired events (Fig A.7).

Despite the loss of the echosounder, the transect can start normally at t=29.44s. The altitude estimation is only made by the DVL sensor during the transect.

D.12 T12: Handling motor2 out of order

In this section, we present the measurements of T12 both in simulation and in the field.

D.12.1 T12 in simulation

During T12, we verify the reaction of the robot to ACTd_PROP_1_1_11: *motor2 is KO* (Fig A.1). This test is first performed in simulation, on three transects at 3m depth. During the first transect, we do not inject faults (golden run). During the second transect, a fault is injected in the software that removes the command of motor2 to PWS1. The fault tolerance manager is not activated during the second transect. During the third transect, the same fault is injected, and we also activate the fault tolerance manager. During the transects, we log the depth, roll and pitch to identify and compare the resulting behavior of the robot. During the third transect, the fault tolerance manager detects that the current measured in PWS1 deviates from the expected current modeled (ACTd_PROP_1_1_8) at t=24.55s (Fig D.11).

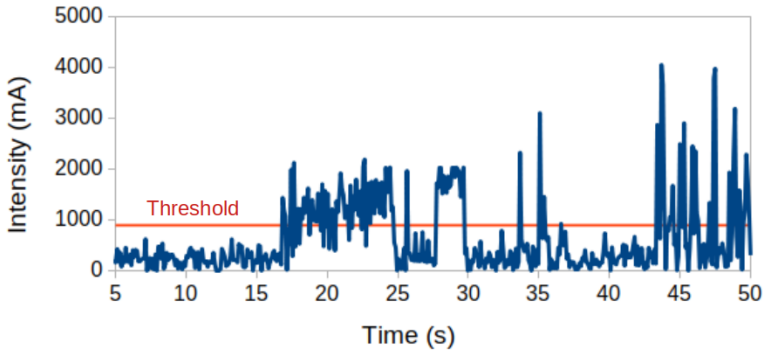


FIGURE D.11: Difference of intensity between the estimated and the measured current in PWS1

It is impossible to diagnose the faulty motor, as the current task is different from *test_actuators_task*. As a consequence, ACTd_PROP_1_1_6 (*The diagnosis of the failed motor cannot be performed*) is also detected as true. The FTM deduces then ACTd_PROP_0 as true and informs the mission manager. The latter chooses to execute *test_actuators_task* at t=24.63s. At t=33.96s, ACTd_PROP_1_1_8 is detected as true and LR1 is performed which consists in removing the failed motor from the actuator configuration matrix.

Fig D.12 illustrates the depth, roll and pitch of the robot during the transect in simulation. The errors are smaller with the FTM than without the FTM, and are even similar to the golden run errors.

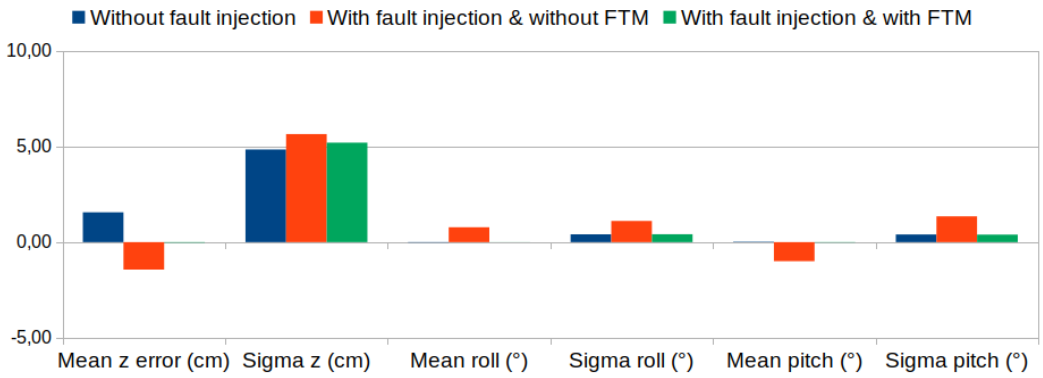


FIGURE D.12: Errors and standard deviations of depth, roll and pitch in simulation during T12

D.12.2 T12 in the field

We repeat the tests detailed in the previous section using the same protocols (3 transects including a golden run) but in the field. During the third transect, the fault tolerance manager detects ACTd_PROP_1_1_8 and ACTd_PROP_1_1_6 during MP0. The mission manager decides to execute *test_actuators_task*, then ACTd_PROP_1_1_8 is detected as true and LR1 is initiated as planned. We illustrate the depth, roll and pitch of the robot during the transect in the field Fig D.13.

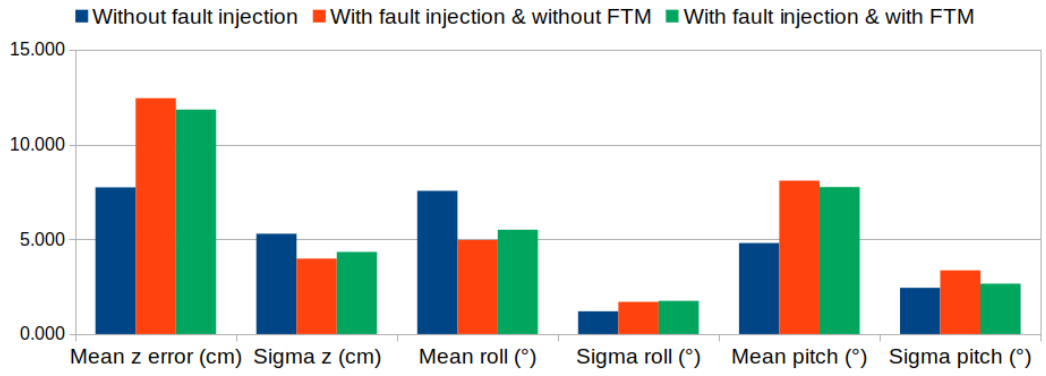


FIGURE D.13: Errors and standard deviations of depth, roll and pitch in the field during T12

Noting that the errors are more significant than in simulation, the introduction of the FTM has however the same effect as in simulation since it reduces the difference between the run errors and the golden run errors. Despite the weak improvement of the performance, disconnecting broken or jammed actuators has the advantage of not soliciting them, which could damage the robot or the motors by over-current.

D.13 T13: Handling motor2 and motor 6 out of order

During T13, we verify the reaction of the robot to ACTd_PROP_1_1_11 (*motor2 is KO*) and ACTd_PROP_1_1_15 (*motor6 is KO*) in the field (Fig A.1). We remove the command of motor2 and motor6 to PWS1 as injected faults.

During the transect, at t=34.09s, ACTd_PROP_1_1_9 (*One or several motors controlled by PWS2 are out of order*) and ACTd_PROP_1_1_6 (*The diagnosis of the failed motor cannot be performed*) are detected as true. ACTd_PROP_0 is deduced as true and the mission manager chooses to execute *test_actuators_task* at t=34.21s. At t=43.43s, both ACTd_PROP_1_1_11 and ACTd_PROP_1_1_15 are detected as true. These events are associated to the same local recovery (LR1), which is then performed once, since it is incompatible with itself. Thus, LR1 removes motor2 and motor6 from the actuator configuration matrix A. At t=43.63s, ACTd_PROP_1_1_5 (*The robot is under actuated*) is detected as true, since the new actuator configuration matrix A has a poor condition ($AA^T < 10^{-5}$). As a consequence, the FTM informs the mission manager that the robot is under-actuated. The mission manager decides to stop the motors (*stop_task*) at t=43.66s.

D.14 T14: Handling no communication with PC

During T14, we verify the reaction of the robot to UPBd_PC_2_2_1_1 (*connection with surface PC is interrupted for a warning time*) and UPBd_PC_2_2_1 (*connection with surface PC is interrupted for a critical time*) (Fig A.2). During this test, the robot performs a transect in the field. UPBd_PC_2_2_1_1 and UPBd_PC_2_2_1 are detected as true if there is no connection between the robot and the surface PC during more than respectively 4s and 10s.

During the transect, we unplug the ethernet cable between the surface PC and the robot around t=102s. UPBd_PC_2_2_1_1 is then detected at t=106.70s, and its associated local recovery LR10 restarts the *comm_module*. Since LR10 is ineffective to treat the injected fault, UPBd_PC_2_2_1 is detected as true at t=112.71s. The transect is then interrupted at t=112.81s by the mission manager, which instructs the robot to surface.

D.15 T15: Handling not enough energy to operate

During T15, we verify the reaction of the robot to ACTd_2_1_2: *the voltage of the batteries is too low* (Fig A.1). During this test, the robot performs a transect in simulation. We inject in the software the fault *Force the battery voltage under threshold* at t=50s. It consist in indicating to the FTM that the measured voltage is 11.19V whereas the threshold is set to 11.2V.

At t=50.05s, the fault tolerance manager indicates the events ACTd_2_1_2 and ACTd_PROP_0 are true. It informs the mission manager that a top event has been deduced and the causes of this event (ACTd_2_1_2). At t=50.08s, the transect is interrupted and the *stop_task* is launched by the mission manager. The robot finally surfaces with its natural buoyancy.

Bibliography

[1] M. Adachi et al. "An approach to optimization of fault tolerant architectures using HiP-HOPS". In: *Software: Practice and Experience* 41.11 (Oct. 2011), pp. 1303–1327. DOI: 10.1002/spe.1044.

[2] H. Ahn, WK. Loh, and WY. Yeo. "A Framework-Based Approach for Fault-Tolerant Service Robots". In: *International Journal of Advanced Robotic Systems* 9.5 (Nov. 1, 2012), p. 200. DOI: 10.5772/54023.

[3] B. Allotta et al. "A comparison between EKF-based and UKF-based navigation algorithms for AUVs localization". In: *OCEANS 2015 - Genova*. OCEANS 2015 - Genova. Genova, Italy: IEEE, May 2015, pp. 1–5. DOI: 10.1109/OCEANS-Genova.2015.7271681.

[4] M. Amy, J.-C. Fabre, and M. Lauer. "Towards Adaptive Fault Tolerance on ROS for Advanced Driver Assistance Systems". In: *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. Denver, CO, USA: IEEE, June 2017, pp. 29–35. DOI: 10.1109/DSN-W.2017.42.

[5] *ANSYS software*. URL: https://www.ansys.com/.

[6] *Arbre Analyst Software*. URL: https://www.arbre-analyste.fr/.

[7] J. Arlat. "Dependable computing and assessment of dependability". In: *CNRS, LAAS* (2011).

[8] J. Arlat et al. "Fault injection and dependability evaluation of fault-tolerant systems". In: *IEEE Transactions on Computers* 42.8 (Aug. 1993), pp. 913–923. DOI: 10.1109/12.238482.

[9] A Armoush. "Design Patterns for Safety-Critical Embedded Systems". PhD thesis. RWTH Aachen University, 2010.

[10] K. Aslansefat, G. Latif-Shabgahi, and M. Kamarlouei. "A Strategy for Reliability Evaluation and Fault Diagnosis of Autonomous Underwater Gliding Robot based on its Fault Tree". In: *ResearchGate* (2014).

[11] K. Aslansefat et al. "Dynamic Fault Tree Analysis". In: *Reliability Management and Engineering*. 1st ed. First edition. | Boca Raton, FL : CRC Press, 2020. |: CRC Press, June 15, 2020, pp. 73–112. DOI: 10.1201/9780429268922-4.

[12] T. Assaf and J.B. Dugan. "Automatic generation of diagnostic expert systems from fault trees". In: *Annual Reliability and Maintainability Symposium, 2003.* Annual Conference on Reliability and Maintainability (RAMS). Tampa, FL, USA: IEEE, 2003, pp. 143–147. DOI: 10.1109/RAMS.2003.1181916.

[13] T. Assaf and J.B. Dugan. "Diagnostic expert systems from dynamic fault trees". In: *Annual Symposium Reliability and Maintainability, 2004 - RAMS*. Los Angeles, CA, USA: IEEE, 2004, pp. 444–450. DOI: 10.1109/RAMS.2004.1285489.

[14] A. Avizienis et al. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), pp. 11–33.

[15] K. Bader, B. Lussier, and W. Schön. "A fault tolerant architecture for data fusion: A real application of Kalman filters for mobile robot localization". In: *Robotics and Autonomous Systems* 88 (Feb. 2017), pp. 11–23.

[16] M. Bajracharya, M W. Maimone, and D. Helmick. "Autonomy for Mars Rovers: Past, Present, and Future". In: *IEEE Computer Society* (2008).

[17] J-J. Borrelly. *The ORCCAD Architecture*. 1998.

[18] Ad Bowen, Dr Yoerger, and C Taylor. "The Nereus hybrid underwater robotic vehicle". In: *Underwater Technology* 28.3 (July 2, 2009), pp. 79–89.

[19] M. A Boyd. "Dynamic Fault-Tree Models for Fault-Tolerant Computer Systems". In: *IEEE Transactions on Reliability* 41.3 (1992).

[20] D. Bozhinoski et al. "Safety for mobile robotic systems: A systematic mapping study from a software engineering perspective". In: *Journal of Systems and Software* 151 (May 2019), pp. 150–179. DOI: 10.1016/j.jss.2019.02.021.

[21] M. P. Brito and G. Griffiths. "Updating Autonomous Underwater Vehicle Risk Based on the Effectiveness of Failure Prevention and Correction". In: *Journal of Atmospheric and Oceanic Technology* 35.4 (Apr. 1, 2018), pp. 797–808. DOI: 10.1175/JTECH-D-16-0252.1.

[22] RA Brooks. "New Approaches to Robotics". In: *Science* (1991).

[23] *Bubot project*. URL: https://www.lirmm.fr/bubot/.

[24] J. Carlson. "Analysis of how mobile robots fail in the field". PhD thesis. 2004.

[25] A. Ceballos. "GenoM as a Robotics Framework for Planetary Rover Surface Operations". In: *LAAS-CNRS* (2011).

[26] F. Chiacchio et al. "Dynamic fault trees resolution: A conscious trade-off between analytical and simulative approaches". In: *Reliability Engineering & System Safety* 96.11 (Nov. 2011), pp. 1515–1526. DOI: 10.1016/j.ress.2011.06.014.

[27] A. L. Christensen. "Fault Detection in Autonomous Robots". PhD thesis. Université Libre de Bruxelles, 2008.

[28] J. E. Cinner et al. "Bright spots among the world's coral reefs". In: *Nature* 535.7612 (July 2016), pp. 416–419.

[29] D. Crestani, K. Godary-Dejean, and L. Lapierre. "Enhancing fault tolerance of autonomous mobile robots". In: *Robotics and Autonomous Systems* 68 (June 2015), pp. 140–155.

[30] Y. Crouzet et al. "The SESAME Experience: from Assembly Languages to Declarative Models". In: *Second Workshop on Mutation Analysi*. Raleigh, NC, USA: IEEE, Nov. 2006. DOI: 10.1109/MUTATION.2006.14.

[31] Tho Dang-Hu. "Underwater robots for karst and marine exploration: a study of redundant AUVs". PhD thesis. Université de Montpellier, 2021.

[32] V. De Carolis et al. "Energy-aware fault-mitigation architecture for underwater vehicles". In: *Autonomous Robots* 41.5 (June 2017), pp. 1083–1105.

[33] B. Dieber et al. "Security for the Robot Operating System". In: *Robotics and Autonomous Systems* 98 (Dec. 2017), pp. 192–203. DOI: 10.1016/j.robot.2017.09.017.

[34] S. Dogramadzi et al. "Environmental Hazard Analysis - a Variant of Preliminary Hazard Analysis for Autonomous Mobile Robots". In: *Journal of Intelligent & Robotic Systems* 76.1 (Sept. 2014), pp. 73–117. DOI: 10.1007/s10846-013-0020-7.

[35] R-X. Duan and H-L. Zhou. "A New Fault Diagnosis Method Based on Fault Tree and Bayesian Networks". In: *Energy Procedia* 17 (2012), pp. 1376–1382. DOI: 10.1016/j.egypro.2012.02.255.

[36] E. Dubrova. "Time Redundancy". In: *Fault-Tolerant Design*. New York, NY: Springer New York, 2013, pp. 137–155. DOI: 10.1007/978-1-4614-2113-9_6.

[37] M. Dunbabin et al. "Real-time Vision-only Perception for Robotic Coral Reef Monitoring and Management". In: *Queensland University of Technology* (2019).

[38] J. Dunjó et al. "Hazard and operability (HAZOP) analysis. A literature review". In: *Journal of Hazardous Materials* 173.1 (Jan. 15, 2010), pp. 19–32. DOI: 10.1016/j.jhazmat.2009.08.076.

[39] B. Durand. "Proposition d'une architecture de contrôle adaptative pour la tolérance aux fautes". PhD thesis. Université de Montpellier, 2011.

[40] K. Durga Rao et al. "Dynamic fault tree analysis using Monte Carlo simulation in probabilistic safety assessment". In: *Reliability Engineering & System Safety* 94.4 (Apr. 2009), pp. 872–883. DOI: 10.1016/j.ress.2008.09.007.

[41] *Eigen C++ library*. URL: https://eigen.tuxfamily.org.

[42] William Excoffon, Jean-Charles Fabre, and Michael Lauer. "Analysis of Adaptive Fault Tolerance for Resilient Computing". In: *2017 13th European Dependable Computing Conference (EDCC)*. Geneva: IEEE, Sept. 2017, pp. 50–57. DOI: 10.1109/EDCC.2017.22.

[43] A. Favier et al. "A hierarchical fault tolerant architecture for an autonomous robot". In: *International Conference on Dependable Systems and Networks Workshops*. Valencia, Spain, June 2020.

[44] P. Feiler and J. Delange. "Automated Fault Tree Analysis from AADL Models". In: *ACM SIGAda Ada Letters* 36.2 (May 10, 2017), pp. 39–46. DOI: 10.1145/3092893.3092900.

[45] R. Ferdous et al. "Methodology for computer aided fuzzy fault tree analysis". In: *Process Safety and Environmental Protection* 87.4 (July 2009), pp. 217–226. DOI: 10.1016/j.psep.2009.04.004.

[46] R. Filippini, S. Sen, and A. Bicchi. "A Study of Antagonistic Actuation". In: *IEEE Robotics & Automation Magazine* (Sept. 2008). DOI: 10.1109/MRA.2008.927696.

[47] T. I. Fossen. *Handbook of Marine Craft Hydrodynamics and Motion Control*. Chichester, UK: John Wiley & Sons, Ltd, Apr. 8, 2011.

[48] T. O. Fossum, M. Ludvigsen, and S. M. Nornes. "Autonomous robotic intervention using ROV: An experimental approach". In: *OCEANS 2016 MTS/IEEE Monterey*. Monterey, CA, USA: IEEE, Sept. 2016.

[49] Z. Gao, C. Cecati, and S. Ding. "A Survey of Fault Diagnosis and Fault-Tolerant Techniques Part II: Fault Diagnosis with Knowledge-Based and Hybrid/Active Approaches". In: *IEEE Transactions on Industrial Electronics* (2015), pp. 3768–3774. DOI: 10.1109/TIE.2015.2419013.

[50] Z. Gao, C. Cecati, and S. X. Ding. "A Survey of Fault Diagnosis and Fault-Tolerant Techniques—Part I: Fault Diagnosis With Model-Based and Signal-Based Approaches". In: *IEEE Transactions on Industrial Electronics* 62.6 (June 2015), pp. 3757–3767. DOI: 10.1109/TIE.2015.2417501.

[51] E. Gat. "On Three-Layer Architectures". In: *MIT press* (1998), pp. 195–210.

[52] *Gazebo simulator*. URL: http://gazebosim.org.

[53] G. Griffiths, N. W. Millard, and S. D. McPhail. "On the Reliability of the Autosub Autonomous Underwater Vehicle". In: *Underwater Technology* 25.4 (Dec. 1, 2003), pp. 175–184.

[54] I. Grosclaude. "Model-based monitoring of component-based software systems". In: France Telecom R&D (2004).

[55] J. Guiochet. "Maitrise de la sécurité des systèmes de la robotique de service". PhD thesis. 2003.

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

[57] P. Haapanen and A. Helminen. "Failure mode and effects analysis of software-based automation systems". In: STUK (2002).

[58] S. Haddadin, A. De Luca, and A. Albu-Schaffer. "Robot Collisions: A Survey on Detection, Isolation, and Identification". In: *IEEE Transactions on Robotics* 33.6 (Dec. 2017), pp. 1292–1312. DOI: 10.1109/TRO.2017.2723903.

[59] B. Halder and N. Sarkar. "Robust nonlinear analytic redundancy for fault detection and isolation in mobile robot". In: *International Journal of Automation and Computing* (2007).

[60] D. L. Hamilton, I. D Walker, and J. K Bennett. "Fault Tolerance versus Performance Metrics for Robot Systems". In: *Proceedings of IEEE International Conference on Robotics and Automation*. 1996, pp. 3073–3080. DOI: 10.1109/ROBOT.1996.509179.

[61] S. Hasan et al. "A Fault-Tolerant Controller Manager for Platooning Simulation". In: *2019 IEEE International Conference on Connected Vehicles and Expo (ICCVE)*. Graz, Austria: IEEE, Nov. 2019. DOI: 10.1109/ICCVE45908.2019.8965220.

[62] O. Hegrenaes et al. "Validation of a new generation DVL for underwater vehicle navigation". In: *2016 IEEE/OES Autonomous Underwater Vehicles (AUV)*. Tokyo, Japan: IEEE, Nov. 2016, pp. 342–348.

[63] E. H. Henriksen, I. Schjolberg, and T. B. Gjersvik. "UW MORSE: The underwater Modular Open Robot Simulation Engine". In: *2016 IEEE/OES Autonomous Underwater Vehicles (AUV)*. Tokyo, Japan: IEEE, Nov. 2016, pp. 261–267. DOI: 10.1109/AUV.2016.7778681.

[64] A. Hereau et al. "A Fault Tolerant Control Architecture Based on Fault Trees for an Underwater Robot Executing Transect Missions". In: *International Conference on Robotics and Automation (ICRA)* (2021).

[65] A. Hereau et al. "Testing an Underwater Robot Executing Transect Missions in Mayotte". In: *21st Towards Autonomous Robotic Systems Conference*. Nottingham, United Kingdom, Sept. 2020.

[66] M. Hofbaur et al. "Improving Robustness of Mobile Robots Using Model-based Reasoning". In: *J Intell Robot Syst* (2007).

[67] C. Huet and F. Mastroddi. "Autonomy for underwater robots—a European perspective". In: *Autonomous Robots* 40.7 (Oct. 2016), pp. 1113–1118. DOI: 10.1007/s10514-016-9605-x.

[68] E. E. Hurdle, L. M. Bartlett, and J. D. Andrews. "System fault diagnostics using fault tree analysis". In: *IMechE* (2007).

[69] N. Ibrahim and R. Ibrahim. "Semantic Rules of UML Specification". In: *Proceedings of MUCEET* (2009), pp. 37–40.

[70] J. Ingibergsson, U. P. Schultz, and D. Kraft. "Towards Declarative Safety Rules for Perception Specification Architectures". In: *University of Southern Denmark* (Jan. 12, 2016).

[71] C. C. Insaurralde and Y. R. Petillot. "Capability-oriented robot architecture for maritime autonomy". In: *Robotics and Autonomous Systems* 67 (May 2015), pp. 87–104. DOI: 10.1016/j.robot.2014.10.003.

[72] A. Inzartsev et al. "Application of artificial intelligence techniques for fault diagnostics of autonomous underwater vehicles". In: *OCEANS MTS/IEEE Monterey*. Monterey, CA, USA: IEEE, Sept. 2016. DOI: 10.1109/OCEANS.2016.7761098.

[73] R. Isermann. *Fault-Diagnosis Systems: An Introduction from Fault Detection to Fault Tolerance*. Springer. 2006.

[74] R Isermann and P Ball. "Trends in the application of model-based fault detection and diagnosis of technical processes". In: *Control Eng. Practice*. Elsevier 5.5 (1997), pp. 709–719.

[75] T. Ishigooka et al. "Graceful Degradation Design Process for Autonomous Driving System". In: *Computer Safety, Reliability, and Security*. Vol. 11698. Cham: Springer International Publishing, 2019, pp. 19–34. DOI: 10.1007/978-3-030-26601-1_2.

[76] M. Jacobi and T. Rauschenbach. "A tool chain for AUV system testing". In: *OCEANS'10 IEEE SYDNEY*. Sydney, Australia: IEEE, May 2010. DOI: 10.1109/OCEANSSYD.2010.5603626.

[77] A. Jacoff et al. "Development of standard test methods for evaluation of ROV/AUV performance for emergency response applications". In: *OCEANS 2015 - MTS/IEEE Washington*. Washington, DC: IEEE, Oct. 2015, pp. 1–10.

[78] L. Jaiem et al. "A Step Toward Mobile Robots Autonomy: Energy Estimation Models". In: *Towards Autonomous Robotic Systems*. Vol. 9716. Cham: Springer International Publishing, 2016, pp. 177–188.

[79] A. T. James, O. P. Gandhi, and S. G. Deshmukh. "Fault diagnosis of automobile systems using fault tree based on digraph modeling". In: *International Journal of System Assurance Engineering and Management* 9.2 (Apr. 2018), pp. 494–508. DOI: 10.1007/s13198-017-0693-6.

[80] L. Jaulin. *Mobile robotics*. Wiley. 2019.

[81] C. P. Jones. "Slocum glider persistent oceanography". In: *2012 IEEE/OES Autonomous Underwater Vehicles (AUV)*. Southampton, United Kingdom: IEEE, Sept. 2012, pp. 1–6. DOI: 10.1109/AUV.2012.6380738.

[82] E. Khalastchi and M. Kalesh. "A sensor-based approach for fault detection and diagnosis for robotic systems". In: *Autonomous Robots* 42 (2018), pp. 1231–1248.

[83] Z. Kira. "Modeling Cross-Sensory and Sensorimotor Correlations to Detect and Localize Faults in Mobile Robots". In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems* (2007), p. 7.

[84] M. F. Kjell, F. Gustafsson, and O. Hegrenaes. "Estimation of AUV dynamics for sensor fusion". In: *10th International Conference on Information Fusion*. Quebec City, QC, Canada: IEEE, 2007. DOI: 10.1109/ICIF.2007.4408044.

[85] P. Klein. "The Safety-Bag Expert System in the Electronic Railway Interlocking System Elektra". In: *Expert Systems WithApplications* 3 (1991), pp. 499–506.

[86] G. K. Kraetzschmar. *Best Practice in Robotics*. Aug. 2010.

[87] A. Laidani et al. "A Low-Cost Test Bench for Underwater Thruster Identification". In: *IFAC-PapersOnLine* 52.21 (2019), pp. 254–259. DOI: 10.1016/j.ifacol.2019.12.316.

[88] P. Lambert. "Contribution à l'autonomie des robots: vers des missions autonomes à garantie de performance incluant l'incertitude de localisation en environnement interieur connu". PhD thesis. Université de Montpellier, 2021.

[89] P. Lambert, L. Lapierre, and D. Crestani. "An Approach for Fault Tolerant and Performance Guarantee Autonomous Robotic Mission". In: *2019 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. Colchester, United Kingdom: IEEE, July 2019, pp. 87–94.

[90] M. Lauer et al. "Resilient computing on ROS using adaptive fault tolerance: Resilient computing on ROS using adaptive fault tolerance". In: *Journal of Software: Evolution and Process* 30.3 (Mar. 2018), e1917. DOI: 10.1002/smr.1917.

[91] N. R.J. Lawrance et al. "Ocean deployment and testing of a semi-autonomous underwater vehicle". In: *OCEANS 2016 MTS/IEEE Monterey*. Monterey, CA, USA: IEEE, Sept. 2016.

[92] A. Leite and A. Pinto. "A Safety Monitoring Model for a Faulty Mobile Robot". In: *Robotics* 7.3 (June 21, 2018). DOI: 10.3390/robotics7030032.

[93] P. Lepej et al. "An integrated Diagnosis and Repair Architecture for ROS-Based Robot Systems". In: *23rd International Workshop on Principles of Diagnosis* (2012).

[94] N. Leveson. *Engineering a safer world: systems thinking applied to safety*. Engineering systems. Cambridge, Mass: MIT Press, 2011.

[95] N. Leveson. *STPA Handbook*. Mar. 2018.

[96] S. Louis. "Système robotisé semi-autonome pour l'observation des espèces marines". PhD thesis. Université de Montpellier, 2018.

[97] B. Lussier. "Tolérance aux fautes dans les systèmes autonomes". PhD thesis. 2007.

[98] B. Lussier et al. "Planning with Diversified Models for Fault-Tolerant Robots". In: *International Conference on Automated Planning and Scheduling (ICAPS)*. Providence, RI, United States, Sept. 2007, pp. 216–223.

[99] H-J. Ma and G-H. Yang. "Simultaneous fault diagnosis for robot manipulators with actuator and sensor faults". In: *Information Sciences* 366 (Oct. 2016), pp. 12–30. DOI: 10.1016/j.ins.2016.05.016.

[100] A. Mahmood et al. "Automatic annotation of coral reefs using deep learning". In: *OCEANS 2016 MTS/IEEE Monterey*. Monterey, CA, USA: IEEE, Sept. 2016, pp. 1–5. DOI: 10.1109/OCEANS.2016.7761105.

[101] Y. A. Mahmood et al. "Fuzzy fault tree analysis: a review of concept and application". In: *International Journal of System Assurance Engineering and Management* 4.1 (Mar. 2013), pp. 19–32. DOI: 10.1007/s13198-013-0145-x.

[102] D. Martin-Guillerez et al. "A UML-based method for risk analysis of human-robot interactions". In: *Proceedings of the 2nd International Workshop on Software Engineering for Resilient Systems - SERENCE '10*. London, United Kingdom: ACM Press, 2010, pp. 32–41. DOI: 10.1145/2401736.2401740.

[103] M. Maslin et al. "Underwater robots provide similar fish biodiversity assessments as divers on coral reefs". In: *Remote Sensing in Ecology and Conservation* 7.4 (Dec. 2021). Ed. by Nathalie Pettorelli and Vincent Lecours, pp. 567–578. DOI: 10.1002/rse2.209.

[104] F. Mazzini and S. Dubowsky. "An Experimental Validation of Robotic Tactile Mapping in Harsh Environments such as Deep Sea Oil Well Sites". In: *Experimental Robotics*. Vol. 79. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 557–570. DOI: 10.1007/978-3-642-28572-1_38.

[105] G. McCall. *MISRA-C: 2004 - Guidelines for the use of the C language in critical systems*. MIRA Limited, 2004.

[106] A. Mesa et al. "A web-based system for marine fishes mapping and assessment". In: *University of the Philippines Mindanao* (2016).

[107] Z. Mian et al. "System Dependability Modelling and Analysis Using AADL and HiP-HOPS". In: *IFAC Proceedings Volumes* 45.6 (May 2012), pp. 1647–1652. DOI: 10.3182/20120523-3-RO-2023.00334.

[108] Z. Micskei, Z. Szatmari, and I. Majzik. "A Concept for Testing Robustness and Safety of the Context-Aware Behaviour of Autonomous Systems". In: *Agent and Multi-Agent Systems. Technologies and Applications*. Vol. 7327. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 504–513.

[109] M. Modasshir et al. "Coral Identification and Counting with an Autonomous Underwater Vehicle". In: *2018 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. Kuala Lumpur, Malaysia: IEEE, Dec. 2018, pp. 524–529. DOI: 10.1109/ROBIO.2018.8664785.

[110] R. Moraes, H. Waeselynck, and J. Guiochet. "UML-Based Modeling of Robustness Testing". In: *IEEE 15th International Symposium on High-Assurance Systems Engineering*. Miami Beach, FL, USA: IEEE, Jan. 2014, pp. 168–175. DOI: 10.1109/HASE.2014.31.

[111] T. Morita and S. Sugano. *Double Safety Measure for Human Symbiotic Manipulator*. Waseda University, 1997.

[112] Y. Mortureux. "Arbres de défaillance, des causes et d'événement". In: *Techniques de l'Ingénieur* (2002).

[113] M. Murad et al. "Surveillance of coral reef development using an Autonomous Underwater Vehicle". In: *IEEE/OES Autonomous Underwater Vehicles (AUV)*. Tokyo, Japan: IEEE, Nov. 2016, pp. 14–19. DOI: 10.1109/AUV.2016.7778713.

[114] N. Muscettola et al. "Remote Agent: to boldly go where no AI system has gone before". In: *Artificial Intelligence* 103.1 (Aug. 1998), pp. 5–47. DOI: 10.1016/S0004-3702(98)00068-X.

[115] I. Nagelkerken and Philip L. Munday. "Animal behaviour shapes the ecological effects of ocean acidification and warming: moving from individual to community-level responses". In: *Global Change Biology* 22.3 (Mar. 2016), pp. 974–989.

[116] L. Nana et al. "Investigation of safety mechanisms for oceanographic AUV missions programming". In: *Europe Oceans 2005*. Oceans 2005 - Europe. Brest, France: IEEE, 2005, 906–913 Vol. 2. DOI: 10.1109/OCEANSE.2005.1513177.

[117] S. Negahdaripour and P. Firoozfam. "An ROV Stereovision System for Ship-Hull Inspection". In: *IEEE Journal of Oceanic Engineering* 31.3 (2006), pp. 551–564. DOI: 10.1109/JOE.2005.851391.

[118] L. Negash, H-Y. Kim, and H-L. Choi. "Emerging UAV Applications in Agriculture". In: *2019 7th International Conference on Robot Intelligence Technology and Applications (RiTA)*. Daejeon, Korea (South): IEEE, Nov. 2019, pp. 254–257. DOI: 10.1109/RITAPP.2019.8932853.

[119] Ulrich Nehmzow. *Scientific methods in mobile robotics: quantitative analysis of agent behaviour*. OCLC: ocm61441340. London: Springer, 2006.

[120] I. A. Nesnas, A. Wright, and M. Bajracharya. "CLARAty: an architecture for reusable robotic software". In: AeroSense 2003. Ed. by G. R. Gerhart and C. M. Shoemaker. Orlando, FL, Sept. 26, 2003, p. 253. DOI: 10.1117/12.497223.

[121] O. Nusbaumer and A. Rauzy. "Fault tree linking versus event tree linking approaches: a reasoned comparison". In: *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability* 227.3 (June 2013), pp. 315–326. DOI: 10.1177/1748006X13490260.

[122] E. Omerdic. "Smart ROV LATIS: From Design Concepts to Test Trials". In: *Manoeuvring and Control of Marine Craft* (2009).

[123] *Orocos framework*. URL: https://orocos.org.

[124] N. Palomeras et al. "Toward persistent autonomous intervention in a subsea panel". In: *Autonomous Robots* 40.7 (Oct. 2016), pp. 1279–1306. DOI: 10.1007/s10514-015-9511-7.

[125] L. E. Parker and B. Kannan. "Adaptive Causal Models for Fault Diagnosis and Recovery in Multi-Robot Teams". In: *International Conference on Intelligent Robots and Systems*. Beijing, China, 2006, pp. 2703–2710.

[126] R. Passama. *ContrACT: une méthodologie de conception et de développement d'architectures de contrôle de robots*. LIRMM, 2010.

[127] M. E. Pate-Cornell. "Fault Trees vs. Event Trees in Reliability Analysis". In: *Risk Analysis* 4.3 (Sept. 1984), pp. 177–186. DOI: 10.1111/j.1539-6924.1984.tb00137.x.

[128] S. Pathak et al. "Ensuring safety of policies learned by reinforcement: Reaching objects in the presence of obstacles with the iCub". In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Tokyo: IEEE, Nov. 2013, pp. 170–175. DOI: 10.1109/IROS.2013.6696349.

[129] R.J. Patton and J. Chen. "A Review of Parity Space Approaches to Fault Diagnosis". In: *IFAC Proceedings Volumes* 24.6 (Sept. 1991), pp. 65–81. DOI: 10.1016/S1474-6670(17)51124-6.

[130] *PID framework*. URL: https://pid.lirmm.net/pid-framework/.

[131] D. Powell et al. "Estimators for fault tolerance coverage evaluation". In: *IEEE Transactions on Computers* 44.2 (Feb. 1995), pp. 261–274. DOI: 10.1109/12.364537.

[132] F. Py and F. Ingrand. "Real-Time Execution Control for Autonomous Systems". In: *2nd Embedded Real Time Software Congress* (2004).

[133] M. Quann et al. "Ground Robot Terrain Mapping and Energy Prediction in Environments with 3-D Topography". In: *2018 Annual American Control Conference (ACC)*. Milwaukee, WI: IEEE, June 2018, pp. 3532–3537. DOI: 10.23919/ACC.2018.8430767.

[134] M. Ranjbaran and K. Khorasani. "Fault recovery of an under-actuated quadrotor Aerial Vehicle". In: *49th IEEE Conference on Decision and Control (CDC)*. Atlanta, GA, USA: IEEE, Dec. 2010, pp. 4385–4392.

[135] I. Rist-Christensen. "Autonomous Robotic Intervention using ROV". PhD thesis. 2016.

[136] C. Robert. "Génération et Analyse de tests pour les systèmes autonomes". PhD thesis. 2021.

[137] C. Robert et al. "The virtual lands of Oz: testing an agribot in simulation". In: *Empirical Software Engineering Journal* (Mar. 2020).

[138] B. Ropars et al. "Redundant actuation system of an underwater vehicle". In: *Ocean Engineering* 151 (Mar. 2018), pp. 276–289.

[139] *ROS middleware*. URL: www.ros.org.

[140] S. I. Roumeliotis, G. S. Sukhatme, and G. A Bekey. "Sensor Fault Detection and Identi cation in a Mobile Robot". In: *Proceedings. IEEE/RSJ International Conference on Intelligent Robots and Systems. Innovations in Theory, Practice and Applications* (1998).

[141] D. L. Rudnick, R. E. Davis, and J. T. Sherman. "Spray Underwater Glider Operations". In: *Journal of Atmospheric and Oceanic Technology* 33.6 (June 1, 2016), pp. 1113–1122.

[142] M.G. Ryschkewitsch. *Fault Management Handbook*. 2012.

[143] M. Saied et al. "Fault diagnosis and fault-tolerant control strategy for rotor failure in an octorotor". In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. Seattle, WA, USA: IEEE, May 2015, pp. 5266–5271.

[144] M. A. Samoilys and G. Carlos. "Determining methods of underwater visual census for estimating the abundance of coral reef fishes". In: *Environmental Biology of Fishes* 57.3 (2000), pp. 289–304.

[145] M. Scheutz and J. Kramer. "Reflection and Reasoning Mechanisms for Failure Detection and Recovery in a Distributed Robotic Architecture for Complex Robots". In: *Proceedings 2007 IEEE International Conference on Robotics and Automation*. ISSN: 1050-4729. Rome, Italy: IEEE, Apr. 2007, pp. 3699–3704. DOI: 10.1109/ROBOT.2007.364045.

[146] D. Selčan, G. Kirbiš, and I. Kramberger. "Nanosatellites in LEO and beyond: Advanced Radiation protection techniques for COTS-based spacecraft". In: *Acta Astronautica* 131 (Feb. 2017), pp. 131–144. DOI: 10.1016/j.actaastro.2016.11.032.

[147] S. Shahamiri, N. Kadir, and S. Z. Mohd-Hashim. "A Comparative Study on Automated Software Test Oracle Methods". In: *2009 Fourth International Conference on Software Engineering Advances*. Porto, Portugal: IEEE, Sept. 2009, pp. 140–145.

[148] A. Shakhimardanov. *Best Practice in Robotics*. Mar. 2010.

[149] A. Shakhimardanov, N. Hochgeschwender, and G. K. Kraetzschmar. "Component models in robotics software". In: *Proceedings of the 10th Performance Metrics for Intelligent Systems*. Baltimore, Maryland: ACM Press, 2010, pp. 82–87. DOI: 10.1145/2377576.2377592.

[150] S. N. Shome et al. "Development of Modular Shallow Water AUV: Issues & Trial Results". In: *Journal of The Institution of Engineers (India): Series C* 93.3 (July 2012), pp. 217–228.

[151] J. Siegmund, N. Siegmund, and S. Apel. "Views on Internal and External Validity in Empirical Software Engineering". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Florence, Italy: IEEE, May 2015, pp. 9–19.

[152] B Smith. *Remote Agent Experiment*. Deep Space 1 Technology Validation Report. 1998.

[153] T. Sotiropoulos et al. "Can Robot Navigation Bugs Be Found in Simulation? An Exploratory Study". In: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. Prague, Czech Republic: IEEE, July 2017, pp. 150–159.

[154] M Stamatelatos et al. *Fault Tree Handbook with Aerospace Applications*. NASA. 2002.

[155] D. Stavrou et al. "Fault detection for service mobile robots using model-based method". In: *Autonomous Robots* 40.2 (Feb. 2016), pp. 383–394.

[156] D. Sward, J. Monk, and N. Barrett. "A Systematic Review of Remotely Operated Vehicle Surveys for Visually Assessing Fish Assemblages". In: *Frontiers in Marine Science* 6 (Apr. 3, 2019). DOI: 10.3389/fmars.2019.00134.

[157] P. Thalamy, B. Piranda, and J. Bourgeois. "A survey of autonomous self-reconfiguration methods for robot-based programmable matter". In: *Robotics and Autonomous Systems* 120 (Oct. 2019). DOI: 10.1016/j.robot.2019.07.012.

[158] U. Thomas et al. "A new skill based robot programming language using UML/P Statecharts". In: *2013 IEEE International Conference on Robotics and Automation*. Karlsruhe, Germany: IEEE, May 2013, pp. 461–466. DOI: 10.1109/ICRA.2013.6630615.

[159] R. Thurstan et al. "Survey Methods Manual Unidive Point Lookout Ecological Assessment (PLEA)". University of Queensland Underwater Club, Brisbane, 2014.

[160] N. Tomatis et al. "Designing a secure and robust mobile interacting robot for the long term". In: *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*. IEEE International Conference on Robotics and Automation. IEEE ICRA 2003 Conference Proceedings. Vol. 3. Taipei, Taiwan: IEEE, 2003, pp. 4246–4251. DOI: 10.1109/ROBOT.2003.1242256.

[161] H. Täubig et al. "Guaranteeing functional safety: design for provability and computer-aided verification". In: *Autonomous Robots* 32.3 (Apr. 2012), pp. 303–331. DOI: 10.1007/s10514-011-9271-y.

[162] I. Ullah et al. "Evaluation of Localization by Extended Kalman Filter, Unscented Kalman Filter, and Particle Filter-Based Techniques". In: *Wireless Communications and Mobile Computing* 2020 (Oct. 2, 2020). Ed. by C. Luo, pp. 1–15. DOI: 10.1155/2020/8898672.

[163] U. K. Verfuss, A. S. Aniceto, and M. Biuw. *Literature review: understanding the current state of autonomous technologies to improve/expand observation and detection of marine species*. July 2016.

[164] V. Verma and R. Simmons. "Scalable robot fault detection and identification". In: *Robotics and Autonomous Systems* 54.2 (Feb. 2006), pp. 184–191. DOI: 10.1016/j.robot.2005.09.028.

[165] V. Verma et al. "Particle Filters for Rover Fault Diagnosis". In: *Robotics & Automation Magazine* (2004), p. 13.

[166] J-L. Vilchis-Medina, K. Godary-Dejean, and C. Lesire. "Autonomous Decision-Making With Incomplete Information and Safety Rules Based on Non - Monotonic Reasoning". In: *IEEE Robotics and Automation Letters* 6.4 (Oct. 2021), pp. 8357–8362. DOI: 10.1109/LRA.2021.3103048.

[167] M.L. Visinsky, J.R. Cavallaro, and I.D. Walker. "Robotic fault detection and fault tolerance: A survey". In: *Reliability Engineering & System Safety* 46.2 (Jan. 1994), pp. 139–158. DOI: 10.1016/0951-8320(94)90132-5.

[168] T. Wang et al. "Fuzzy Reliability Design of Robot Parts Based on Weibull and Normal Distribution". In: *2008 International Conference on Computer Science and Software Engineering*. Wuhan, China: IEEE, 2008, pp. 1081–1084. DOI: 10.1109/CSSE.2008.430.

[169] L. Wen et al. "A New Convolutional Neural Network-Based Data-Driven Fault Diagnosis Method". In: *IEEE Transactions on Industrial Electronics* 65.7 (July 2018), pp. 5990–5998. DOI: 10.1109/TIE.2017.2774777.

[170] Johannes Wienke and Sebastian Wrede. "Autonomous Fault Detection for Performance Bugs in Component-Based Robotic Systems". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2016).

[171] S. Williams, O. Pizarro, and M. Jakuba. "Monitoring of Benthic Reference Sites: Using an Autonomous Underwater Vehicle". In: *IEEE Robotics & Automation Magazine* 19.1 (Mar. 2012), pp. 73–84.

[172] C. Wong, E. Yang, and X-T. Yan. "Autonomous robots for harsh environments: a holistic overview of current solutions and ongoing challenges". In: *Systems Science & Control Engineering* 6 (2018), pp. 213–219.

[173] O. Wongwirat, T. Hanidthikul, and N. Vuthikulvanich. "A formal approach in robot development process using a UML model". In: *2008 10th International Conference on Control, Automation, Robotics and Vision*. Hanoi, Vietnam: IEEE, Dec. 2008, pp. 1888–1893. DOI: 10.1109/ICARCV.2008.4795816.

[174] R. Woodman et al. "Building safer robots: Safety driven control". In: *The International Journal of Robotics Research* 31.13 (Nov. 2012), pp. 1603–1626. DOI: 10.1177/0278364912459665.

[175] X. Xiang, C. Yu, and Q. Zhang. "On intelligent risk analysis and critical decision of underwater robotic vehicle". In: *Ocean Engineering* 140 (Aug. 2017), pp. 453–465. DOI: 10.1016/j.oceaneng.2017.06.020.

[176] B. Xinqian et al. "Simulation model and Fault Tree Analysis for AUV". In: *International Conference on Mechatronics and Automation*. Changchun, China: IEEE, Aug. 2009, pp. 4452–4457. DOI: 10.1109/ICMA.2009.5246716.

[177] H. Xu, G. Li, and J. Liu. "Reliability analysis of an autonomous underwater vehicle using fault tree". In: *2013 IEEE International Conference on Information and Automation (ICIA)*. Yinchuan, China: IEEE, Aug. 2013, pp. 1165–1170. DOI: 10.1109/ICInfA.2013.6720471.

[178] W. Xue, Y-Q. Guo, and X-D. Zhang. "A Bank of Kalman Filters and a Robust Kalman Filter Applied in Fault Diagnosis of Aircraft Engine Sensor/Actuator". In: *Second International Conference on Innovative Computing, Informatio and Control (ICICIC 2007)*. Kumamoto, Japan: IEEE, Sept. 2007. DOI: 10.1109/ICICIC.2007.3.

[179] Y. C. Yeh. "Triple-Triple Redundant 777 Primary Flight Computer". In: *IEEE Aerospace Applications Conference, Proceedings* (1996).

[180] M. Zelkowitz. *Advances in Computers: Advances in Software Engineering*. Elsevier. Vol. 62. 2004.

[181] M. Zhang et al. "Thruster fault identification method for autonomous underwater vehicle using peak region energy and least square grey relational grade". In: *Advances in Mechanical Engineering* 7.12 (Dec. 2015). DOI: 10.1177/1687814015622905.

[182] T. Zhang. "Resilient Robots: Concept, Review, and Future Directions". In: *Robotics* 6.4 (Sept. 25, 2017). DOI: 10.3390/robotics6040022.

[183] Y. Zhang and J. Jiang. "Bibliographical review on reconfigurable fault-tolerant control systems". In: *Annual Reviews in Control* 32.2 (Dec. 2008), pp. 229–252. DOI: 10.1016/j.arcontrol.2008.03.008.

[184] A. Zolghadri. "Advanced model-based FDIR techniques for aerospace systems: Today challenges and opportunities". In: *Progress in Aerospace Sciences* 53 (Aug. 2012), pp. 18–29.