Float for multidisciplinary monitoring of the marine environment. From business expertise to embedded codes

Sébastien Bonnieux

To cite this version:

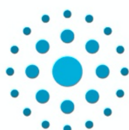
Sébastien Bonnieux. Float for multidisciplinary monitoring of the marine environment. From business expertise to embedded codes. Earth Sciences. Université Côte d'Azur, 2020. English. NNT : 2020COAZ4072 . tel-03197915

HAL Id: tel-03197915
https://theses.hal.science/tel-03197915v1

Submitted on 14 Apr 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ CÔTE D'AZUR 

ÉCOLE DOCTORALE
SCIENCES FONDAMENTALES ET APPLIQUÉES

THÈSE DE DOCTORAT

Flotteur pour la surveillance pluridisciplinaire de l'environnement marin : de l'expertise métier aux codes embarqués

Sébastien BONNIEUX

Géoazur

Présentée en vue de l'obtention
du grade de docteur en Sciences de la Planète et de l'Univers
de l'Université Côte d'Azur
Dirigée par : Frédéric Cappa
Co-encadrée par : Guust Nolet
Soutenue le : 14/12/2020

Devant le jury, composé de :
Mireille Blay-Fornarino, Professeure, Université Côte d'Azur
Dorian Cazau, Maitre de conférence, ENSTA[1] Bretagne
Frédéric Cappa, Professeur, Université Côte d'Azur
Mireille Laigle, Directrice de recherche, CNRS[2]
Guust Nolet, Professeur, Université Côte d'Azur
Jean-Yves Royer, Directeur de recherche, CNRS[2]
Karin Sigloch, Professeure, Oxford University
Frederik Simons, Professeur, Princeton University

Invités :
Yann Hello, Ingénieur, IRD[3]
Olivier Philippe, Ingénieur, OSEAN S.A.S

1. École Nationale Supérieure de Techniques Avancées
2. Centre National de la Recherche Scientifique
3. Institut de Recherche pour le Développement

RÉGION SUD PROVENCE ALPES CÔTE D'AZUR

Flotteur pour la surveillance pluridisciplinaire de l'environnement marin : de l'expertise métier aux codes embarqués

Thèse présentée devant le jury composé de :

Rapporteurs :

Karin Sigloch, Professeure, Oxford University
Frederik Simons, Professeur, Princeton University

Examinateurs :

Mireille Blay-Fornarino, Professeure, Université Côte d'Azur
Dorian Cazau, Maitre de conférence, ENSTA [1] Bretagne
Mireille Laigle, Directrice de recherche, CNRS [2]
Guust Nolet, Professeur, Université Côte d'Azur
Jean-Yves Royer, Directeur de recherche, CNRS[2]

Directeur de thèse :

Frédéric Cappa, Professeur, Université Côte d'Azur

Invités :

Yann Hello, Ingénieur, IRD [3]
Olivier Philippe, Ingénieur, OSEAN S.A.S

1. École Nationale Supérieure de Techniques Avancées
2. Centre National de la Recherche Scientifique
3. Institut de Recherche pour le Développement

Flotteur pour la surveillance pluridisciplinaire de l'environnement marin : de l'expertise métier aux codes embarqués

Dans le cadre d'un projet ERC (European Research Council) mené à Geoazur de 2009 à 2015 par Guust Nolet, un flotteur profileur autonome équipé d'un hydrophone et pouvant accueillir jusqu'à 8 capteurs a été développé. Il vise à acquérir des données en zones océaniques, faiblement couvertes par l'instrumentation actuelle. Ces données sont pourtant nécessaires pour réaliser des études dans différents domaines scientifiques, par exemple, pour étudier la structure interne de la terre en géosciences (via l'enregistrement d'ondes sismiques ayant traversé l'intérieur de la terre), le bilan thermique des océans en climatologie, ou encore la répartition des cétacés dans les océans en biologie. Toutes ces données doivent être traitées avant leur transmission en raison des capacités de transmission par satellite qui sont très limitées. Les applications de traitement des données sont usuellement développées par des spécialistes en systèmes embarqués ayant une bonne connaissance des caractéristiques spécifiques à l'instrument. Cependant, recourir à ces spécialistes limite grandement les possibilités d'adaptation des applications en fonction des besoins des scientifiques.

Afin de faciliter l'écriture d'applications pour l'instrument et par des non-spécialistes, nous avons créé le langage de programmation MeLa (Mermaid Language), spécifiquement conçu pour le flotteur Mermaid. Le langage permet de cacher les aspects propres aux systèmes embarqués. Il est basé sur des modèles informatiques à partir desquels nous calculons l'utilisation des ressources de l'instrument (i.e., processeur, énergie, transmission satellite) afin de s'assurer que les limites de l'instrument ne sont pas dépassées. Les modèles sont aussi utilisés pour composer (i.e., combiner) plusieurs applications à installer sur un même instrument et assurer leur compatibilité. Finalement, les modèles sont utilisés pour générer du code fiable et efficace (i.e., sans bugs et performant), d'une part pour simuler et tester les applications sur un ordinateur personnel, et d'autre part pour générer le code embarqué servant à la programmation des instruments.

Ce manuscrit de thèse est organisé en quatre chapitres. Le premier chapitre, nous présentons des différentes problématiques scientifiques et sociales concernées par l'acquisition de données dans les océans, nous introduisons le flotteur Mermaid et la manière dont il peut répondre à ces problématiques et terminons par les différentes approches de programmation de ce type d'instruments. Le deuxième chapitre, publié à l'occasion de la conférence OCEANS 2019, décrit les aspects techniques du langage MeLa et en particulier la manière dont nous utilisons les modèles et validons cette approche sur une carte de développement Arduino. Le troisième chapitre, publié dans la revue Sensors, est axé sur l'utilisation du langage, une méthode de développement est proposée et deux applications sont présentées pour la détection de séismes et la détection de baleines bleues. Le chapitre final récapitule les conclusions et dresse les perspectives pour des développements futurs.

Mots clés : Ingénierie Dirigée par les Modèles, Langage dédié, Systèmes embarqués, Traitement du signal, Capteurs, Océans

Float for multidisciplinary monitoring of the marine environment : from business expertise to embedded codes

As part of an ERC (European Research Council) project conducted at Geoazur from 2009 to 2015 by Guust Nolet, an autonomous profiler float equipped with a hydrophone and able to carry up to 8 sensors has been developed. It aims to acquire data in oceanic areas, poorly covered by current instrumentation. However, these data are necessary to carry out studies in various scientific fields, for example, to study the internal structure of the earth in geosciences (via the recording of seismic waves propagating inside the earth), the thermal balance of oceans in climatology, or the distribution of marine mammals in the oceans in biology. Most data must be processed before being transmitted by satellite because of the very limited transmission bandwidth. Data processing applications are usually developed by embedded systems specialists who have a good knowledge of the characteristics specific to the instrument. However, the need to involve these specialists greatly limits the flexibility, or even the ability of scientists to adapt the applications to their needs.

In order to enable scientists to write applications for the instrument, we have created the MeLa (Mermaid Language) programming language, specifically designed for the Mermaid float. The language makes it possible to hide embedded systems specific aspects. It is based on computer models that allow computing the resources usage of the instrument (i.e., processor, power, satellite transmission) in order to ensure that the instrument limits are not exceeded. Models are also used to compose (i.e., combine) several applications to be installed on the same instrument and to ensure that they are compatible. Finally, models are used to generate reliable and efficient code (i.e., without bugs and efficient), on the one hand, to simulate applications on a personal computer and verify their behavior, and, on the other hand, to generate the embedded code used to program the instruments.

This thesis is organized into four chapters. In the first chapter, we start by presenting the scientific and social issues involved in the acquisition of data in the oceans, then we introduce the Mermaid float and how it can respond to these issues and end by presenting different programming approaches for this type of instrument. The second chapter corresponds to an article published in the OCEANS 2019 conference proceedings. It shows the technical aspects of the MeLa language and in particular how we use models and how the approach is validated on an Arduino development board. The third chapter corresponds to an article published in the Sensors journal and is more focused on the use of language, a development method is proposed, and two applications are developed for the detection of earthquakes and the detection of blue whales. In the final chapter, we summarize the conclusions and offer a perspective of future developments.

Keywords : Model Driven Engineering, Domain Specific Language, Embedded systems, Signal processing, Sensors, Oceans

Remerciements

Je tiens à remercier les membres de mon jury de thèse. Je remercie Karin Sigloch et Frederik Simons pour avoir accepté d'être rapporteurs. Je remercie également les examinateurs Mireille Blay-Fornarino, Mireille Laigle, Dorian Cazau, Jean-Yves Royer et Guust Nolet.

Je remercie tout particulièrement la Région Sud et l'entreprise OSEAN pour avoir financé cette thèse. Merci à toute l'équipe d'OSEAN pour le soutien technique qu'ils m'ont apporté, notamment Romain Verfaillie, Franck Hieramente et Olivier Philippe.

Je remercie les personnes avec qui j'ai travaillé durant ces trois ans. Mes collègues du laboratoire Geoazur, et en particulier Guust Nolet pour sa disponibilité et l'aide qu'il m'a apporté dans la rédaction des articles, et pour avoir corrigé mes fautes d'anglais. Je remercie Yann Hello qui m'a soutenu durant ces 3 ans et aussi mon directeur de thèse Frédéric Cappa. Je remercie également mes collègues du laboratoire I3S, en particulier Mireille Blay-Fornarino et Sébastien Mosser qui m'ont appris énormément sur les méthodes de développements logiciels. Je remercie Dorian Cazau, du laboratoire Lab-STICC, qui m'a apporté sont expertise pour la détection de cétacés par acoustique passive.

Plus généralement je tiens à remercier les équipes de recherche qui m'ont accueillie. Les équipes SPARKS (I3S), SEISMES (Geoazur) et MARGES (Geoazur) pour avoir participé financièrement à cette thèse, mais aussi pour les moments de détente et d'échange passés au restaurant, à la plage ou à la montagne. Merci à Martijn van den Ende pour avoir géré le café et les mardis séismes au bâtiment 4. Je remercie aussi le la direction et personnel administratif de Géoazur et I3S.

Je remercie finalement tout mes collègues et amis doctorants que j'ai rencontré durant ces trois ans, je pense, en particulier à John-Anderson, Edson, Thu-Huong, Leandro, Benjamin, Sami, Daniel, Hector, Joseph, Carolina, Tiziano, Chao, Diego, Elif, Lionel, Nicolas, Florian, Caroline, Zoé, Alexiane, Maria-José, Hans, Carlo, Jean, Laurie et bien d'autres. Je garde de bons souvenirs des moments passés ensemble.

Table des matières

1 La surveillance des océans avec le flotteur Mermaid **10**
1.1 La surveillance des océans 11
1.1.1 Quels intérêts scientifiques et sociétaux? 11
1.1.2 Quels moyens d'observation? 12
1.2 Mermaid 14
1.2.1 Un instrument pour imager l'intérieur de la terre 14
1.2.2 Vers un instrument pluridisciplinaire 16
1.3 Programmer le Mermaid 17
1.3.1 Les applications embarquées 17
1.3.2 La programmation des systèmes embarqués 18
1.3.3 Écrire des algorithmes de traitement du signal 19
1.3.4 Les langages dédiés 19
Références 21

2 MeLa: un langage de programmation basé sur les modèles **24**
2.1 Introduction 26
2.2 Challenges 27
2.3 Models of applications 27
2.3.1 Overview 27
2.3.2 Introduction to MeLa 29
2.3.3 Description of MeLa 29
2.3.4 Platform model 30
2.3.5 Code generation 32
2.3.6 Analysis 33
2.3.7 Composition 34
2.4 Validation 34
2.4.1 Introduction to validation 34
2.4.2 Experimental setup 35
2.4.3 Functional validation 36
2.4.4 Reduction of expertise 36
2.4.5 Analysis validation 38
2.4.6 Produce efficient applications 39
2.5 Limitations and perspectives 40

2.6 State of the art . . . 40
2.7 Conclusion . . . 42
Références . . . 43

3 MeLa: un langage de programmation pluridisciplinaire 45
3.1 Introduction . . . 48
3.1.1 Context . . . 48
3.1.2 Motivations and objectives . . . 49
3.2 A programming language based on models . . . 50
3.2.1 Models for programming . . . 50
3.2.2 Description of MeLa . . . 52
3.2.3 Mermaid float architecture . . . 55
3.2.4 Code generation . . . 56
3.2.4.1 Overview . . . 56
3.2.4.2 Priority rules . . . 57
3.2.4.3 Benefits of MeLa for embedded software programming . . 57
3.2.5 Application verification . . . 58
3.2.5.1 Static analysis of applications . . . 58
3.3 Developing with MeLa . . . 62
3.4 Experiments . . . 64
3.4.1 Detection of earthquakes . . . 64
3.4.1.1 Scientific context . . . 64
3.4.1.2 The seismic detection algorithm . . . 64
3.4.1.3 Implementation with MeLa . . . 65
3.4.1.4 Evaluation of the algorithm . . . 66
3.4.2 Detection of blue whales . . . 66
3.4.2.1 Scientific context . . . 66
3.4.2.2 The blue whale detection algorithm . . . 67
3.4.2.3 Implementation with MeLa . . . 68
3.4.2.4 Evaluation of the algorithm . . . 69
3.5 Discussion and conclusion . . . 70
Références . . . 73

4 Perspectives et conclusion 75
4.1 Les perspectives pour MeLa . . . 76
4.1.1 Augmenter le niveau d'abstraction . . . 76
4.1.2 Automatiser le développement d'algorithmes . . . 76
4.1.3 Améliorer le processus de développement . . . 77
4.1.4 Améliorer les capacité d'analyse, de composition d'applications et de génération de code . . . 77
4.1.5 Étendre l'utilisation de MeLa à la programmation d'autres instruments . . . 78
4.2 Les perspectives pour l'instrument . . . 78
4.2.1 Étendre les capacités du flotteur . . . 78

4.2.2 Déploiement des applications . . . 79
4.3 Conclusion . . . 79
Références . . . 81
. . . 82

A Short list of MeLa data types and functions 88

B MeLa v1.0 reference manual 90
B.1 Language description . . . 92
B.1.1 Mission configuration . . . 92
B.1.2 Coordinator . . . 92
B.1.3 Acquisition modes . . . 92
B.1.4 Instructions . . . 93
B.2 Constants . . . 94
B.3 Data types . . . 95
B.3.1 Boolean . . . 95
B.3.2 Integer . . . 95
B.3.3 Float . . . 95
B.3.4 Complex numbers . . . 95
B.3.5 Arrays . . . 95
B.3.6 Buffers . . . 96
B.3.7 FFT . . . 96
B.3.8 IIR . . . 97
B.3.9 FIR . . . 97
B.3.10 CDF24 . . . 98
B.3.11 STALTA . . . 98
B.3.12 Trigger . . . 99
B.3.13 Distribution . . . 99
B.3.14 File . . . 100
B.4 Array and Buffer functions . . . 100
B.4.1 clear . . . 100
B.4.2 copy . . . 101
B.4.3 get . . . 101
B.4.4 put . . . 102
B.4.5 push . . . 102
B.4.6 select . . . 103
B.4.7 unselect . . . 103
B.5 Math functions . . . 104
B.5.1 abs . . . 104
B.5.2 add . . . 104
B.5.3 cdf24 . . . 105
B.5.4 cdf24ScalesPower . . . 105
B.5.5 conv . . . 106
B.5.5.1 corr . . . 106

B.5.5.2 cos . . . 107
B.5.6 cumulativeDistribution . . . 107
B.5.7 diff . . . 108
B.5.8 div . . . 108
B.5.9 dotProduct . . . 109
B.5.10 energy . . . 109
B.5.11 fft . . . 110
B.5.12 fir . . . 110
B.5.13 iir . . . 111
B.5.14 log10 . . . 111
B.5.15 magnitude . . . 111
B.5.16 max . . . 112
B.5.17 mean . . . 112
B.5.18 min . . . 113
B.5.19 mult . . . 113
B.5.20 negate . . . 114
B.5.21 pow . . . 114
B.5.22 rms . . . 114
B.5.23 sin . . . 115
B.5.24 sqrt . . . 115
B.5.25 stalta . . . 116
B.5.26 stdDev . . . 116
B.5.27 sub . . . 116
B.5.28 sum . . . 117
B.5.29 trigger . . . 117
B.5.30 var . . . 118
B.6 Utility functions . . . 118
B.6.1 ascentRequest . . . 118
B.6.2 convert . . . 119
B.6.3 getSampleIndex . . . 119
B.6.4 getTimestamp . . . 120
B.6.5 record . . . 120

C MeLa tutorial **121**
C.1 Continuous recording . . . 122
C.2 Detection algorithm . . . 124
C.3 Short acquisition . . . 127
C.4 Composition . . . 127

Chapitre 1

La surveillance des océans avec le flotteur Mermaid

Contents

1.1 La surveillance des océans **11**
1.1.1 Quels intérêts scientifiques et sociétaux? 11
1.1.2 Quels moyens d'observation? 12
1.2 Mermaid **14**
1.2.1 Un instrument pour imager l'intérieur de la terre 14
1.2.2 Vers un instrument pluridisciplinaire 16
1.3 Programmer le Mermaid **17**
1.3.1 Les applications embarquées 17
1.3.2 La programmation des systèmes embarqués 18
1.3.3 Écrire des algorithmes de traitement du signal 19
1.3.4 Les langages dédiés 19
Références **21**

1.1 La surveillance des océans

1.1.1 Quels intérêts scientifiques et sociétaux ?

Les océans couvrent 70% de la surface de la Terre. Ils produisent plus de 50% de l'oxygène sur terre dont 10% de l'oxygène atmosphérique que nous respirons chaque jour [Huang et al., 2018]. Ils participent activement à la régulation du climat notamment en redistribuant l'énergie de l'équateur vers les pôles via les principaux courants océaniques [Palter, 2015]. Ils absorbent 23% du dioxyde de carbone émis par les activités humaines [Friedlingstein et al., 2019]. Les changements climatiques peuvent avoir des conséquences importantes pour nos sociétés, en particulier pour l'apport en eau douce, la production de nourriture ou encore l'économie [Calel et al., 2020] La montée du niveau des océans risque également d'entraîner la migration de plusieurs centaines de millions de personnes vivant près des côtes [Neumann et al., 2015].

Bien que permettant de réguler le climat, l'absorption du dioxyde carbone par les océans les acidifie ce qui a un effet négatif sur l'écosystème marin [Guinotte and Fabry, 2008]. La pollution résultant du rejet de déchets plastiques, pétroliers ou autres produits chimiques a également un effet négatif [Sigler, 2014, Sindermann, 1996]. Cet écosystème est une source de nourriture importante pour de nombreuses personnes il est donc important de le maintenir en bonne santé. Qui plus est la pollution absorbée par les organismes marins se retrouve ensuite dans nos assiettes, par exemple sous forme de microplastiques, ce qui peut avoir des effets néfastes sur notre santé. La biodiversité marine est également une source d'inspiration pour les scientifiques qui développent de nouveaux médicaments [Malve, 2016] ou de nouvelles technologies [Fish, 2020]. Un déclin de la biodiversité risque donc d'avoir des conséquences importantes pour nos sociétés actuelles et futures.

Les océans couvrent de grandes zones ayant une activité géologique intense, en particulier les zones de divergence et de convergence des plaques et les points chauds [Bouysse, 2014]. Ces zones peuvent être d'importantes sources de séismes et de tsunamis et présenter des risques pour les populations [Sørensen et al., 2012]. Les géophysiciens se servent aussi de mesures réalisées en mer pour imager l'intérieur de la Terre et comprendre son fonctionnement [Mazzullo et al., 2017]. En effet les échanges de chaleur à l'intérieur de la Terre entrent en compte dans le bilan énergétique global, et donc sur notre compréhension du climat. Par ailleurs de nombreuses ressources pétrolière et gazière, qui sont importantes pour le fonctionnement de notre société, se trouvent en mer [Pinder, 2001].

La surveillance des océans permet de les étudier afin d'améliorer notre compréhension du climat, des écosystèmes marins et de l'impact des activités humaines sur ces derniers [Halpern et al., 2008]. Une meilleure compréhension des océans peut aider les responsables politiques à prendre des décisions pour limiter l'impact de nos activités sur l'environnement et prévenir les risques naturels associés qu'ils soient d'ordre climatiques ou géologiques (*i.e.*, les séismes, tsunamis, volcans, etc.)[Meiner, 2010] [1].

1. European Union Coastal and Marine Policy website : `https://ec.europa.eu/environment/marine/eu-coast-and-marine-policy/index_en.htm>`

1.1.2 Quels moyens d'observation ?

Surveiller les océans à l'échelle globale est difficile et onéreux en raison des technologies qui doivent supporter des conditions extrêmes (*e.g.*, météo, pression, salinité), et de la nécessité de recourir à des navires hauturiers pour leur mise en oeuvre et leur maintenance.

Un des moyens les plus efficaces pour une surveillance à l'échelle globale est la télédétection par satellite. Ces derniers permettent par exemple de mesurer la température des océans en surface [Minnett et al., 2019] et de suivre le phytoplancton [Joint and Groom, 2000]. La télédétection par satellite peut aussi permettre de compter les baleines [Fretwell et al., 2014] et il envisagé de détecter les tsunamis en temps réel à partir des perturbations électromagnétiques qu'ils génèrent dans l'ionosphère [Rolland et al., 2010]. En revanche la télédétection par satellite ne permet pas de mesurer ce qu'il se passe sous la surface des océans, comme la température de l'eau à 2000 mètres de profondeur ou de réaliser des acquisitions acoustiques.

Les réseaux câblés tels que le réseau *North-East Pacific Time-Series Undersea Networked Experiments* (NEPTUNE)[Barnes et al., 2008] au Canada, le *Dense Oceanfloor Network System for Earthquakes and Tsunamis* (DONET)[Kawaguchi et al., 2015] au Japon, le *Monterey Accelerated Research System* (MARS) [Dawe et al., 2005] aux États-Unis, ou encore le réseau *European Multidisciplinary Seafloor and water column Observatory* (EMSO)[Lefevre et al., 2018] en Europe, permettent de réaliser différents types de mesures à grande profondeur. Les réseaux câblés disposent de nombreux capteurs interchangeables, connectés sur une boite de jonction, elle-même reliée à la côte par un câble de plusieurs kilomètres pour la transmission des données en temps réel et l'alimentation en énergie. Ces dispositifs ont un coût élevé (*e.g.*, budget de 100 millions d'euros pour le réseau NEPTUNE[Barnes et al., 2008]), en raison du coût de la pose de câble sous-marin de plusieurs dizaines de kilomètres et des coûts de maintenance des instruments qui se fait à l'aide de véhicules sous-marins téléguidés (*i.e.*, ROV pour *Remotely Operated underwater Vehicle*) extrêmement sophistiqués et pilotés en temps réel depuis un navire [Fletcher et al., 2009].

Les bouées océanographiques ancrées (*i.e.*, mouillages) telles qu'ATLAS et TRITON faisant partie du Global Tropical Moored Buoy Array [McPhaden et al., 2009] peuvent être placées loin des côtes. Les capteurs sont placés le long de la ligne de mouillage jusqu'à plusieurs centaines de mètres de profondeur et certaines données peuvent être transmises par satellite. Ces systèmes demandent beaucoup de maintenance, typiquement une visite tous les 6 mois et une opération de redéploiement tous les ans ; ils peuvent aussi être victimes de dégradation intentionnelles ou non [Teng et al., 2009].

Lorsqu'il n'est pas nécessaire d'acquérir des données en surface, on peut recourir à des mouillages de sub-surface (*i.e.*, bouées restent sous la surface), par exemple pour réaliser des mesures acoustiques [D'Eu et al., 2012, Hello et al., 2019] ou de courants océaniques [Song et al., 2018]. D'autres instruments n'ont pas de système de flottabilité et sont directement posés au fond des océans comme les sismomètres fond de mer ou *Ocean Bottom Seismometers* (OBS). Ces systèmes nécessitent également une maintenance régulière, notamment pour récupérer les données une fois par an et changer les

batteries. La station de fond de mer MUG-OBS [Hello et al., 2019] permet de réduire en partie cette nécessité grâce à un système de petites navettes larguées par la station principale posée au fond et récupérées en surface, la station pouvant fonctionner plusieurs années au même emplacement.

Les véhicules de surface autonomes permettent de réaliser des mesures semblables à ce que ferait un bateau océanographique bien qu'avec des capacités plus limitées. Le Sphyrna [2] par exemple est un navire autonome à part entière équipé de nombreux instruments. Le Saildrone [Gentemann et al., 2020] et le Wave-glider [Hine et al., 2009, Manley and Willcox, 2010] sont des équipements plus petits, mais aussi plus accessibles financièrement qui utilisent l'énergie du vent ou des vagues pour avancer et peuvent donc naviguer plusieurs mois en toute autonomie.

Les planeurs (ou glider) [Rudnick et al., 2004] sont des robots sous-marins autonomes pouvant être pilotés à distance et capables de plonger à plusieurs centaines de mètres de profondeur. Des petits ailerons leurs permettent d'avancer en même temps qu'ils montent ou descendent et donc de parcourir de grandes distances dans les océans. Le système de ballast rempli d'huile qui se gonfle et se dégonfle pour les faire monter ou descendre est économe en énergie ce qui leur donne plusieurs mois d'autonomie.

Avec le même principe de fonctionnement, les flotteurs profileurs (comme ceux présentés dans cette thèse) [Riser et al., 2018] ont une autonomie encore plus grande, de plusieurs années, car ils dérivent au gré des courants marins et n'ont donc pas besoin de changer constamment de profondeur. Ils sont également capables de plonger plus profond que les planeurs, jusqu'à 6000 mètres pour certains modèles (APEX-Deep float). À l'inverse des systèmes en surface exposés à la lumière, ils sont peu sujets aux phénomènes d'encrassement biologique susceptibles de provoquer le dysfonctionnement de certains capteurs, en particulier les capteurs optiques. Le coût d'un flotteur n'est que de quelques dizaines de milliers d'euros donc plusieurs milliers de flotteurs peuvent être déployés pour le coût d'un réseau câblé à 100 millions d'euros. Leur coût d'utilisation est également plus faible sachant qu'ils sont autonomes et peuvent être déployés et récupérés depuis les côtes avec de petits navires. A titre d'exemple, le fonctionnement du flotteur Mermaid étudié dans le cadre de cette thèse est présenté figure 1.1.

Une dernière approche consiste à équiper des mammifères marins de capteurs (CTD-SRDL) [Boehme et al., 2009]. Cela permet d'étudier leurs comportements, mais aussi d'obtenir des données relatives à la climatologie. Les mammifères marins peuvent plonger très profondément plusieurs fois par jour et dans des zones potentiellement inaccessibles, en particulier dans les régions polaires couvertes de glace suivant la saison. Les capteurs doivent cependant avoir une taille relativement petite pour ne pas perturber les mammifères.

Tous ces instruments ont des avantages et des inconvénients. Certains sont plus adaptés pour mesurer une variable physique qu'un autre, par exemple les flotteurs profileurs sont plus adaptés pour réaliser des mesures dans la colonne d'eau, leur faible coût fait qu'ils peuvent être déployés en grand nombre sur toute la surface des océans comme dans le programme ARGO [Roemmich et al., 2009]. Ces moyens d'observations peuvent

2. Sphyrna website : `http://www.sphyrna-odyssey.com/`

être utilisés de façon combinée, par exemple il est envisageable d'utiliser des gliders ou wave-gliders pour estimer la position de stations sous marines [Bingham et al., 2012].

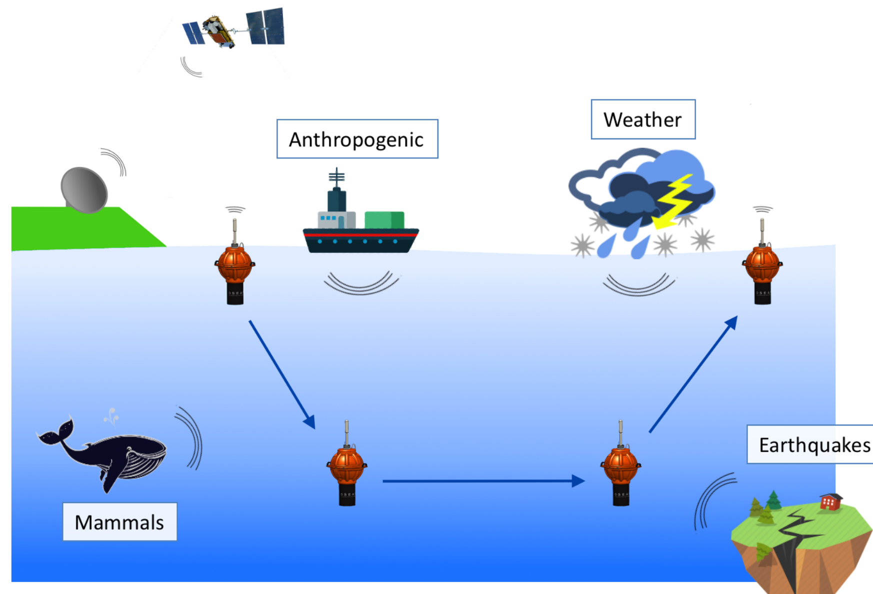


Figure 1.1 – Cycle de fonctionnement d'un flotteur Mermaid enregistrant différents types de signaux acoustiques dans les océans. Les étapes de descente et de remontée durent quelques heures alors que l'étape de parking pendant laquelle le flotteur reste en profondeur et dérive au gré des courants marins peut durer plusieurs jours. Les données sont transmises par satellite lorsque le flotteur est en surface.

1.2 Mermaid

1.2.1 Un instrument pour imager l'intérieur de la terre

Imager la répartition de chaleur à l'intérieur de la Terre est important pour comprendre son fonctionnement interne. Par exemple, l'origine des points chauds ayant formé des îles comme la Réunion ou les Galapagos (parmi d'autres) reste matière à débat. La tomographie sismique est une méthode d'imagerie qui se sert des ondes sismiques émises par les grands séismes. Lors d'un séisme, une onde de pression (ou onde P) équivalente à une onde acoustique est émise, cette onde se propage à l'intérieur de la Terre et ralentit ou accélère en fonction de la température du milieu qu'elle traverse. La mesure du temps de propagation de cette onde en différents points du globe permet d'obtenir une image en trois dimensions de la répartition de chaleur à l'intérieur du globe.

Pour obtenir une image fidèle à la réalité, il faut acquérir des données sur toute la surface terrestre. Ces données peuvent être obtenues grâce à des sismomètres, mais ils ne peuvent être installés de manière permanente que sur terre. La couverture en mer est insuffisante pour réaliser une tomographie à l'échelle globale (Figure 1.2). Les

sismomètres sous-marins (OBS) peuvent pallier à ce manque sur des durées relativement courtes de quelques mois. Cette approche est parfaitement adaptée pour la surveillance de crises sismiques locales [Feuillet et al., 2019], mais l'est beaucoup moins pour faire une tomographie à l'échelle globale qui requiert des mesures sur de longues périodes de temps et à différents emplacements.

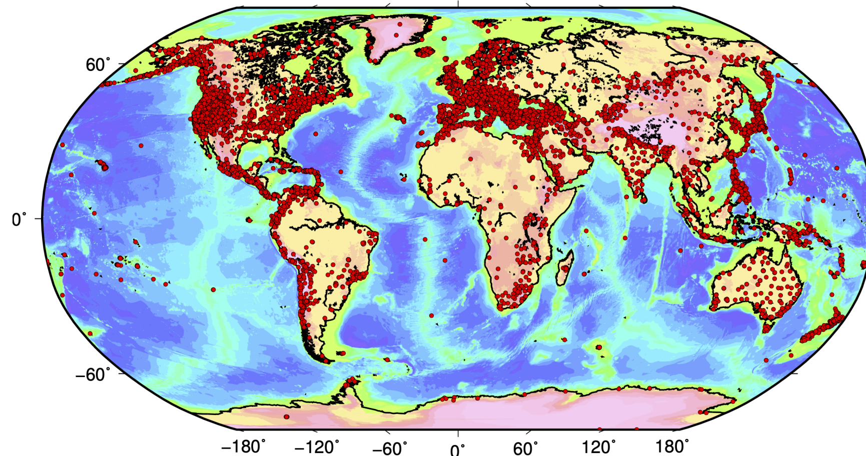


FIGURE 1.2 – Carte des stations enregistrées par l'*International Seismological Centre* (ISC), chaque station est représentée par un point rouge.

Les flotteurs profileurs sont en revanche bien adaptés pour la tomographie, car ils peuvent fonctionner plusieurs années en toute autonomie et se déplacent au gré des courants pour couvrir de larges zones. Il a par ailleurs été prouvé que ces flotteurs étaient parfaitement capables d'enregistrer des sismogrammes avec un hydrophone [Simons et al., 2009] (un microphone fonctionnant sous l'eau). Le sismogramme enregistré correspond à l'onde P s'étant convertie au fond des océans en onde acoustique (les deux types d'ondes sont identiques, mais le milieu de propagation change). Déployer ces instruments à une échelle globale pourrait augmenter significativement la résolution des images tomographiques (Figure 1.3).

L'instrument Mermaid, pour *Mobile Earthquake Recording Device in Marine Areas*, est un flotteur profileur équipé d'un hydrophone pour enregistrer les séismes en mer. Un algorithme de détection a été intégré afin que le flotteur remonte en surface lorsqu'il détecte un séisme et mesurer sa position grâce au GPS. Cet algorithme permet aussi de ne transmettre que les signaux correspondant à des séismes, car les capacités de transmission sont très limitées. Cet instrument a été développé dans le cadre de l'ERC GLOBALSEIS mené par Guust Nolet au laboratoire Geoazur (Université Côte d'Azur, CNRS et IRD). Lors de cet ERC une dizaine de flotteurs ont étés déployés dans l'océan Pacifique autour des îles Galapagos. Cette première expérience à permis d'améliorer significativement la résolution des images de répartition de chaleur sous le point chaud des Galapagos [Nolet et al., 2019]. D'autres flotteurs ont aussi été déployés en Méditerranée [Joubert et al., 2016] et dans l'océan Indien autour de l'île de la Réunion [Sigloch, 2013] pour les premiers essais.

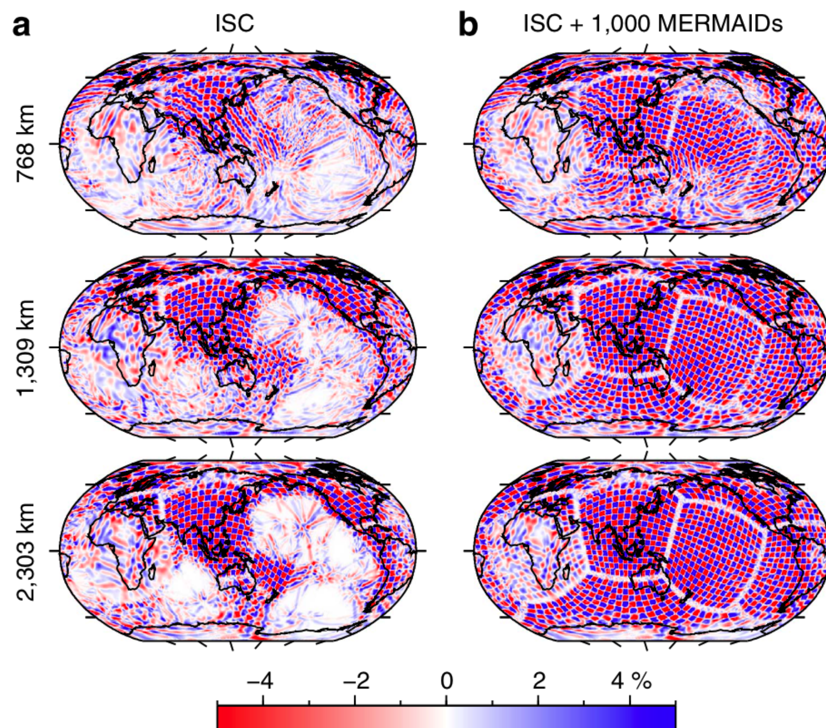


Figure 1.3 – Test de résolution d'imagerie tomographique pour un motif de damier à 768 km, 1309 et 2303 km de profondeur. La résolution obtenue dans les zones océaniques par le réseau de stations terrestres ISC en (a) est grandement améliorée par l'ajout de stations Mermaid en (b). La figure est extraite de l'article de A. Sukhovich, S. Bonnieux et al. [Sukhovich et al., 2015].

Un autre instrument appelé Son-O-Mermaid [Simon et al., 2014] et développé par Frederik J. Simons a l'intérêt de rester en surface ce qui lui permet de récupérer l'énergie des vagues et du soleil ainsi que d'avoir un positionnement GPS précis lors de l'enregistrement de séismes (qui est une information nécessaire pour la tomographie). Étant en surface il est cependant plus sensible aux aléas climatiques. Le Son-O-Mermaid est resté à l'état de développement depuis 2016 [3].

1.2.2 Vers un instrument pluridisciplinaire

L'instrument Mermaid a été développé initialement pour les besoins des sismologues. Afin d'attirer plus de scientifiques et ainsi favoriser son déploiement à l'échelle globale, l'instrument a été amélioré pour intégrer plusieurs autres capteurs. En plus de l'acquisition acoustique basse fréquence, jusqu'à 20 Hz, une chaîne d'acquisition pour l'acoustique haute fréquence, jusqu'à 10 kHz, a été ajoutée ainsi que des connecteurs pour ajouter différents capteurs.

3. Son-O-Mermaid website : `http://geoweb.princeton.edu/people/simons/Son-O-Mermaid.html`

Parmi les différents capteurs susceptibles d'être ajoutés, les capteurs de *Conductivity Temperature Depth* (CTD) sont installés sur les milliers de flotteurs du programme ARGO, et permettent de récolter des données utilisées pour les études de climatologie. Il existe aussi des capteurs chimiques et optiques utilisés dans le cadre de Biogeochemical Argo pour mesurer les concentrations d'oxygène, de nitrate, de chlorophylle, de particule en suspensions (*e.g.*, les algues, les bactéries, etc.), ou encore le pH ou la luminosité. Il serait également possible d'installer un écho-sondeur pour mesurer la profondeur sous le flotteur et éventuellement obtenir des informations sur la composition du fond marin. N'importe quel type de capteur peut théoriquement être installé, mais doivent résister à la pression, avoir une taille relativement réduite par rapport au flotteur et consommer peu d'énergie par rapport à ce dont dispose le flotteur (ce qui n'empêche pas d'utiliser des capteurs demandant une puissance électrique importante, mais sur de très courtes durées).

Les capteurs acoustiques peuvent être utilisés pour différentes applications. En plus de l'enregistrement des séismes, lointains ou locaux, ils peuvent aussi détecter la présence de cétacés, qui émettent des sons différents pour chaque espèce, mesurer le vent et la pluie en surface, qui génèrent un bruit spécifique en fonction de leur intensité, détecter le craquement de la glace qui fond dans les régions polaires, ou encore de mesurer la pollution acoustique générée par les navires ou activités humaines en mer. Certaines études cherchent également à utiliser les bruits émis par les navires en surface pour estimer la position d'instruments immergés ce qui est particulièrement intéressant pour les flotteurs qui dérivent au fond des océans sans possibilité de positionnement.

Les aspects techniques du Mermaid introduits dans le chapitre 3 de cette thèse sont également présentés dans l'article encyclopédique de Hello et Nolet [Hello and Nolet, 2020].

1.3 Programmer le Mermaid

1.3.1 Les applications embarquées

L'instrument Mermaid gagnerait à être utilisé par des scientifiques de différentes disciplines pour étudier les océans. Pour démocratiser son utilisation, il faut permettre aux scientifiques de programmer cet instrument avec différentes applications selon leurs besoins.

Pour gérer les capteurs et traiter les données acquises avant leur transmission, il faut écrire un logiciel adapté. Le besoin en traitement de données est particulièrement important pour les applications acoustiques qui génèrent un volume si important de données qu'il n'est pas envisageable de les transmettre par satellite. Ainsi, plutôt que d'envoyer les données acoustiques, il serait plus adapté de n'envoyer que les dates de détection d'une espèce particulière de cétacé, ou encore une estimation de la quantité de pluie tombée pendant une certaine période.

Ce type de gestion des données est appelé Edge Computing [Shi et al., 2016], par opposition au calcul centralisé sur un serveur (ou cloud computing). Cette méthode pré-

sente de plus en plus d'intérêt, notamment pour limiter les flux de données en provenance de caméras, ou de réseaux de capteurs en plein essor (*i.e.*, internet des objets) ou encore d'instruments scientifiques envoyés sur d'autres planètes. Cela est nécessaire lorsque les débits de transmission du système de communication sont limités, mais aussi pour réduire la consommation d'énergie (si l'instrument fonctionne avec des batteries ou pour des raisons écologiques).

Le flotteur Mermaid est un système embarqué, c'est-à-dire qu'il doit être capable de fonctionner de façon autonome, sans intervention humaine, et ce malgré le fait qu'il ait des capacités limitées, que ce soit en termes de puissance de calcul, de consommation d'énergie, de mémoire ou de transfert de données. Le logiciel développé doit également respecter des contraintes de temps d'exécution pour garantir l'intégrité des données. Programmer ce type d'instrument pour des scientifiques qui ne sont pas spécialistes des systèmes embarqués peut s'avérer complexe.

1.3.2 La programmation des systèmes embarqués

La programmation des systèmes embarqués commence souvent par la création d'une architecture logicielle [Gomaa, 2016, Grolleau et al., 2018, White, 2011]. Cette architecture peut être définie sur le papier en langage usuel ou grâce à des langages de modélisation de logiciel spécialisés. Il existe par exemple le langage UML (*Unified Modeling Language*) [Miles and Hamilton, 2006, Pilone and Pitman, 2005] qui est un langage graphique composé de 14 types de diagrammes différents pour représenter différents aspects structurels et comportementaux des logiciels. Une version spécialisée d'UML appelée MARTE (*Modeling and Analysis of Real-time and Embedded systems*) [Selić and Gérard, 2014] est plus orientée vers la modélisation de systèmes embarqués, avec par exemple avec la prise en compte des limitations des systèmes embarqués (également appelés propriétés non fonctionnelles). Le langage AADL (*Architecture Analysis and Design Language*) [Delange, 2017] est un autre langage d'architecture plus fortement spécialisé dans les systèmes embarqués. Il offre entre autres la possibilité de définir des tâches et la manière dont elles échangent des données. Les modèles créés avec UML et AADL représentent une architecture logicielle à différents niveaux de détails. Ces deux types de modèles peuvent être utilisés de façon complémentaire. Le code des logiciels est ensuite écrit en respectant ce qui a été définit dans les modèles d'architecture logicielle.

Le code de l'architecture logicielle peut être écrit manuellement ou généré automatiquement depuis les modèles. Les algorithmes contenus dans l'architecture doivent être écrits manuellement. Le langage de programmation C reste le langage le plus utilisé dans le domaine des systèmes embarqués [4]. Certains langages de haut niveau comme Matlab peuvent générer du code C, mais ce code est nettement moins efficace en termes de temps d'exécution et d'utilisation d'espace mémoire que s'il était écrit manuellement par un programmeur. Il reste que des adaptations sont toujours nécessaires pour interfacer les fonctionnalités avec le code d'architecture. Par ailleurs, une fois le code implémenté,

4. Aspencore 2019 Embedded Markets Study : `https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf`

des tests doivent être réalisés sur le système embarqué pour valider son fonctionnement.

Toutes ces étapes de développement requièrent une expertise spécifique en systèmes embarqués. Pour définir l'architecture logicielle, il est nécessaire de :

- maîtriser les modèles de tâches et d'échanges de données utilisées dans ce domaine.
- prendre en compte les limitations du système en estimant les temps d'exécution, la mémoire et l'énergie utilisée.
- coder avec un langage de bas niveau comme le C sans provoquer de bugs, par exemple en écrivant des données à un mauvais emplacement mémoire.
- comprendre le fonctionnement du microcontrôleur utilisé et de tous les composants intégrés et connectés à la carte électronique, le système d'exploitation temps réel utilisé doit lui aussi être étudié.
- maîtriser les outils de tests, tels que les oscilloscopes et analyseurs logiques.

1.3.3 Écrire des algorithmes de traitement du signal

Les algorithmes de traitement du signal sont développés dans des langages de plus haut niveau qui permettent d'être plus productifs. Les langages tels que Python ou Matlab sont bien adaptés pour développer des algorithmes. Ils offrent des bibliothèques de fonctions très complètes et des fonctionnalités de visualisation indispensables.

Les algorithmes sont écrits sur des ordinateurs avec de grandes capacités de mémoire et calcul. Les données à traiter sont contenues dans des fichiers et peuvent être traitées d'un bloc contrairement à un système embarqué ou les données proviennent directement des capteurs et doivent être traitées en temps réel. Par exemple un algorithme de détection pourra lire un fichier contenant une journée de donnée, appeler une fonction capable de traiter ces données en une seule exécution, puis enregistrer les signaux détectés dans un tableau et finalement dans un fichier. Écrire un algorithme de la sorte nécessite potentiellement beaucoup plus de mémoire que ce qui est disponible dans un système embarqué. Par ailleurs les mécanismes d'allocation de mémoire utilisés dans ces langages peuvent conduire à des temps d'exécution excessifs empêchant le traitement des données en temps réel sur un système embarqué.

Les algorithmes développés pour un ordinateur n'ont pas non plus besoin d'être très fiables. Si le code ne fonctionne pas normalement, par exemple à cause d'une boucle infinie, alors il suffit de forcer l'arrêt du programme et dans le pire des cas de redémarrer l'ordinateur. Pour un système embarqué autonome fonctionnant en pleine mer, il n'est pas possible d'intervenir. Il est donc important que les applications développées soient exemptes d'erreurs sous peine de risquer la perte de l'instrument, comme un déchargement prématuré des batteries si le processeur est utilisé façon excessive ou si un capteur n'est pas éteint après son utilisation.

1.3.4 Les langages dédiés

Les langages dédiés (ou DSL pour *Domain Specific Language*) [Fowler, 2010] sont des langages spécialisés pour une application donnée. Il existe de très nombreux langages spécifiques. On peut citer le tableur Excel qui permet de traiter des données sous forme

de tableaux ou LaTeX pour la rédaction de documents. Pour la sismologie le logiciel SAC est aussi un langage dédié au traitement de données sismologiques. Le langage Devito [Louboutin et al., 2019] est spécialisé dans l'inversion de formes d'ondes complètes utilisée en tomographie sismique. Pour les systèmes embarqués, il existe quelques langages dédiés, le langage AADL est spécialisé dans la description d'architecture de systèmes embarqués, d'autres langages comme PREESM [Pelcat et al., 2014] s'adressent à la programmation de systèmes multicoeurs, les langages synchrones comme Lustre [Halbwachs et al., 1991] et Esterel [Berry and Gonthier, 1992] sont dédiés à la programmation de systèmes temps réels critiques, tel que les avions ou les centrales nucléaires.

Les DSL font partie de l'Ingénieurie Dirigée par les Modèles (ou MDE pour *Model Driven Engineering*) [Rodrigues da Silva, 2015], et sont une approche de développement des logiciels étudiée dans le domaine du génie logiciel. Cette approche consiste à utiliser des modèles comme base pour le développement des logiciels. Les modèles permettent de représenter des éléments logiciels (mais aussi matériels) et de les agencer entre eux pour développer un logiciel (ou un système matériel). Les DSL permettent d'éditer les modèles avec un langage adapté au domaine d'expertise de l'utilisateur, c'est-à-dire à ses connaissances. Des règles de transformation permettent de générer automatiquement un code spécifique, par exemple du code pour un système embarqué, à partir des modèles. Il est également possible d'analyser et vérifier le comportement du logiciel en développement tout en prenant en compte les aspects matériels, comme les limitations d'un système embarqué.

Nous avons choisi d'utiliser cette approche de programmation pour permettre aux scientifiques qui ne sont pas spécialistes en systèmes embarqués d'écrire des applications pour le Mermaid. Nous avons donc développé le langage MeLa (pour *Mermaid Language*) qui permet de programmer cet instrument sans être un expert des systèmes embarqués, tout en vérifiant que les applications développées ne dépassent pas les limites de fonctionnement, et en permettant l'installation sur un même instrument de plusieurs applications développées de façon indépendante par plusieurs scientifiques.

Dans la suite de cette thèse, le chapitre 2 présente le langage MeLa du point de vue de l'Ingénieurie Dirigée par les Modèles et l'approche est validée sur une carte de développement Arduino. Il a été publié sous la forme d'un article pour la conférence OCEANS 2019. Le chapitre 3 s'adresse plus aux futurs utilisateurs du langage avec des exemples d'applications pour la détection de séismes et de baleines bleues. Il correspond à un article publié dans le journal Sensors en 2020. Le chapitre 4 discute des perspectives de recherche et de développement associées à MeLa. Finalement, un manuel de référence pour l'utilisation du langage MeLa est donné en annexe.

Références

Barnes, C. R., Best, M. M. R., and Zielinski, A. (2008). The NEPTUNE Canada regional cabled ocean observatory. *Sea Technology*, 49(7) :10–14.

Berry, G. and Gonthier, G. (1992). The Esterel synchronous programming language : design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152.

Bingham, B., Kraus, N., Howe, B., Freitag, L., Ball, K., Koski, P., and Gallimore, E. (2012). Passive and active acoustics using an autonomous wave glider. *Journal of Field Robotics*, 29(6) :911–923.

Boehme, L., Lovell, P., Biuw, M., Roquet, F., Nicholson, J., Thorpe, S. E., Meredith, M. P., and Fedak, M. (2009). Technical Note : Animal-borne CTD-Satellite Relay Data Loggers for real-time oceanographic data collection. *Ocean Science*, 5(4) :685–695.

Bouysse, P. (2014). Geological map of the world, explanatory note, 3rd revised edition at the 1 :35 000 000 scale. Commission For The Geological Map Of The World.

Calel, R., Chapman, S. C., Stainforth, D. A., and Watkins, N. W. (2020). Temperature variability implies greater economic damages from climate change. *Nature Communications*, 11(5028) :1–5.

Dawe, T. C., Bird, L., Talkovic, M., Brekke, K., Osborne, D. J., and Etchemendy, S. (2005). Operational Support of Regional Cabled Observatories The MARS Facility. In *Proceedings of OCEANS 2005 MTS/IEEE*, pages 1–6.

Delange, J. (2017). *AADL In Practice*. Reblochon Development Company.

D'Eu, J., Royer, J., and Perrot, J. (2012). Long-term autonomous hydrophones for large-scale hydroacoustic monitoring of the oceans. In *2012 Oceans - Yeosu*, pages 1–6.

Feuillet, N., Jorry, S., Crawford, W. C., Deplus, C., Thinon, I., Jacques, E., Saurel, J. M., Lemoine, A., Paquet, F., Daniel, R., Gaillot, A., Satriano, C., Peltier, A., Aiken, C., Foix, O., Kowalski, P., Laurent, A., Beauducel, F., Grandin, R., Ballu, V., Bernard, P., Donval, J. P., Géli, L., Gomez, J., Pelleau, P., Guyader, V., Rinnert, E., Besançon, S., Bertil, D., Lemarchand, A., and Vanderwoerd, J. (2019). Birth of a large volcano offshore Mayotte through lithosphere-scale rifting. In *AGU Fall Meeting Abstracts*, volume 2019, pages V52D–01.

Fish, F. E. (2020). Advantages of aquatic animals as models for bio-inspired drones over present AUV technology. *Bioinspiration & Biomimetics*, 15(2) :025001.

Fletcher, B., Bowen, A., Yoerger, D., and Whitcomb, L. (2009). Journey to the challenger deep : 50 years later with the nereus hybrid remotely operated vehicle. *Marine Technology Society Journal*, 43 :65–76.

Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional, 1st edition.

Fretwell, P. T., Staniland, I. J., and Forcada, J. (2014). Whales from space : Counting southern right whales by satellite. *PLOS ONE*, 9(2) :1–9.

Friedlingstein, P., Jones, M. W., O'Sullivan, M., Andrew, R. M., Hauck, J., Peters, G. P., Peters, W., Pongratz, J., Sitch, S., Le Quéré, C., Bakker, D. C. E., Canadell, J. G., Ciais, P., Jackson, R. B., Anthoni, P., Barbero, L., Bastos, A., Bastrikov, V., Becker, M., Bopp, L., Buitenhuis, E., Chandra, N., Chevallier, F., Chini, L. P., Currie, K. I., Feely, R. A., Gehlen, M., Gilfillan, D., Gkritzalis, T., Goll, D. S., Gruber, N., Gutekunst, S., Harris, I., Haverd, V., Houghton, R. A., Hurtt, G., Ilyina, T., Jain, A. K., Joetzjer, E., Kaplan, J. O., Kato, E., Klein Goldewijk, K., Korsbakken, J. I., Landschützer, P., Lauvset, S. K., Lefèvre, N., Lenton, A., Lienert, S., Lombardozzi, D., Marland, G., McGuire, P. C., Melton, J. R., Metzl, N., Munro, D. R., Nabel, J. E. M. S., Nakaoka, S.-I., Neill, C., Omar, A. M., Ono, T., Peregon, A., Pierrot, D., Poulter, B., Rehder, G., Resplandy, L., Robertson, E., Rödenbeck, C., Séférian, R., Schwinger, J., Smith, N., Tans, P. P., Tian, H., Tilbrook, B., Tubiello, F. N., van der Werf, G. R., Wiltshire, A. J., and Zaehle, S. (2019). Global Carbon Budget 2019. *Earth System Science Data*, 11(4) :1783–1838.

Gentemann, C. L., Scott, J. P., Mazzini, P. L. F., Pianca, C., Akella, S., Minnett, P. J., Cornillon, P., Fox-Kemper, B., Cetinić, I., Chin, T. M., Gomez-Valdes, J., Vazquez-Cuervo, J., Tsontos, V., Yu, L., Jenkins, R., De Halleux, S., Peacock, D., and Cohen, N. (2020). Saildrone : Adaptively Sampling the Marine Environment. *Bulletin of the American Meteorological Society*, 101(6) :E744–E762.

Gomaa, H. (2016). *Real-Time Software Design for Embedded Systems*. Cambridge University Press.

Grolleau, E., Hugues, J., Yassine, O., and Henri, B. (2018). *Introduction aux systèmes embarqués temps réel : Conception et mise en oeuvre*. Dunod.

Guinotte, J. and Fabry, V. (2008). Ocean acidification and its potential effects on marine ecosystems. *Annals of the New York Academy of Sciences*, 1134 :320–42.

Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991). The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320.

Halpern, B. S., Walbridge, S., Selkoe, K. A., Kappel, C. V., Micheli, F., D'Agrosa, C., Bruno, J. F., Casey, K. S., Ebert, C., Fox, H. E., Fujita, R., Heinemann, D., Lenihan, H. S., Madin, E. M. P., Perry, M. T., Selig, E. R., Spalding, M., Steneck, R., and Watson, R. (2008). A global map of human impact on marine ecosystems. *Science*, 319(5865) :948–952.

Hello, Y. and Nolet, G. (2020). Floating Seismographs (MERMAIDS). In Gupta, H. K., editor, *Encyclopedia of Solid Earth Geophysics*, pages 1–6, Cham, Switzerland. Springer.

Hello, Y., Royer, J. Y., Rivet, D., Charvis, P., Yegikyan, M., and Philippe, O. (2019). New versatile autonomous platforms for long-term geophysical monitoring in the ocean. In *OCEANS 2019 - Marseille*, pages 1–8.

Hine, R., Willcox, S., Hine, G., and Richardson, T. (2009). The Wave Glider : A Wave-Powered autonomous marine vehicle. In *OCEANS 2009*, pages 1–6.

Huang, J., Huang, J., Liu, X., Li, C., Ding, L., and Yu, H. (2018). The global oxygen budget and its future projection. *Science Bulletin*, 63(18) :1180–1186.

Joint, I. and Groom, S. B. (2000). Estimation of phytoplankton production from space : current status and future potential of satellite remote sensing. *Journal of Experimental Marine Biology and Ecology*, 250(1) :233–255.

Joubert, C., Nolet, G., Bonnieux, S., Deschamps, A., Dessa, J.-X., and Hello, Y. (2016). P-Delays from Floating Seismometers (MERMAID), Part I : Data Processing. *Seismological Research Letters*, 87(1) :73–80.

Kawaguchi, K., Kaneko, S., Nishida, T., and Komine, T. (2015). *Construction of the DONET real-time seafloor observatory for earthquakes and tsunami monitoring*, pages 211–228. Springer, Berlin, Heidelberg.

Lefevre, D., Zakardkjian, B., and Embarcio, D. (2018). Unique observatories for sea science and particle astrophysics : The EMSO-Antares and EMSO-Western Ionian nodes in the Mediterranean Sea. In *8th Very Large Volume Neutrino Telescope Workshop*, volume 207, page 09004, Dubna, Russia.

Louboutin, M., Lange, M., Luporini, F., Kukreja, N., Witte, P. A., Herrmann, F. J., Velesko, P., and Gorman, G. J. (2019). Devito (v3.1.0) : an embedded domain-specific language for finite differences and geophysical exploration. *Geoscientific Model Development*, 12(3) :1165–1187.

Malve, H. (2016). Exploring the ocean for new drug developments : Marine pharmacology. *Journal of Pharmacy And Bioallied Sciences*, 8(2) :83–91.

Manley, J. and Willcox, S. (2010). The Wave Glider : A persistent platform for ocean science. In *OCEANS'10 IEEE SYDNEY*, pages 1–5.

Mazzullo, A., Stutzmann, E., Montagner, J.-P., Kiselev, S., Maurya, S., Barruol, G., and Sigloch, K. (2017). Anisotropic Tomography Around La Réunion Island From Rayleigh Waves. *Journal of Geophysical Research : Solid Earth*, 122(11) :9132–9148.

McPhaden, M. J., Ando, K., Bourlès, B., Freitag, H. P., Lumpkin, R., Masumoto, Y., Murty, V. S. N., Nobre, P., Ravichandran, M., Vialard, J., Vousden, D., and Yu, W. (2009). The global tropical moored buoy array. In *OceanObs'09 : Sustained Ocean Observations and Information for Society*, volume 2, Venice, Italy.

Meiner, A. (2010). Integrated maritime policy for the European Union — consolidating coastal and marine information to support maritime spatial planning. *Journal of Coastal Conservation*, 14(1) :1–11.

Miles, R. and Hamilton, K. (2006). *Learning UML 2.0*. O'Reilly Media.

Minnett, P., Alvera-Azcárate, A., Chin, T., Corlett, G., Gentemann, C., Karagali, I., Li, X., Marsouin, A., Marullo, S., Maturi, E., Santoleri, R., Saux Picart, S., Steele, M., and Vazquez-Cuervo, J. (2019). Half a century of satellite remote sensing of sea-surface temperature. *Remote Sensing of Environment*, 233 :111366.

Neumann, B., Vafeidis, A. T., Zimmermann, J., and Nicholls, R. J. (2015). Future coastal population growth and exposure to sea-level rise and coastal flooding - a global assessment. *PLOS ONE*, 10(3) :1–34.

Nolet, G., Hello, Y., Lee, S. v. d., Bonnieux, S., Ruiz, M. C., Pazmino, N. A., Deschamps, A., Regnier, M. M., Font, Y., Chen, Y. J., and Simons, F. J. (2019). Imaging the Galápagos mantle plume with an unconventional application of floating seismometers. *Scientific Reports*, 9(1326) :1–12.

Palter, J. B. (2015). The Role of the Gulf Stream in European Climate. *Annual Review of Marine Science*, 7(1) :113–137.

Pelcat, M., Desnos, K., Heulot, J., Guy, C., Nezan, J., and Aridhi, S. (2014). Preesm : A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. In *2014 6th European Embedded Design in Education and Research Conference (EDERC)*, pages 36–40.

Pilone, D. and Pitman, N. (2005). *UML 2.0 in a Nutshell*. O'Reilly Media.

Pinder, D. (2001). Offshore oil and gas : global resource knowledge and technological change. *Ocean & Coastal Management*, 44(9) :579–600.

Riser, S. C., Swift, D., and Drucker, R. (2018). Profiling Floats in SOCCOM : Technical Capabilities for Studying the Southern Ocean. *Journal of Geophysical Research (Oceans)*, 123(6) :4055–4073.

Rodrigues da Silva, A. (2015). Model-driven engineering : A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43 :139–155.

Roemmich, D., Johnson, G., Riser, S., Davis, R., Gilson, J., Owens, W., Garzoli, S., Schmid, C., and Ignaszewski, M. (2009). The Argo Program : Observing the Global Ocean with Profiling Floats. *Oceanography*, 22(2) :34–43.

Rolland, L. M., Occhipinti, G., Lognonné, P., and Loevenbruck, A. (2010). Ionospheric gravity waves detected offshore Hawaii after tsunamis. *Geophysical Research Letters*, 37(17).

Rudnick, D. L., Davis, R. E., Eriksen, C. C., Fratantoni, D. M., and Perry, M. J. (2004). Underwater Gliders for Ocean Research. *Marine Technology Society Journal*, 38(2) :73–84.

Selić, B. and Gérard, S. (2014). *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE*. Elsevier.

Shi, W., Cao, J., Zhang, Q., Li, Y., and Xu, L. (2016). Edge Computing : Vision and Challenges. *IEEE Internet of Things Journal*, 3(5) :637–646.

Sigler, M. (2014). The Effects of Plastic Pollution on Aquatic Wildlife : Current Situations and Future Solutions.

Water, Air, & Soil Pollution, 225(11) :2184.

Sigloch, K. (2013). Short Cruise Report for Research Cruise M101 on RV "METEOR", RHUM-RUM project, Oct-Dec 2013. RHUM-RUM – Seismological Imaging of a mantle plume under La Réunion, western Indian Ocean. Technical report, Universität Hamburg.

Simon, J. D., Simons, F. J., and Nolet, G. (2014). Data and Analysis Methods of the Son-O-Mermaid and MERMAID Experiments. In *AGU Fall Meeting Abstracts*, pages S13B–4448.

Simons, F. J., Nolet, G., Georgief, P., Babcock, J. M., Regier, L. A., and Davis, R. E. (2009). On the potential of recording earthquakes for global seismic tomography by low-cost autonomous instruments in the oceans. *Journal of Geophysical Research : Solid Earth*, 114(B5).

Sindermann, C. J. (1996). *Ocean Pollution : Effects on Living Resources and Humans*. CRC Press.

Song, L., Li, Y., Wang, J., Wang, F., Hu, S., Liu, C., Diao, X., and Guan, C. (2018). Tropical Meridional Overturning Circulation Observed by Subsurface Moorings in the Western Pacific. *Scientific Reports*, 8(7632) :1–8.

Sørensen, M. B., Spada, M., Babeyko, A., Wiemer, S., and Grünthal, G. (2012). Probabilistic tsunami hazard in the Mediterranean Sea. *Journal of Geophysical Research : Solid Earth*, 117(B1).

Sukhovich, A., Bonnieux, S., Hello, Y., Irisson, J.-O., Simons, F. J., and Nolet, G. (2015). Seismic monitoring in the oceans by autonomous floats. *Nature Communications*, 6(8027) :1–6.

Teng, C., Cucullu, S., McArthur, S., Kohler, C., Burnett, B., and Bernard, L. (2009). Buoy vandalism experienced by noaa national data buoy center. In *OCEANS 2009*, pages 1–8.

White, E. (2011). *Making Embedded Systems : Design Patterns for Great Software*. O'Reilly Media, Inc.

Chapitre 2

MeLa : un langage de programmation basé sur les modèles

Contents

2.1 Introduction **26**
2.2 Challenges **27**
2.3 Models of applications **27**
2.3.1 Overview 27
2.3.2 Introduction to MeLa 29
2.3.3 Description of MeLa 29
2.3.4 Platform model 30
2.3.5 Code generation 32
2.3.6 Analysis 33
2.3.7 Composition 34
2.4 Validation **34**
2.4.1 Introduction to validation 34
2.4.2 Experimental setup 35
2.4.3 Functional validation 36
2.4.4 Reduction of expertise 36
2.4.5 Analysis validation 38
2.4.6 Produce efficient applications 39
2.5 Limitations and perspectives **40**
2.6 State of the art **40**
2.7 Conclusion **42**
Références **43**

Ce chapitre correspond à l'article publié pour la conférence OCEANS 2019 : *S. Bonnieux, S. Mosser, M. Blay-Fornarino, Y. Hello and G. Nolet, "Model driven programming of autonomous floats for multidisciplinary monitoring of the oceans," OCEANS 2019 - Marseille, Marseille, France, 2019, pp. 1-10, doi : 10.1109/OCEANSE.2019.8867453.* Il présente les défis liés à la programmation du flotteur Mermaid, introduit l'utilisation des modèles dans le cadre de la création du langage MeLa et la manière dont ils peuvent nous permettre de répondre aux problèmes posés. Finalement, l'approche est validée sur une carte de développement Arduino.

Model driven programming of autonomous floats for multidisciplinary monitoring of the oceans

Sébastien Bonnieux, Sébastien Mosser, Mireille Blay-Fornarino,
Yann Hello and Guust Nolet

Résumé

La surveillance des océans à l'aide de flotteurs autonomes présente un grand intérêt pour de nombreuses disciplines. La surveillance à l'échelle mondiale nécessite une approche multidisciplinaire pour être abordable. Dans ce but, nous proposons une approche qui permet aux océanographes de différentes spécialités de développer des applications pour les flotteurs autonomes. Cependant, le développement de telles applications nécessite généralement une expertise en systèmes embarqués. Les applications doivent être fiables et efficaces compte tenu des ressources limitées des flotteurs (énergie, puissance de traitement). Nous avons suivi une approche d'ingénierie guidée par les modèles (Model Driven Engineering) composée d'un langage dédié (Domain Specific Language) pour permettre aux océanographes de développer des applications, d'outils d'analyse pour garantir l'efficacité et la fiabilité des applications, d'un outil de composition pour permettre le déploiement de différentes applications sur un même flotteur et d'un générateur de code qui produit un code efficace et fiable pour le flotteur. Nous présentons notre approche avec une application biologique et une application sismologique. Nous la validons par des métriques techniques et une expérimentation.

Abstract

Monitoring of the oceans with autonomous floats is of great interest for many disciplines. Monitoring on a global scale needs a multidisciplinary approach to be affordable. For this purpose, we propose an approach that allows oceanographers from different specialities to develop applications for autonomous floats. However, developing such applications usually requires expertise in embedded systems, and they must be reliable and

efficient with regards to the limited resources of the floats (*e.g.*, energy, processing power). We have followed a *Model Driven Engineering* approach composed of *i)* a *Domain Specific Language* to allow oceanographers to develop applications, *ii)* analysis tools to ensure that applications are efficient and reliable, *iii)* a composition tool to allow the deployment of different applications on a same float, and *iv)* a code generator that produces efficient and reliable code for the float. We present our approach with a biological and a seismological application. We validate it with technical metrics and an experiment.

2.1 Introduction

Autonomous floats [Gould, 2005] are instruments designed to monitor the oceans over long periods, as it is done for several years by the Argo project [Roemmich et al., 2009]. These instruments drift at several thousands of meters (*e.g.*, 2000 meters) for several days (*e.g.*, 10 days) to conduct measurements and transmit the collected data at the surface through satellite communication. Global acoustic monitoring of the oceans with autonomous floats is of great interests for the *Passive Acoustic Monitoring* (PAM) community [Baumgartner et al., 2018] but efforts are spread among the different specialties such as *i)* biologists with cetacean click detection applications [Matsumoto et al., 2013], *ii)* meteorologists with rainfall detection [Ma and Nystuen, 2005] or *iii)* seismologists with earthquake detection [Sukhovich et al., 2015].

Large-scale experiments have already begun in the field of seismology [Nolet et al., 2019]. However, acoustic monitoring of the oceans on a global scale cannot be done without a multidisciplinary collaboration. Indeed, even if the cost of one float is low compared to other solutions (*e.g.*, moored buoys), an array of floats at the global scale is still expensive.

Our approach is to bring different monitoring efforts together with an adaptive float on which several applications can be installed. In this article, we focus on applications for passive acoustic monitoring of the oceans with a hydrophone, but a float can include other arbitrary sensors for physical, or chemical measurements, bringing together a broader community.

There are actually no standard applications in the PAM community, each speciality has different needs that may change over time. But developing such applications by traditional means takes a lot of time and is expensive. Therefore, we want to give oceanographers, the capacity to write applications by themselves with a minimum of effort. Moreover, we want to give them the guarantee that applications will work correctly on the instrument, that they are reliable.

A float has typically four states of operations : the descent, the park, the ascent and the surface. It uses actuators to regulate its depth and uses communication devices at the surface. The applications define the depth and duration of a dive, and the measurements to realize and transmit by satellite. These have an impact on the battery lifetime and on the communication costs. Thus, we want to help oceanographers to take these properties into consideration such that they can produce efficient applications.

2.2 Challenges

We illustrate our problem with an example : assume that two oceanographers, a geoscientist and a biologist, want to develop their own applications, the *seismic* application and the *whales* applications that will be installed together on the same instrument.

The *seismic* application consists of continuously listening to the acoustic signal received by the hydrophone during the "park" state of the float. If a change is detected in the acoustic signal, an algorithm determines if the change corresponds to a seismic wave ; then, depending on the level of confidence, the signal may be recorded for satellite transmission, and the application may command the float to ascend.

The *whales* application consists of listening to the acoustic signal during a short time at a fixed time interval (*e.g.*, an acquisition of 5 minutes every 15 minutes). For each acquisition, an algorithm determines the probability of presence of whales that is further transmitted by satellite. This application is activated during the descent, park and ascent states of the float.

From this example, we identify three challenges. The first challenge is to allow oceanographers to develop applications for the float by themselves, without the help of embedded software experts (C1). Indeed, developing such applications requires skills in embedded software to program the microprocessor and to define a software architecture [Gomaa, 2016]. For example, the *seismic* and the *whales* applications must be implemented as tasks with appropriate execution priority and synchronisation mechanisms to access data from the hydrophone.

The second challenge is to help the production of efficient and reliable applications (C2). For that purpose, we want to help oceanographers to consider the battery lifetime, the satellite communication costs and execution time constraints, indeed if an application does not respect these constraints, data from the hydrophone could be altered. Computing these properties requires specific knowledge and analysis methods that are not in the domain of expertise of our developers, the oceanographers.

The third challenge is to allow several applications defined separately to be executed on the same instrument (C3). Since the applications operate concurrently, they must share the functionalities of the instrument. In the example, the applications must share the data coming from the hydrophone, as well as the processing time of the processor.

2.3 Models of applications

2.3.1 Overview

To overcome these challenges, we propose a *Model Driven Engineering* (MDE) [Mussbacher et al., 2014], [Schmidt, 2006], [Kent, 2002] approach, illustrated in figure 2.1 and described below :

(1) Developers describe applications using a *Domain Specific Language* (DSL) [Fowler, 2010] called MeLa, that stands for *Mermaid Language*, where Mermaid is the name of the float targeted by this contribution. The MeLa language responds to C1, by allowing

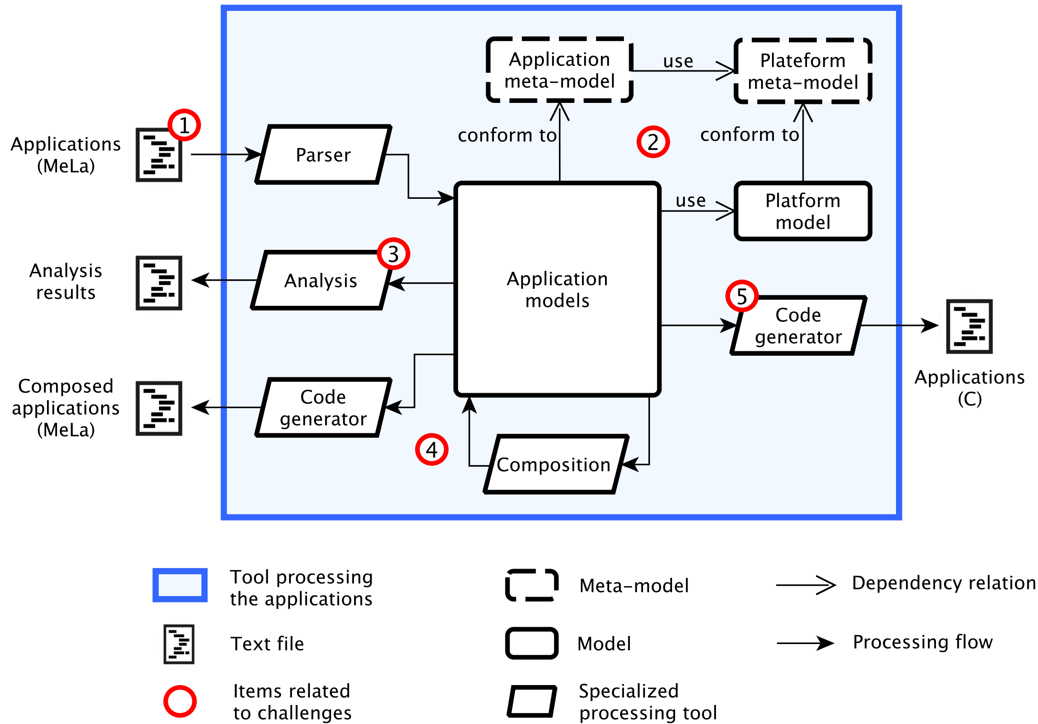


Figure 2.1 – Modeling of applications.

oceanographers to express applications for the instrument. Applications expressed with the MeLa language are transformed into models with a specialized tool (*i.e.*, the parser).

(2) These models are conform to a dedicated meta-model. To manage the dependency on the platform itself, that is the float, the application models use a platform model, itself conform to a meta-model. The dependency between these models is supported by the relation between the application meta-model and the platform meta-model. The platform model contains information about available functions and sensors and the amount of resources they use (*e.g.*, processing time or power consumption). Models and meta-models are the backbone for processing the applications.

(3) The analysis tool uses the application models and associated platform model to compute the battery lifetime, the satellite communication costs and verify real-time constraints of the applications. The results of the analysis are returned to the developer so that she can modify the application accordingly, this responds to C2.

(4) The composition tool aims to merge several applications into a single one, responding to C3. The developer can look at the composition results with the MeLa code generated from the model.

(5) To deploy these applications on the instrument, the platform specific code is generated from the model ; this code follows an architecture defined in the platform meta-model. This part is linked to C1, by allowing the deployment of applications on

the float, to C2, by producing efficient and reliable code, and to C3, by allowing the concurrent execution of applications.

In the next sections we present the MeLa DSL, then we describe the architecture of the generated code, and finally we give an overview of the analysis and composition methods implemented in the tools processing the applications.

2.3.2 Introduction to MeLa

The goal of MeLa is to allow developers to write applications using features of the instrument without having to worry about details of embedded software development.

Such features include : defining the depth and duration of a dive, choosing a sensor among those available, a procedure to acquire the data (*e.g.*, continuously for the *seismic* application or during a short time for the *whales* application), the algorithms to process the data, selecting data to record for satellite transmission, or for a later recovery of the float, and requesting the float to ascend according to results of data processing.

The language does not allow for the definition of new processing algorithms ; however, common processing algorithms such as filters or Fourier transforms are accessible at the level of the platform model in the form of a library of functions.

2.3.3 Description of MeLa

We chose to describe the language through examples introduced in the motivation part. The MeLa code for those two applications is given in table 2.1. An application written in MeLa consists of different parts [1] :

(1) A mission configuration, `Mission` (lines s2, w2) [2], that contains the park time and depth of the float.

(2) A coordinator, `Coordinator` (lines s7, w7), that defines politics of activation of acquisition modes. Acquisition modes can be executed according to the active state of the float (descent, park or ascent) and a period of execution can be given for short time acquisition (lines w9-w13).

(3) One or more acquisition modes. There are of two types, `ContinuousAcqMode` and `Short- AcqMode` (lines s12, w16). The first one corresponds to the continuous acquisition of the *seismic* application, the second one corresponds to the short acquisition of the *whales* application. For the first one the sensor is always active, whereas for the second one the sensor is stopped after the acquisition. The *seismic* application uses a continuous acquisition mode because it aims to detect the beginning of a seismic signal. The *whales* application does not have this constraint, and choosing a short acquisition mode allows to save the batteries. An acquisition mode is constituted of different parts :

(3.a) The `Input` (lines s15, w19) is associated to a sensor and a variable containing the data to process. In this contribution, we consider only periodic sensors (*i.e.*, sensors

1. The parts are identified in the comments of the table (*e.g.*, *# 1.*)

2. Reference to line numbers are given with an *s* for the *seismic* application and a *w* for the *whales* application (*e.g.*, s1, w1).

that send samples periodically), in this case a hydrophone. The input variable, an array with a size chosen by the developer, receives the data from the sensors.

(3.b) The `Variables` (lines s20, w24) part contains a list of variables. They are only accessible from the acquisition mode to which they belong. There are several variable types available, for example, the type `transmitFile` allows to define a file transmitted through satellite.

(3.c) The sequences of instructions (lines s27, s36, w30) contains the instructions to process the data. The first sequence defined in an application (lines s27, w30) is executed each time the array defined in the `Input` is full. Sequences of instructions can be of two types, `RealTimeSequence` or `ProcessingSequence`. A real-time sequence (line s27) has an execution time constraint to guarantee the continuous acquisition of the acoustic signal. A processing sequence (lines s36, w30) has no execution time constraint, thus it can contain algorithms with a long execution time. A real-time sequence can call a processing sequence (line s32), but in that case the data possibly sent by the sensor are ignored, so that the input variable containing the data can be used.

(4) An instruction can be a function call or a condition. Functions allow to use algorithms (lines s29, w31), to record data (lines s40, w34, w36) or to request the float to ascend (line s44). Conditions allow to call different instructions depending on variable values. Conditions must be annotated with an average probability of execution (lines s31, w32). For example, we specify line s31, that the condition is true with an average of ten per week. This condition is true if a signal that could have a seismic origin is detected by the `seisDetection` algorithm. The `@` stands for annotation. The annotations do not change the behaviour of the code but are used to estimate the energy consumption of applications and quantity of data recorded in memory for satellite transmission. It is up to the user to choose values in accordance with realistic expectations.

2.3.4 Platform model

Function prototypes and models of sensors are defined in the platform model. Each function prototype is defined by a name (*i.e.*, the name of the function), a list of parameter types and a return type. They also contain information about their resource usage, like power consumption, memory usage or execution time (processor usage), that can depend on parameters passed to a function when it is called. Some information represents the capacity of a function to request the float to ascend. The models of sensors can contain specific information, for example, the model of the hydrophone contains its sampling period. Moreover the platform model contains information about available resources of the instrument. This information is used to compute energy consumption, cost of satellite transmission and execution time constraints.

The platform model also has the advantage to enable the use of different platform configurations without having to change the MeLa application. For example, if the processor used by the float is changed, the MeLa code remains valid. It is up to experts in embedded software to create a new platform model with adapted code generation.

TABLE 2.1 – MeLa code for the seismic and whales applications.

Seismic application

```
#1 Mission configuration
Mission:
    ParkTime: 10 days;
    ParkDepth: 1500 meters;

#2 Coordination of acquisition modes
Coordinator:
    ParkAcqModes:
        Seismic;

#3 Definition of a continuous
#  acquisition mode
ContinuousAcqMode Seismic:

#3.a Input
Input:
    sensor: Hydrophone;
    data: x[40];

#3.b Variables
Variables:
    int[2400] lastminute;
    bool detect;
    float criterion;
    transmitFile f;

#3.c Sequences of instructions
RealTimeSequence detection:
    appendArray(lastminute, x);
    detect = seisDetection(x);
    if detect:
        @probability = 10 per week
        call discriminate;
    endif;
endseq;

ProcessingSequence discriminate:
    criterion = seisDis(lastminute);
    if criterion > 0.25:
        @probability = 4 per week
        recordIntArray(f, lastminute);
    endif;
    if criterion > 0.75:
        @probability = 1 per week
        ascent();
    endif;
endseq;

endmode;
```

Whales application

```
#1 Mission configuration
Mission:
    ParkTime: 20 days;
    ParkDepth: 1000 meters;

#2 Coordination of acquisition modes
Coordinator:
    DescentAcqModes:
        Whales every 10 minutes;
    ParkAcqModes:
        Whales every 3 hours;
    AscentAcqModes:
        Whales every 10 minutes;

#3 Definition of a short
#  acquisition mode
ShortAcqMode Whales:

#3.a Input
Input:
    sensor: Hydrophone;
    data: x[1024];

#3.b Variables
Variables:
    float presence;
    int timestamp;
    transmitFile f;

#3.c Sequences of instructions
ProcessingSequence identify:
    presence = whalesDetection(x);
    if presence > 0.2:
        @probability = 1 per day
        recordFloat(f, presence);
        timestamp = getTimestamp()
        recordInt(f,timestamp);
    endif;
endseq;

endmode;
```

2.3.5 Code generation

The code for the platform is generated from the application models and the platform model. In this subsection we describe the correspondence between the MeLa language and the generated code for which we have defined a suitable architecture. Before that, we describe in more detail the design of the instrument.

A Mermaid float contains two electronic boards, *i)* a control board that manages the actuators, the localization and the satellite communication and *ii)* an acquisition board for accessing sensors and processing data. They can communicate such that the acquisition board can request the float to ascend and send data through satellite communication, and the control board can provide information about the state of the float. The acquisition board is based on a single-core processor, and contains a real-time operating system with a priority based preemptive scheduling policy, allowing the applications defined with the MeLa language to be executed concurrently.

The code generation from models is illustrated in Figure 2.2. The mission configuration is used to generate a configuration file containing commands for the control board (*e.g.*, `stage 1500m 10000min`). The rest of the application is used to generate the code for the acquisition board. The coordinator is converted to a task containing a state machine (*i.e.*, a model of computation) reacting to messages sent by the control board and managing the execution of acquisition modes. Each acquisition mode is converted to a processing task containing the sequences of instructions defined in the MeLa language. Moreover, a sensor task receiving data from sensors (one task for one sensor) is configured for each acquisition mode using the sensor. A sensor task handles the data sent by a sensor, fills the input variables of processing tasks using the sensor, and triggers their execution when their input variable is full. Global variables, not presented in the figure, are used to share data between tasks.

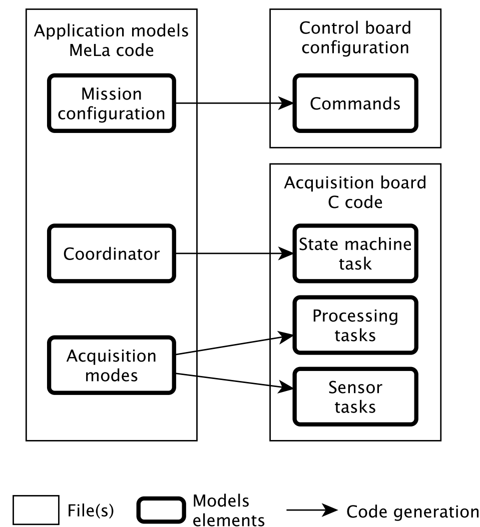


FIGURE 2.2 – Code generation from MeLa applications.

A priority of execution is assigned to each task. The highest priority is assigned to tasks with the shortest period, that is a rate-monotonic priority assignment [Liu and Layland, 1973]. For a continuous acquisition mode, the period of the task is the sampling period of the sensor multiplied by the size of the input variable. This priority is only used for the execution of the real-time sequence. The processing sequence of a continuous acquisition mode has a fixed low priority and is executed in background. For a short acquisition mode the period of the task corresponds to the periodicity defined in the coordinator.

For schedulability analysis, presented in the next section, we assume that tasks are independent. When possible, we implement the functions such that tasks can be executed concurrently, without interfering. If not, the execution time of functions must be estimated to take possible interferences into account.

2.3.6 Analysis

The analysis computes properties of the applications from the information contained in the application models and the platform model. Since the models of applications are created from the MeLa language, results of analysis can be reported to the developer with a reference to the MeLa code.

To determine if the tasks are schedulable their worst case execution time is computed from the model. Then, knowing their period of execution, their processor utilization rate (*e.g.*, 2%, 50%, 120%) is computed. Finally, we use the Liu and Layland utilization bound [Liu and Layland, 1973], a schedulability test for the rate monotonic scheduling algorithm. The Liu and Layland utilization bound gives the maximum processor utilization rate for a set of tasks (*e.g.*, 100% for one task, 83% for two tasks) that guarantee the schedulability of tasks. This test is only valid if the scheduling algorithm is optimal, that is if the tasks have a deadline equal to their period, and if they are independent from each other. Both conditions are verified since we made the assumption in the preceding section that tasks are independent.

The battery lifetime is estimated in several steps : *i)* power consumption of each acquisition mode is computed from each instruction, probabilities defined in conditions, sampling frequencies of sensors and periods defined in the coordinator, *ii)* power consumption of each state of the float is computed according to activated acquisition modes, input sensors used by acquisition modes, sleep time of the processor and actuators utilization for depth regulation (*e.g.*, ascent or descent), *iii)* energy consumption of each state is computed from their power consumption and their duration, which depends on probabilities of ascent request for the park state, *iv)* the energy for each float cycle is obtained by summing up the energy of each state, including the surface step which consumes energy for satellite transmission, *v)* knowing the battery capacity, the duration of a cycle, and the consumption of each cycle, the battery lifetime can be estimated.

The last property to estimate is the satellite transmission cost. To this end, the quantity of data recorded in files of type `transmitFile` is computed from variables passed as parameters of recording functions.

The analysis results are displayed to the developer with more or less detail. For

example, if the processor utilization rate is above the Liu and Layland utilization bound, an error is displayed with the instruction having the strongest impact on processor utilization. The same can be done for the battery lifetime and the satellite transmission costs. This allows the developer to identify parts of the MeLa code that contribute the most to processor utilization, energy consumption or satellite transmission costs.

2.3.7 Composition

The composition of applications is done at the model level. To be composed, the mission configuration of the two applications must be the same. Acquisition modes of each application are copied into the composed application with their politics of activation defined in the coordinator. Concurrent execution of acquisition modes is handled at the implementation level with schedulable tasks. The MeLa code generated from the composition of the *seismic* and *whales* applications is shown in table 2.2.

Table 2.2 – Composed application.

```
1  # 1. Mission configuration
2  Mission:
3      ParkTime: 10 days;
4      ParkDepth: 1500 meters;
5
6  # 2. Coordination of acquisition modes
7  Coordinator:
8      DescentAcqModes:
9          Whales every 10 minute;
10     ParkAcqModes:
11         Seismic;
12         Whales every 3 hours;
13     AscentAcqModes:
14         Whales every 10 minute;
15
16 # 3. Acquisition modes
17 ContinuousAcqMode Seismic:
18 # Content identical to the original application
19 endmode;
20
21 ShortAcqMode Whales:
22 # Content identical to the original application
23 endmode;
```

2.4 Validation

2.4.1 Introduction to validation

In this section, we show that our approach responds to the three challenges defined in the motivation section. We focus on technical metrics and present results of an experiment.

Tools and meta-models presented on figure 1 are implemented in Java. The syntax of the MeLa language and associated tooling are created with ANTLR [Parr and Quong, 1995]. The generated code has been deployed on an acquisition board in a controlled environment. The experimental setup is described below.

2.4.2 Experimental setup

The *seismic* and *whales* applications are tested on a test bench (figure 2.3). The acquisition board (1) is powered with a 9 V alkaline battery (2). A computer (3) emulates seismic and whales signals, sent to the acquisition board with an audio sound card (4). It is also used to monitor the execution of applications through a serial communication port (5), and record the voltage of the battery with an Arduino board (6).

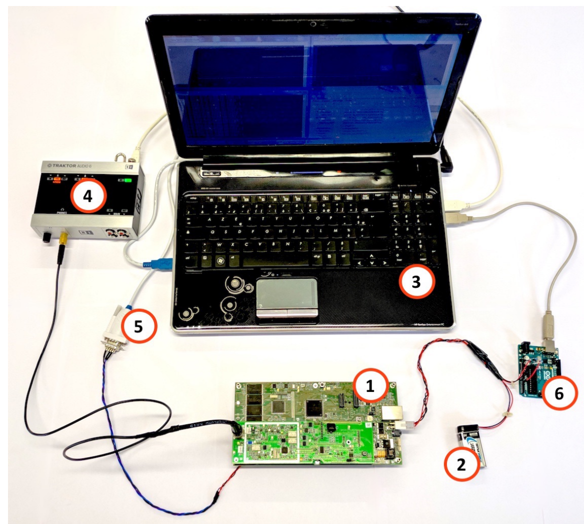


FIGURE 2.3 – Experimental setup.

Seismic events are emulated with a 1 Hz signal, or 2 Hz for major events triggering the ascent of the float. Each event has a fixed duration of 30 seconds. We generate four events every hour. One of these events is randomly chosen to be a major event. The algorithm to detect seismic events `seisDetection` (line s29 in table 2.1) is implemented as an absolute average of the last forty samples send by the hydrophone, and stored in the input variable (line s17). The average is compared to a threshold. When the value of the average passes under the threshold, the `detect` boolean is set to `true` (detection on the falling edge of the average). This triggers the execution of the processing sequence that identifies if the signal has a seismic origin. The `seisDis` algorithm is implemented as a Fourier transforms processing the last minute of signal (the `lastminute` variable). This algorithm returns a criterion which represents the level of confidence of the seismic origin of the signal. If the spectrum amplitude at 1 Hz or 2 Hz is above a threshold, the criterion is set respectively to 0.5 and 0.9. If the criterion is above 0.25, the last minute of signal is recorded, and if the criterion is above 0.75, the application requests

an ascent (lines s38, s42).

Whales events are emulated with a frequency of 10 Hz and a lower amplitude than seismic events so that the signals do not interfere with the detection of seismic events. As for the *seismic* application, whales events have a fixed time of 30 seconds. The `whalesDetection` algorithm is also implemented using a Fourier transform, processing the signal acquired during a short time. If the spectrum amplitude at 10 Hz is above a threshold, the algorithm returns a probability of presence equal to 1, triggering the recording of the value and of a timestamp (line w32).

The *seismic* application is executed continuously during the park state and the *whales* application is executed every 30s during the descent, park and ascent state. The behavior of the float is simulated by the acquisition board. We define a park time of one hour, shortened if the *seismic* application request to ascend. The ascent time and the descent time are fixed to 2 minutes and the surface time to 1 minute.

2.4.3 Functional validation

The two applications have been tested, first independently, and then after composition to verify if both applications behave as expected. The applications have worked correctly, seismic and whales events have been recorded. This shows that the language can be used to develop applications for the float (C1) and that several applications can be deployed on the same float and share its functionalities (C3).

2.4.4 Reduction of expertise

When an application is written in MeLa, the developer does not have to consider the control board and the acquisition board. The development of acquisition modes does not require to think about embedded software concerns, for example defining tasks, their initialization, their execution priority, the way they are started and stopped or the synchronization between tasks receiving data from sensors and tasks processing the data. Thus, the MeLa language hides several embedded software concerns.

Table 2.3 illustrates the reduction of expertise given by the MeLa language compared to the generated code. This table presents the generation of an acquisition mode into a sensor task and a processing task, as shown in figure 2.2. For that purpose, we use a simple application that reads data from the hydrophone and computes an average. The `Input` part of the MeLa code generates the sensor task. This task waits for data from the hydrophone. When the hydrophone is ready to send data, it sends a signal to the processor (a hardware interrupt) that triggers the execution of the sensor task (line c3) [3].

The input variable (line m5) is generated as two array of data (lines c19-c24). One is filled by the sensor task (line c4), while the other one is processed by the processing task (line c31). When the `x_simple_fill` array is full (line c6), the array is switched with `x_simple_process` (lines c7-c13) and the execution of the processing task is triggered with a signal (line c14). The `RealTimeSequence` part of the acquisition mode is

3. Reference to line numbers are given with an *m* for the MeLa code and a *c* for the C code (*e.g.*, m1, c1).

Table 2.3 – MeLa code compared to generated code.

MeLa code

```
ContinuousAcqMode Simple:

Input:
    sensor: Hydrophone;
    data: x[10];

Variables:
    int y;

RealTimeSequence:
     y = mean(x, 10);
endseq;

endmode;
```

C

```
// Sensor task - hydrophone.c
void hydrophone_task(void * parameters){
  wait_ExtInt (_adconverter.drdy_eint_id);
  x_simple_fill->data[x_simple_fill->i]
                                = read_sample();
  x_simple_fill->i += 1;
  if (x_simple_fill->isFull) {
      if (x_simple_fill == &x_simple_buff_1) {
          x_simple_process = &x_simple_buff_1;
          x_simple_fill = &x_simple_buff_2;
      } else {
          x_simple_process = &x_simple_buff_2;
          x_simple_fill = &x_simple_buff_1;
      }
      semaphoreSignal(simple_Semaphore);
  }
}

// Global variables - global.c
int32_array_t* x_simple_fill;
int32_array_t* x_simple_process;
int32_t x_simple_1[10];
int32_array_t x_simple_buff_1
              = {x_simple_1, 0, 10, false};
int32_t x_simple_2[10];
int32_array_t simple_x_buff_2
              = {x_simple_2, 0, 10, false};

// Processing task - simple_task.c
static int32_t y;
void simple_task(void* parameters){
    while(1) {
        semaphoreWait(simple_Semaphore);
        y = mean(x_simple_process, 10);
    }
}
```

converted in a processing task (lines c28-c33), that is waiting for the sensor task (line 30). The `Variables` part is converted to local variables contained in the task (line 27). In the application written in MeLa, the developer only defines the input sensor, the input variable, and the algorithm to use. She can focus on the behavior of applications rather than on embedded software concerns.

Another way to estimate the reduction expertise is to compare the amount of code to write in MeLa, with the amount of generated code, that would be written manually. Looking at the total number of lines of the composed application, one has to write 90 lines of code in MeLa, while 600 lines must be written to develop the application with the C language.

By hiding embedded software concerns and reducing the amount of code to write, the MeLa language allows oceanographers to develop applications for the float by them-

selves (C1). Moreover, generating a code tailored for the MeLa applications helps to produce efficient and reliable applications (C2). For example, in MeLa the sensors are automatically shut down when they are not used. In C, this behavior must be written by the developer.

2.4.5 Analysis validation

In this subsection we show that analysis results are consistent with experiments so that they can be used to produce efficient and reliable applications (C2). We compare the estimation from the model with measurement performed on the acquisition board. We do this comparison for the *seismic* and the *whales* applications independently and for the composition of both applications.

For the model estimation, probabilities defined in the applications must be coherent with the expected behavior of the deployed applications. Thus, for our experimental setup, the probability to detect and record a seismic event is set to 4 per hour (lines s31, s39 in table 2.1), and the probability of ascent request is set to 1 per hour (line s43). For the whales application, the probability of presence of whales is set to 10 per hour (line w33).

We measure the battery lifetime by measuring the voltage of the battery. When the voltage passes under 6 V the battery is considered as discharged. Instead of giving a cost for satellite transmission, we measure the size of files that would have been transmitted. We do not measure the utilization of the processor since the processor utilization is too low in our experiment to be measured efficiently.

Results for the battery lifetime in hours, and amount of recorded data in bytes and kilobytes per hour, are presented in table 2.4. The *seismic* and composed applications have similar power consumption. This is because both *seismic* and *whales* applications use the same sensor and the sensor is always switched on for the *seismic* application. For the *whales* application alone, the sensor is regularly switched off, giving more autonomy.

Differences of 10% are observed between estimations and measurements of the battery lifetime. For recorded data, the estimations fit well with measurements because probabilities annotated in the applications are consistent with the reality. The precision of these estimations are enough to detect if an application will drain the battery rapidly (*e.g.*, 3 years instead of 5 years), or if the amount of transmitted data will exceed the budget limits (*e.g.*, 20 MB instead of 10 MB).

Table 2.4 – Model estimations compared to measurements.

Application	**Battery lifetime**		**Recorded data**	
	Estimation	Measure	Estimation	Measure
Seismic	14 h	15 h	35 kB/h	36 kB/h
Whales	22 h	20 h	79 B/h	72 B/h
Composed	14 h	13 h	35 kB/h	32 kB/h

2.4.6 Produce efficient applications

In table 2.5 we show different results of scheduling analysis for three applications.

(1) The *excessive* application can be viewed as a first attempt of the *seismic* application. For this attempt, the discrimination algorithm is put in the real-time sequence, and the input variable is set to a size of 1 instead of 40 (*i.e.*, `data: x[1];`). The analysis displays an error to the developer telling him that the processor usage is above the maximum allowable and showing the responsible instruction. At this point some guidelines are necessary to help the developer to edit the application. There are only three possible choices, *i)* put the algorithm in a processing sequence and use a detection algorithm to trigger the processing sequence, *ii)* increase the size of the input variable to give more time to the processing, but it can also increase the processing time, *iii)* chose another algorithm in the library.

(2) The second analysis result is for the *seismic* application of our example. The developer has followed the first and second guidelines, such that the processor usage is reduced to almost zero.

(3) The third result is for the composed application. One can notice that the maximum processor usage is 83% which corresponds to the Liu and Layland utilization bound for 2 tasks.

In addition, to ensure that applications will work correctly on the instrument, the analysis results allow a developer to try different configurations. For example, she can try to record the raw acoustic signals containing the presence of whales and see the impact on battery lifetime and satellite transmission costs. Thus, the analysis results help the developers to produce efficient and reliable applications (C2).

Table 2.5 – Scheduling analysis results

```
# 1. Excessive application
 Processor usage during PARK:
   Error: 140 % > 100 %
     detect = seisDiscrimination(x): 140%


# 2. Seismic application
 Processor usage during PARK:
   Valid: 0,4 % < 100 %


# 3. Composed application
 Processor usage during DESCENT:
   Valid: 0,03 % < 100 %

 Processor usage during PARK:
   Valid: 0,43 % < 83 %
   Seismic continuously : 0,4 %
   Whales with period 30s : 0,03 %

 Processor usage during ASCENT:
   Valid: 0,03 % < 100 %
```

2.5 Limitations and perspectives

The MeLa language has a limited expressiveness. For example, the politics of acquisition managed by the coordinator are limited to few concepts (*i.e.*, periodic or continuous), a developer could want to use other kinds of sensors or to choose the sampling frequency of a hydrophone. However, the approach is flexible enough to add new features to the language.

The ability for applications to adapt to the environment and to have complex interactions with the float is currently missing. For example, when a whale is detected, the monitoring period, the sensor sampling frequency or the algorithm parameters could be changed. Additionally, an application could ask the float to go to a specific depth or block any depth regulation during a certain amount of time. Several approaches exist to handle the adaptation of applications running concurrently on a same device with possible conflicts between them [Kakousis et al., 2010] but they must be adapted to our problem.

Additional analysis capabilities could be added with new models of analysis. For example, analysis of volatile memory usage could be added. Moreover, more precise analysis methods could improve accuracy of estimations. However, the main limit for accuracy lies in the definition of probabilities of execution by the developers. These probabilities are important to estimate the quantities of recorded data or the battery lifetime, but the developer may enter wrong information to the model, so that estimations will also be wrong. Simulation, based on experimental data recovered from experiments, could give accurate estimation ; moreover, simulation could be used for functional validation of applications. Probabilities could also be measured on the instrument, after deployment, and then be used to correct the model.

The MeLa language is specific for programming the instrument, but is not conceived to create new algorithms. Thus, capabilities to design algorithms for the float could be added to MeLa. The algorithms could be organized in different categories with specific constraints, for example detection algorithms should behave as an impulse function to trigger the execution of processing sequences. Moreover, capabilities to develop *Deep Learning* algorithms could be added. They are well suited for classification problems, but deploying them on a constrained device is challenging [Lane et al., 2017].

Finally, the MeLa language could be extended to other domains of applications by adding features to handle actuators or displaying devices that have specific constraints. This would allow the development of a wide range of applications in the embedded software domain while keeping the efficiency and reliability demonstrated this article.

2.6 State of the art

We compare our work with approaches related to the development of embedded software.

Programming languages like Scilab-Xcos[4] and Matlab-Simulink[5] are widely used in different domains and focus on development of algorithms and modeling of physical systems. Code for embedded systems can be generated from these algorithms but they do not incorporate models used in embedded systems such as tasks, thus they cannot be used to develop entire applications.

Low-level programming languages such as C or Ada, and real-time operating systems [Gaur and Tahiliani, 2015], allow to develop applications that use the platform efficiently [Pereira et al., 2017], but they need a specific expertise to be used. Our contribution generates such a code to be implemented on the platform. Analysis of applications developed with these languages rely on tools that generate models from the code. The models can be annotated with tailored measurements [Iyenghar and Pulvermüller, 2018], [la Fosse et al., 2018], or relies on generic models of processors [Ferdinand and Heckmann, 2004]. The tools need a specific expertise to be used [Woodside et al., 2007], our approach separates developers concern from analysis, allowing them to focus on their applications.

Some operating systems for embedded systems consider resource consumption at runtime [Lorincz et al., 2008], [Sorber et al., 2007]. They target energy harvesting systems (*e.g.*, systems with solar panels). These systems have an energy budget that changes over time (*e.g.*, there is less energy during cloudy days). To handle this, the quality of algorithms is degraded depending on the available energy of the system at execution time. The floats do not have such constraints since they do not incorporate energy harvesting systems. Moreover, in these approaches the developer does not have an estimation of resource consumption during the development of applications.

Modeling languages like UML-MARTE [Group, 2011] or AADL [Feiler et al., 2006] are conceived for modeling different kinds of embedded systems, software and hardware included. They offer generic concepts for these domains. They are often used for the design of systems from high-level specifications that are refined several times until the implementation. Modeling languages offer high level abstractions that are too generic for our developer. Moreover, analysis tools on which they rely, like UPPAAL [Larsen et al., 1997] or TimeSquare [DeAntoni and Mallet, 2012], require a specific expertise to be used.

DSL like CPAL [Navet and Fejoz, 2016] and MAUVE [Gobillot et al., 2018] are dedicated to programming cyber-physical systems and robots with a component-based approach. In these languages developers define components with desired inputs and outputs and a state machines to describe their internal behavior. Our acquisition modes can be viewed as components tailored for acquisition of data from sensors, thus, we offer a more specific abstraction to developers. For these languages it is up to the developer to measure the execution times of components and to put this information in the developed application. In our approach this information is hidden to developers in a platform model allowing them to focus on their applications. Moreover these approaches do not consider energy consumption or recording of data, which are critical for our instrument.

4. https ://www.scilab.org/
5. https ://www.mathworks.com/

2.7 Conclusion

In this paper, we have proposed a *Model Driven Engineering* approach to allow oceanographers from different specialities to develop applications for an autonomous float. We have presented a *Domain Specific Language* to allow them to develop their own applications without the help of experts in embedded systems. Estimation of battery lifetime, costs of satellite transmission and verification of execution time constraints helps the developers to write reliable and efficient applications. The application models can be composed such that several applications developed independently can be installed on the same instrument. We have validated the approach with technical metrics and an experiment on a test bench.

In the long term, we envision a float that can be reprogrammed at distance. This, associated with the MeLa language would allow to use the float as a real experimental platform where developers could try several applications. But, this requires adapted over-the-air programming methods that save the use of the satellite communication, which is challenging because of the very high latency of this kind of network (about one second).

Références

Baumgartner, M. F., Stafford, K. M., and Latha, G. (2018). Near real-time underwater passive acoustic monitoring of natural and anthropogenic sounds. In Venkatesan, R., Tandon, A., D'Asaro, E., and Atmanand, M. A., editors, *Observing the Oceans in Real Time*, pages 203–226. Springer International Publishing, Cham.

DeAntoni, J. and Mallet, F. (2012). Timesquare : treat your models with logical time. In *TOOLS*.

Feiler, P., Gluch, D., and Hudak, J. (2006). The architecture analysis & design language (aadl) : An introduction.

Ferdinand, C. and Heckmann, R. (2004). ait : Worst-case execution time prediction by static program analysis. In Jacquart, R., editor, *Building the Information Society*, pages 377–383, Boston, MA. Springer US.

Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional, 1st edition.

Gaur, P. and Tahiliani, M. P. (2015). Operating systems for iot devices : A critical survey. *2015 IEEE Region 10 Symposium*, pages 33–36.

Gobillot, N., Lesire, C., and Doose, D. (2018). A design and analysis methodology for component-based real-time architectures of autonomous systems. *Journal of Intelligent and Robotic Systems*, pages 1–16.

Gomaa, H. (2016). *Real-Time Software Design for Embedded Systems*. Cambridge University Press.

Gould, W. J. (2005). From swallow floats to argo—the development of neutrally buoyant floats. *Deep Sea Research Part II : Topical Studies in Oceanography*, 52(3) :529 – 543. Direct observations of oceanic flow : A tribute to Walter Zenk.

Group, O. M. (2011). Uml profile for marte : Modeling and analysis of real-time embedded systems.

Iyenghar, P. and Pulvermüller, E. (2018). A model-driven workflow for energy-aware scheduling analysis of iot-enabled use cases. *IEEE Internet of Things Journal*, 5 :4914–4925.

Kakousis, K., Paspallis, N., and Papadopoulos, G. A. (2010). A survey of software adaptation in mobile and ubiquitous computing. *Enterprise Information Systems*, 4(4) :355–389.

Kent, S. (2002). Model driven engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 286–298, London, UK, UK. Springer-Verlag.

la Fosse, T. B., Mottu, J.-M., Tisi, M., Rocheteau, J., and Sunyé, G. (2018). Characterizing a source code model with energy measurements. In *MeGSuS@ESEM*.

Lane, N. D., Bhattacharya, S., Mathur, A., Georgiev, P., Forlivesi, C., and Kawsar, F. (2017). Squeezing deep learning into mobile and embedded devices. *IEEE Pervasive Computing*, 16(3) :82–88.

Larsen, K. G., Pettersson, P., and Yi, W. (1997). Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1 :134–152.

Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20 :46–61.

Lorincz, K., Chen, B.-r., Waterman, J., Werner-Allen, G., and Welsh, M. (2008). Resource aware programming in the pixie os. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 211–224, New York, NY, USA. ACM.

Ma, B. B. and Nystuen, J. A. (2005). Passive acoustic detection and measurement of rainfall at sea. *Journal of Atmospheric and Oceanic Technology*, 22(8) :1225 – 1248.

Matsumoto, H., Jones, C. R., Klinck, H., Mellinger, D. K., Dziak, R. P., and Meinig, C. (2013). Tracking beaked whales with a passive acoustic profiler float. *The Journal of the Acoustical Society of America*, 133 2 :731–40.

Mussbacher, G., Amyot, D., Breu, R., Bruel, J.-M., Cheng, B. H. C., Collet, P., Combemale, B., France, R. B., Heldal, R., Hill, J. H., Kienzle, J., Schöttle, M., Steimann, F., Stikkolorum, D. R., and Whittle, J. (2014). The relevance of model-driven engineering thirty years from now. In *MoDELS*.

Navet, N. and Fejoz, L. (2016). Cpal : high-level abstractions for safe embedded systems. In *DSM@SPLASH*.

Nolet, G., Hello, Y., van der Lee, S., Bonnieux, S., Ruiz, M. J. C., Pazmino, N. A., Deschamps, A., Regnier, M. M., Font, Y., Chen, Y. J., and Simons, F. J. (2019). Imaging the Galápagos mantle plume with an unconventional application of floating seismometers. In *Scientific Reports*.

Parr, T. J. and Quong, R. W. (1995). Antlr : A predicated- ll(k) parser generator. *Softw., Pract. Exper.*, 25 :789–810.

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. a. P., and Saraiva, J. a. (2017). Energy efficiency across programming languages : How do energy, time, and memory relate ? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, page 256–267, New York, NY, USA. Association for Computing Machinery.

Roemmich, D., Johnson, G., Riser, S., Davis, R., Gilson, J., Owens, W., Garzoli, S., Schmid, C., and Ignaszewski, M. (2009). The Argo Program : Observing the Global Ocean with Profiling Floats. *Oceanography*, 22(2) :34–43.

Schmidt, D. C. (2006). Guest editor's introduction : Model-driven engineering. *Computer*, 39 :25–31.

Sorber, J., Kostadinov, A., Garber, M., Brennan, M., Corner, M. D., and Berger, E. D. (2007). Eon : A language and runtime system for perpetual systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 161–174, New York, NY, USA. ACM.

Sukhovich, A., Bonnieux, S., Hello, Y., Irisson, J. O., Simons, F. J., and Nolet, G. (2015). Seismic monitoring in the oceans by autonomous floats. In *Nature communications*.

Woodside, C. M., Franks, G., and Petriu, D. C. (2007). The future of software performance engineering. *Future of Software Engineering (FOSE '07)*, pages 171–187.

Chapitre 3

MeLa : un langage de programmation pluridisciplinaire

Contents

3.1 Introduction 48
3.1.1 Context 48
3.1.2 Motivations and objectives 49
3.2 A programming language based on models 50
3.2.1 Models for programming 50
3.2.2 Description of MeLa 52
3.2.3 Mermaid float architecture 55
3.2.4 Code generation 56
3.2.4.1 Overview 56
3.2.4.2 Priority rules 57
3.2.4.3 Benefits of MeLa for embedded software programming 57
3.2.5 Application verification 58
3.2.5.1 Static analysis of applications 58
3.3 Developing with MeLa 62
3.4 Experiments 64
3.4.1 Detection of earthquakes 64
3.4.1.1 Scientific context 64
3.4.1.2 The seismic detection algorithm 64
3.4.1.3 Implementation with MeLa 65
3.4.1.4 Evaluation of the algorithm 66
3.4.2 Detection of blue whales 66
3.4.2.1 Scientific context 66
3.4.2.2 The blue whale detection algorithm 67
3.4.2.3 Implementation with MeLa 68
3.4.2.4 Evaluation of the algorithm 69
3.5 Discussion and conclusion 70

Références **73**

Ce chapitre correspond à l'article publié dans la revue Sensors : *S. Bonnieux, D. Cazau, S. Mosser, M. Blay-Fornarino, Y. Hello and G. Nolet, "MeLa : a programming language for a new multidisciplinary oceanographic float", Sensors, 20, 6081, 2020, doi :10.3390/s20216081.* En complément de l'article de conférence OCEANS 2019, cet article est plutôt axé sur l'utilisation du langage. , les méthodes d'analyse des applications y sont également plus détaillées. Cet article place MeLa dans le cadre d'un processus de développement des applications, notamment en proposant une méthode de développement originale rendue possible grâce aux fonctionnalités de MeLa. Cette méthode est illustrée avec deux applications, une pour la détection de séismes et l'autre pour la détection de baleines bleues. Les avantages de MeLa pour programmer le flotteur Mermaid sont discutés tout au long de cet article.

MeLa : A Programming Language for a New Multidisciplinary Oceanographic Float

Sébastien Bonnieux, Dorian Cazau, Sébastien Mosser, Mireille Blay-Fornarino, Yann Hello and Guust Nolet

Résumé

Dans les océans, à 2000 mètres de profondeur, on peut entendre l'activité biologique, sismologique, météorologique et anthropique. La surveillance acoustique des océans à l'échelle mondiale et sur de longues périodes de temps pourrait apporter des informations importantes pour divers domaines scientifiques. Le projet Argo surveille les propriétés physiques des océans à l'aide de flotteurs autonomes, et certains sont équipés d'un hydrophone. Ces derniers ont une capacité de transmission par satellite limitée, les données acoustiques doivent donc être traitées à bord de l'instrument. Or, le développement d'algorithmes de traitement du signal pour ces instruments nécessite une réelle expertise en matière de logiciels embarqués. Pour réduire ce besoin, nous avons développé un langage de programmation appelé MeLa. Ce langage dissimule plusieurs aspects des logiciels embarqués grâce à des concepts de programmation spécialisés. Il utilise des modèles pour calculer la consommation d'énergie, l'utilisation du processeur et les coûts de transmission des données dès le début du développement des applications. Cela permet de choisir une stratégie de traitement des données avec un impact minimal sur les performances. Les simulations sur ordinateur permettent de vérifier les performances des algorithmes avant leur déploiement sur l'instrument. Pour montrer les capacités du langage MeLa, nous avons implémenté un algorithme de détection d'ondes sismiques P et un algorithme de détection de cris de baleines bleues (D calls). Ce sont les premiers efforts vers une surveillance pluridisciplinaire des océans, qui peut s'étendre au-delà des applications acoustiques.

Abstract

At 2000 meters depth in the oceans one can hear biological, seismological, meteorological and anthropogenic activity. Acoustic monitoring of the oceans at a global scale and over long periods of time could bring important information for various sciences. The Argo project monitors physical properties of the oceans with autonomous floats, some of which are also equipped with a hydrophone. These have a limited transmission bandwidth requiring acoustic data to be processed on board. However, developing signal processing algorithms for these instruments requires a real expertise in embedded software. To reduce such need, we have developed a programming language called MeLa. The language hides several aspects of embedded software with specialized programming concepts. It uses models to compute energy consumption, processor usage and data transmission costs early during the development of applications ; this helps to choose a strategy of data processing that has a minimum impact on performances. Simulations on a computer allow verifying the performance of the algorithms before their deployment on the instrument. We have implemented a seismic P wave detection and a blue whales D call detection algorithm with the MeLa language to show its capabilities. These are the first efforts toward multidisciplinary monitoring of the oceans, which can extend beyond acoustic applications.

3.1 Introduction

3.1.1 Context

Scientists all over the globe are permanently monitoring how our planet is changing. Knowing how much heat is stored in the ocean, how fast the sea levels are rising and sea ice is melting, where living ecosystems are migrating in response to anthropic activity, are only a very few of the many key questions for understanding the current state and changes in the ocean and climate. This information is critical for assessing and confronting oceanic and atmospheric changes associated with global warming and they can be used by decision-makers, environmental agencies, the general public, and in measuring our responses to environmental directives.

Oceans have been monitored since the 19th century [Wüst, 1964]. The first oceanographic campaigns were done from ships with manually handled instruments. When electronics and batteries were emerging, instruments started to become autonomous [Marcelli et al., 2011]. For moored instruments like moored lines [Meindl, 1996, Venkatesan et al., 2018] or *Ocean Bottom Seismometers* (OBS) [Hello et al., 2019], they can now be deployed at sea for periods up to several months or years. However, the elevated costs of maintenance reduce our ability to deploy them globally. Alternatively, remote sensing from satellites [Devi et al., 2015] allows working at a global scale but has only access to the ocean's surface and has relatively low spatial and temporal resolution in comparison to *in situ* sensors.

Nevertheless, satellite communication systems provide the necessary technology to locate and transmit in near real-time data collected by autonomous underwater vehicles.

Such vehicles include profiling floats [Davis et al., 1992, Gould, 2005] and wave gliders [Manley and Willcox, 2010]. Both have different advantages and drawbacks depending on the usage. Profiling floats are widely used in the Argo [1] program with thousands of floats deployed world-wide [Roemmich et al., 2009]. They monitor the temperature and salinity from the surface to a depth of 2000 meters to study the climate.

Most of the floats follow the same operational cycle : 1) they descend to a depth programmed by the operator, 2) they park at this depth during several days or weeks and drift with currents, 3) they ascend to the surface, and 4) they measure their position with a *Global Positioning System* (GPS) receiver and send their data through a satellite link. One operational cycle is called a dive. The depth is regulated by changing the float density using an external bladder filled with oil. Measurements are done during any step of the dive by sensors integrated into the float to measure conductivity, temperature, depth, chlorophyll, nitrate, as well as acoustic signals and others.

In more recent work, *Underwater Passive Acoustic* (UPA) measurements have been integrated into profiling floats for different monitoring applications such as whale tracking in marine ecology [Matsumoto et al., 2013] and above-surface wind speed or rainfall estimations in marine meteorology [Riser et al., 2008, Yang et al., 2015]. Among recent drifting floats, a breakthrough has recently been obtained by seismologists with the *Mobile Earthquake Recording Device in Marine Areas by Independent Divers* (Mermaid), an autonomous float equipped with a hydrophone and a smart acquisition board able to recognize seismic sounds [Sukhovich et al., 2015]. The recognition of seismic sounds allows it 1) to trigger the ascent of the float in order to get a precise estimation of the recording position, and 2) to transmit only relevant seismogram data through the low bandwidth satellite link. So far, 60 floats have been deployed to image mantle plumes beneath hotspots in the Pacific Ocean.

3.1.2 Motivations and objectives

In this paper, we aim to develop a multidisciplinary version of the Mermaid float making it possible to combine different monitoring applications during the same campaign. Although we focus on UPA monitoring, the float is not limited to acoustics and can integrate other sensors.

The main motivation of our work is to enable scientists to write signal processing applications for the instrument. Indeed the sensors, and more especially the acoustic, generates high volume of data (*e.g.*, a 5-minute record at 78.1 kHz produces 70 MB of data, and 7 TB for one year). These data can be stored by the floats, but due to cost effectiveness, the floats are usually not recovered from the oceans, as it is the case with most Argo floats. The satellite communication system has a very limited bandwidth and is not capable of transmitting such amount of data. Moreover, many applications such as monitoring of earthquakes or volcanic activity, require data transmission in (quasi) real time. A generic algorithm able to handle different signal processing applications from different domains does not exist. Even machine learning algorithms have different

1. Argo website : `https://argo.ucsd.edu/about/`

architectures, depending on the application. Thus the Mermaid cannot be configured with just a few parameters, it must be programmed with fully fledged applications.

However, developing signal processing applications to be embedded in an instrument such as Mermaid is challenging for the following four reasons :

1. Embedded software programming requires specific technical skills, and can be off-putting for less technically skilled scientists who will have to learn C language programming and know specialized technical details about the instrument such as the operation of the real time operating system, micro-controller, sensors, etc.
2. The embedded applications must comply with the limited resources of the instrument. Otherwise they may not behave as expected, induce elevated costs of data transmission, or reduce considerably the instrument lifetime.
3. The embedded applications must be reliable, without software bugs, and efficient, with a minimum impact on the instrument resources. Any miss-conceived code may compromise the instrument that is not directly accessible when deployed in the oceans. Less technically skilled developers are more prone to write miss-conceived code.
4. The embedded applications developed independently and installed on the same instrument must not interfere with each other. Whether the applications are installed alone or with other applications their behavior must not change.

To overcome these challenges, we have developed a programming language dedicated to the Mermaid. This language is designed to meet the needs of signal processing experts and does not require embedded software programming skills. The language is called MeLa, for Mermaid Language, and is presented in the next section.

3.2 A programming language based on models

3.2.1 Models for programming

Scientists use models to understand the world, for example with climate models. Engineers use models to develop new systems (i.e., a bridge, a computer program). These models are specific to a domain of expertise, for example electronic engineers use models of resistances or transistors. The models are assembled together to develop a system, for example an electronic circuit. A language, graphical or textual that allows to edit the models, it is usually called a *Domain Specific Language* (DSL) [Fowler, 2010]. The MeLa language is a DSL dedicated to the development of signal processing algorithms for the Mermaid floats. Models can be used in several ways in order to respond to the challenges introduced in the first section.

First, models allow us to represent systems at several levels of abstraction. In software engineering assembly instructions are low-level models, whereas functions or tasks of an operating system are models with a high-level of abstraction. The MeLa language gains in abstraction with models dedicated to the development of applications for the instrument. Instead of programming applications with tasks, which need an expert in

embedded software, the MeLa language offers models called *acquisition modes*. Using these models, the developer does not have to manage the execution priority of tasks, or the synchronization of execution with other tasks. Instead, developing an application with an acquisition mode only requires that the user defines the input sensor and the sampling frequency. This high level of abstraction allows developers that are not embedded software experts to write applications for the instrument (challenge 1).

Second, models can be used to compute properties of the developed system before building it. The models of the MeLa language allow to estimate properties of the developed applications such as the lifetime of batteries, the cost of satellite transmission and the processor usage, but other properties can also be incorporated in the models. Since the models are associated with the programming language, the estimations can be linked to the content of the applications. For example, the estimations can indicate which function uses most of the processor time. Thus, using models allows to verify that the instrument limits are not exceeded (challenge 2). Moreover, these estimations are computed during the writing of the applications, improving productivity. This would not be the case if measurements were done on a real instrument with specialized equipment ; it would require expertise and time to realize the measures and interpret them.

Third, models are used to generate the specific embedded software specific code to program the instrument. The code generation process is managed by a tool integrated in the language, such as a compiler. The transformation rules to generate the code are defined by embedded software experts, this ensures to have a reliable and efficient code (challenge 3). For example, the acquisition modes generate several tasks, with their synchronization mechanism and execution priority. The embedded software code would not be reliable and efficient if written directly by a non-expert. It is also possible to generate several specific codes to program different platforms. In our case, we also generate code to execute the applications on a personal computer, it allows scientists to settle the applications and verify that they behave as expected.

Finally, models allow to compose (*i.e.*, combine) applications that have been developed independently (challenge 4). The composition of applications consists mostly of verifying that applications are compatible in terms of both used computational resources (*e.g.*, they cannot use more resources than what is available) and sensors (*e.g.*, they cannot share a sensor if their configurations on this sensor differ). The concurrent execution of applications is also managed by the operating system at the time of execution time.

Figure 3.1 illustrates how models are used in MeLa :

1. Scientists write applications in MeLa, which avoids embedded software concerns. The applications written in a text file are transformed into models (implemented as Java objects) with a parser.
2. The analysis verifies that the limits of the floats are not exceeded and results are returned to developers allowing them to identify problems and rectify them.
3. The code for simulating the applications on a computer is generated to settle them and verify that they behave as expected.
4. Composition combines several applications to install on the same float after verifying that they are not incompatible.

5. The embedded software code to program the float is generated.

More details about the use of models in the MeLa language can be found in [Bonnieux et al., 2019].

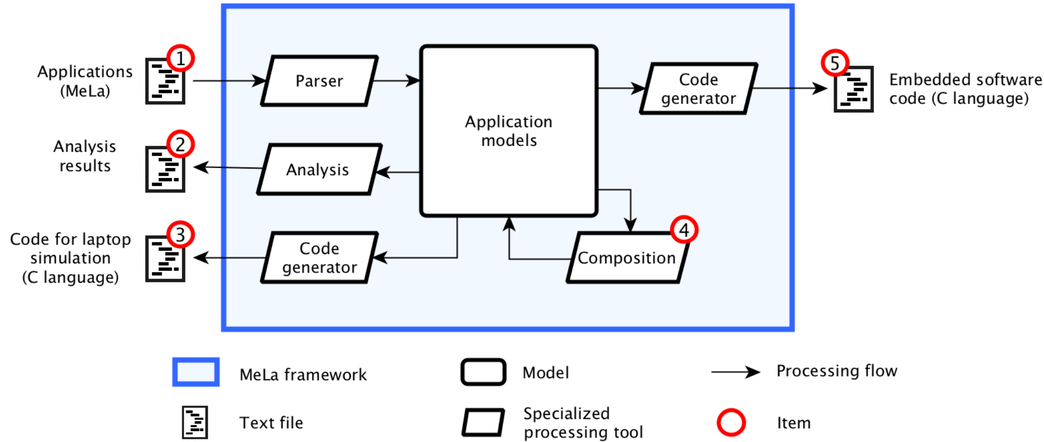


FIGURE 3.1 – Models used in MeLa.

3.2.2 Description of MeLa

The MeLa language is both imperative and declarative. The imperative part of the language allows writing the content of the algorithms with sequences of instructions, conditions and loops. The declarative part allows declaring the depth and duration of the float dives, when to execute an application, and which sensor an application has to use. The MeLa language is implemented with ANTLR [Parr and Quong, 1995], a DSL dedicated to the creation of other languages. The models behind MeLa are written in Java, an object-oriented programming language. The code to program the instrument generated from the models is in C language.

An example of an application written with MeLa is given in the table 3.1. This is a very simplified version of a seismic detection application. The mission configuration part (lines 1-4) allows the developer to define the depth and duration of a dive. The coordinator (lines 7-9) allows her to define when to execute an algorithm during the descent, parking or ascent steps of the dive. In this example she has chosen to run the *Seismic* algorithm only during the parking stage, because the ascent and descent are too noisy for seismic monitoring.

The algorithmic part of an application is contained in what we called acquisition modes. The developer has to define the input of the acquisition mode. There are two parameters for the input, 1) the sensor with its sampling frequency and 2) the array name and size in which the sensor puts the samples. In the example the sensor is the low frequency hydrophone with a sampling frequency of 40 Hz (line 16). The samples are put

Table 3.1 – Application example. The functions 'seisDetection' and 'seisDiscrimination' have been left to keep the example short, they do not exist in the language, the real algorithm for seismic detection is presented in section 3.4.1.

```
# 1. Mission configuration
Mission:
    ParkTime: 10 days;
    ParkDepth: 1500 meters;

# 2. Coordination of acquisition modes
Coordinator:
    ParkAcqModes:
        Seismic;

# 3. Definition of a continuous acquisition mode
ContinuousAcqMode Seismic:

# 3.a. Input
Input:
    sensor: HydrophoneBF(40);
    data: x(40);

# 3.b. Variables
Variables:
    ArrayInt lastminute(2400);
    Bool detect;
    Float criterion;
    File f;

# 3.c. Sequences of instructions
RealTimeSequence detection:
    append(lastminute, x);
    detect = seisDetection(x);
    if detect:
        @probability = 10 per week
        call discriminate;
    endif;
endseq;

ProcessingSequence discriminate:
    criterion = seisDiscrimination(lastminute);
    if criterion > 0.25:
        @probability = 4 per week
        record(f, lastminute);
    endif;
endseq;

endmode;
```

in an array called `x` containing 40 samples (line 17). The samples are processed when the array is fully filled by the sensor.

There are two kinds of acquisition modes, the `ContinuousAcqMode` for which packets of data are recovered continuously, in a streamed way, and the `ShortAcqMode` for which single packet of data is recovered. The latter can be executed periodically with a time interval defined in the coordinator (*e.g.*, `ParkAcqModes: Temperature every 1 hour`). A comparison between the two acquisition modes is given in Figure 3.2. Choosing an acquisition mode mainly depends on what is monitored and has implications for the resources used by the applications. Details about how to choose an acquisition mode are given in section 3.

The algorithmic part inside a continuous acquisition mode is executed periodically, each time the sensor sends a packet of data. During the acquisition, it is necessary to ensure that all packets of data are processed, to guarantee the integrity of acoustic signals. However, it may be necessary to suspend the acquisition to temporary execute an algorithm that has a long execution time. In order to help the developer to think about these important aspects, the MeLa language separates the content of acquisition modes in two kinds of sequences of instructions :

1. The `RealTimeSequence` (lines 27-34) is associated with real time constraints to guarantee that all the data from the sensor are processed. The execution time of the sequence must be shorter than the time between each packet of data, and other applications cannot delay the processing of a block of data until the point of missing a packet of data. The method used to verify these constraints is a scheduling analysis and is described in section 2.5.1.
2. The `ProcessingSequence` (lines 36-42) does not have real time constraints. Using this sequence means that developer accepts to miss some data from the sensor. However this sequence allows to call instructions with a long execution time that would raise an error inside a `RealTimeSequence`.

A `RealTimeSequence` can be called only from a `ContinuousAcqMode`. Indeed, the acquisition is stopped between each execution of a `ShortAcqMode`, thus it does not require a `RealTimeSequence`. It is also verified that the execution time of a `ShortAcqMode` is less than its execution time interval (the time between two subsequent calls). However a delayed acquisition in this case is less problematic since it would not affect the integrity of data (the acquisition is suspended anyway). Furthermore, it is very unlikely because

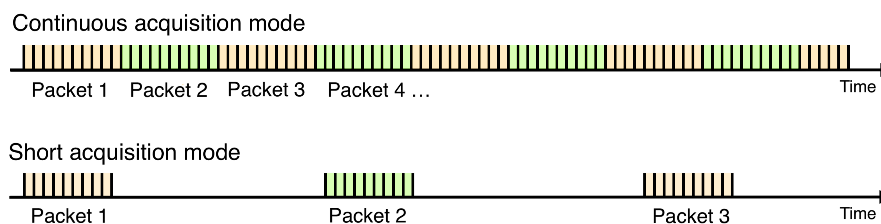


Figure 3.2 – Data packets for ContinuousAcqMode and ShortAcqMode.

the execution time interval of this mode is set in the coordinator and is intended to be of several minutes or hours (not less than one minute).

Inside the sequences of instructions, the developer writes the algorithm with variables and functions. The variables must be first declared inside each acquisition mode in a section called `Variables` (lines 20-24). The data types currently available to declare a variable are given in table A.1. The functions are called inside the sequences of instruction (lines 28, 29, 37 and 40 of table 3.1). A list of functions currently available is given in table A.2. Many of them have been selected from the CMSIS DSP library [2].

The functions can be organized using statements commonly found in imperative programming languages, such as `if` conditions and `for` loops. A particular aspect of the `if` statements in MeLa is that each branch requires a probability to be given for its occurrence (*e.g.*, the probability that an earthquake signal is strong enough to trigger a detection). That probability is used to compute properties of the application, especially the energy consumption and the volume of data transmitted through satellite communication (see section 3.2.4.2).

3.2.3 Mermaid float architecture

The instrument (figure 3.3) is made of a glass sphere that resists until a depth of 5000 meters. A hydraulic circuit transfers oil between a tank inside the sphere and an outside bladder. When the bladder deflates, the instrument volume decreases and its density increases, allowing it to dive. Up to red eight sensors can be installed on the instrument, although currently we have only experimented with a hydrophone and a *Conductivity Temperature Depth* (CTD) sensor to monitor water temperature, salinity and density. The hydrophone has two outputs to monitor sounds at low and high frequencies, between 0.1 Hz to 100 Hz and between 10 Hz to 10 kHz. The sampling frequency of each output can be chosen by the developer. A satellite antenna at the top of the instrument is used for positioning with the GPS and data transmission with the Iridium *Router-Based Unrestricted Digital Internetworking Connectivity Solutions* (RUDICS) protocol. Batteries have a total capacity of 4 kW.h, equivalent to four hundred times those of a smartphone (∼10 W.h), that can power the float for years or months depending on the power consumption of the applications. For the Mermaid floats currently monitoring the seismic activity, the expected lifetime is 5 years.

The float contains two electronic boards. The pilot board manages the hydraulics (*i.e.*, depth regulation) and communications (*i.e.*, GPS and Iridium). The acquisition board manages the sensors and data processing. It has 512 kB of programmable memory, 8 MB of *Static Random Access Memory* (SRAM) and 128 GB of flash (SD card). The microcontroller is based on a Cortex-M4 core which integrates a *Digital Signal Processor* (DSP) and works at a frequency of 32 MHz. This is a very limited configuration compared to a smart phone, but it has a low power consumption that is adapted for long-term operation.

2. CMSIS DSP library manual page : `https://www.keil.com/pack/doc/CMSIS/DSP/html/index.html`

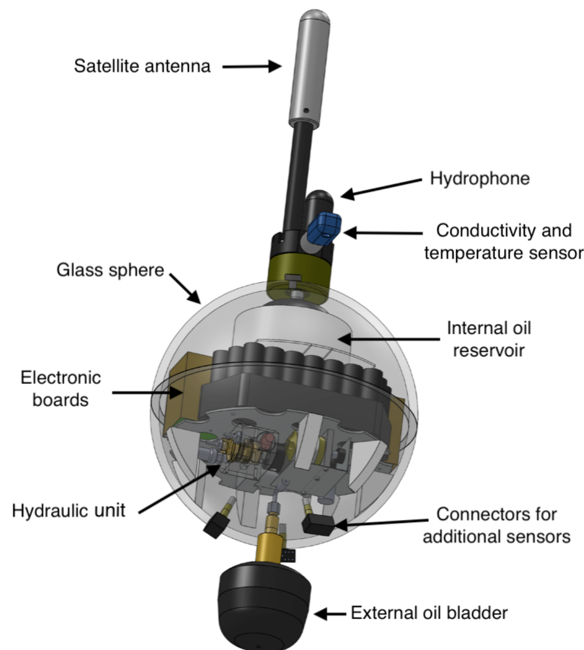


FIGURE 3.3 – Mermaid float.

Both electronic boards are programmed with C language which allows writing software with efficient execution time and low memory usage. Both have a *Real Time Operating System* (RTOS) allowing them to execute several tasks concurrently. The pilot board software can be configured with several parameters such as the dive duration, depth and other technical parameters such as the time interval between each depth correction. The acquisition board has access to the sensors and can be programmed by scientists with MeLa, that generates C code. Both boards can communicate with each other, for example the acquisition board can ask to the pilot board for the ascent of the float.

3.2.4 Code generation

3.2.4.1 Overview

The applications written in MeLa are transformed in C code suitable to program the Mermaid floats. This process is called code generation and is equivalent to a compiler but it generates C code instead of the binary file used to program the microcontroller. The mapping between the MeLa code and the generated code is illustrated in the figure 3.4.

A file to configure the pilot board is generated from the mission configuration part of the MeLa code. The rest of the MeLa code is used to generate the C code to program the acquisition board. The coordinator is mapped to a task containing a state machine

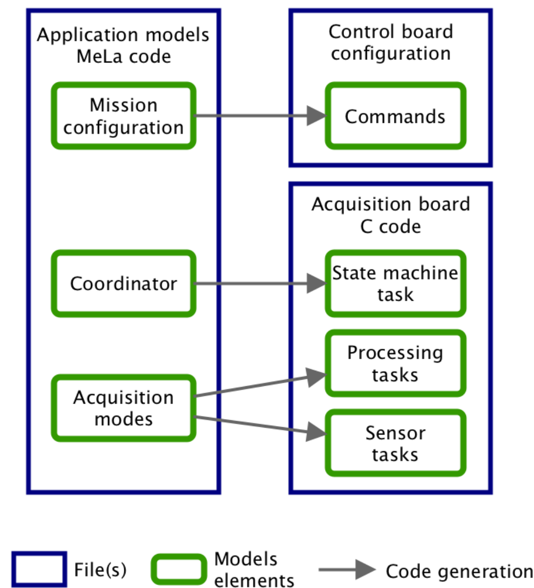


Figure 3.4 – Code generation mapping.

(i.e., a model of computation) [Miles and Hamilton, 2006] that manages the execution of acquisition modes and the messages exchanged with the pilot board. The acquisition modes are converted to processing tasks containing the sequences of instructions and sensor tasks handling the data from sensors (one sensor task can feed several processing tasks).

3.2.4.2 Priority rules

The real time operating system requires a priority of execution to be defined for each task. The highest priority is assigned to tasks with the shortest time interval between each execution (*i.e.*, the shortest period of execution), that is a rate-monotonic priority assignment [Sha et al., 1991]. An exception is during the execution of a processing sequence of a continuous acquisition mode ; the lowest priority is assigned to the processing sequence that can have a long execution time, so that the other tasks cannot be blocked.

3.2.4.3 Benefits of MeLa for embedded software programming

In MeLa, the functions accept different data types. For example, the `max()` function that searches the maximum value and index in an array accept integer and floating point arrays. On the contrary the C language requires specific functions for each data type, for example `maxArrayFloat()` and `maxArrayInt()`. In the MeLa library, the functions are defined by their MeLa name, their parameters types and their C name. Several functions can have the same MeLa name but different C names with different parameters types. When a function is called from the MeLa language, the parameters types are used to

choose the function with the appropriate C name. For example, if the MeLa code is `max(a)` with `a` an array of integers, the corresponding function is the one that accepts arrays of integers and have the C name `maxArrayInt()`, if `a` is an array of floats the corresponding function is the one with the C name `maxArrayFloat()`. This mechanism is also called type inference, it makes programs easier to read and write by reducing the language verbosity [Pierce and Turner, 2000].

The verbosity is also reduced for variable declarations. A variable declared in MeLa can have a much more verbose C equivalent. For example, declaring a StaLta variable in MeLa requires only one line, `StaLtaFloat stalta(5, 15, 10)`, but declaring it in C requires 3 lines because it corresponds to an array, a *circular buffer* structure that contains the array, and a *stalta* structure that contains the circular buffer :

```
extram float32_t stalta_cbuff_data[25];
circular_buffer_f32_t stalta_cbuff = {stalta_cbuff_data, 25, 0, 0, true};
stalta_f32_t stalta = {&stalta_cbuff, 5, 15, 10, 0, 0};
```

The *circular buffer* structure contains the pointer to the array of data, the length of the array, the index of the begin and end of the buffer, and a boolean indicating if the buffer is empty or not. An improper initialization or usage of these variables could lead to undefined behavior. In MeLa these are automatically initialized and they are hidden to the developer such that it prevents an improper usage (encapsulation principle). Also, the notion of pointer does not exist in MeLa, avoiding errors of writing to unknown memory addresses. For example, forgetting the & operator when defining the *stalta* structure would break the software.

MeLa also hides the mapping of variables that can be stored into the microcontroller SRAM memory (128 kB) or an external SRAM memory that is in a separate chip (8 MB). In the generated C code, each array is mapped to the external memory that has more storage capacity but small variables are mapped to the internal memory that is faster.

Thus writing an application with MeLa allows programmers that are not expert in embedded software programming to write reliable and efficient applications (challenge 1 and 3). Even for an embedded software expert, writing an application with MeLa instead of C reduces the possibilities of making errors.

3.2.5 Application verification

3.2.5.1 Static analysis of applications

The static analysis consists of computing properties of the applications from models without executing them. We use it to verify that the applications do not exceed the instrument capacities during their development. In its current version, MeLa is able to compute the processor usage, the battery life time and the amount of data to be transmitted by satellite, but other properties such as memory usage can be added to the analysis. Each function of the MeLa library is associated with information about execution time used to compute the processor usage. The execution time can be a constant value or an equation with parameters such as the size of arrays used during the call of

the function. Indeed the execution time can change by several orders of magnitude for different sizes of arrays. The functions can also have a specific meaning, such as "this function requests the ascent of the float" or "this function records data to transmit by satellite", that are interpreted by the analysis tools.

Processor usage The processor usage of one task U is defined by $U = C/T$ where C is the worst-case execution time of the task and T is the period of execution of the task. For n tasks, the processor usage is $U = \sum_{i=1}^{n} C_i/T_i$. The worst-case execution time C is the longest execution time among the possible execution path of a task (the processing sequence in a continuous acquisition mode is not taken in account since it does not have real time constraints). The execution time of paths is computed with information recorded in the library and size of arrays passed as parameters of functions. The period of execution T of a continuous acquisition mode is computed from the sampling frequency of the sensor and the size of the input array, while for a short acquisition mode it corresponds to the period defined in the *Coordinator*.

To determine if the processor is able to execute the tasks in time (*i.e.*, if tasks are schedulables), we use the Liu and Layland utilization bound [Liu and Layland, 1973]. This bound defines the maximum processor usage for a set of tasks. A set of n tasks is schedulable only if $U \leq n \cdot (2^{1/n} - 1)$. It means that a processor can be used at 100% of its capacity for one task ($n = 1$), but only 83% for two tasks ($n = 2$), 76% for four tasks ($n = 4$), and it tends to 69% for an infinite number of tasks. This bound is only valid if 1) tasks have a rate-monotonic priority assignment (as described in section 3.2.4.2), 2) tasks have a deadline equal to their period of execution, 3) tasks are independent from each other. These constraints are respected since 1) the code generation process assigns rate-monotonic priorities to tasks, 2) the deadline corresponds to the arrival period of samples (*i.e.*, time interval between each packet of data) that is the period of execution of tasks, 3) the functions of the library are written so that the execution of a task cannot be delayed by another one (*i.e.*, they cannot interfere), or at least the delay must be negligible (*e.g.*, the time to write on the SD card is negligible compared to the time to switch it on, thus if two tasks write at the same time, the writing time is negligible, only the switch on time is taken into account).

Duration of the dive The duration of a dive is needed to estimate the lifetime of the float and the amount of data transmitted each month. It depends on the duration of each stage of the dive. For the surface stage, we consider a constant duration of one hour, even if it can be shorter or longer for a real float since it depends on the amount of data to be transmitted by satellite. The descent and ascent stage duration are computed from the depth defined in the mission configuration and an estimated speed of the float. The parking stage has a maximum duration if no ascent request occurs for some time. This maximum duration has to be defined by the developer in the mission configuration. It may be shortened if an application requests the ascent of the float.

To estimate the mean duration of the parking stage, we use a Poisson law that gives the probability to have k ascent requests during the default parking stage duration (consi-

dering a fixed probability for ascent requests) :

$$p(k) = \frac{\lambda^k}{k!} e^{-\lambda} \tag{3.1}$$

The λ parameter represents the mean number of ascent requests during the default parking stage duration. It is computed from the mission configuration and the probabilities defined in an application (only probabilities leading to a function that trigger the ascent of the float are used). For example, if the maximum duration of the parking stage is 10 days, and the probability of the ascent is 2 per week, the λ parameter is equal to $10 * 2/7 = 2.86$.

The probability to have zero ascent request is :

$$p(0) = e^{-\lambda} \tag{3.2}$$

And the probability to have at least one ascent request is :

$$p(k > 0) = 1 - e^{-\lambda} \tag{3.3}$$

The mean parking duration, that is also the mean duration before the first ascent request, corresponds to the mean interval of time between each ascent request (i.e., the invert of the probability defined in the MeLa language) multiplied by the probability to have at least one ascent request during the default parking duration.

For example, the probability to have at least one ascent request is $1 - e^{-2.86} = 0.94$, and the mean parking duration is $7/2 * 0.94 = 3.3$ days. A curve of the mean parking duration as a function of the mean number of ascent requests for a default parking duration of 10 days is given figure 3.5.

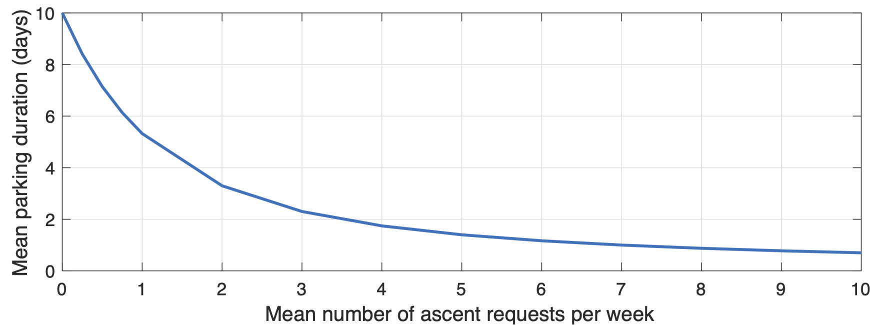


Figure 3.5 – Mean parking duration in function of the mean number of ascent requests for a default parking duration of 10 days.

Satellite transmission The amount of data transmitted by the float is computed from the recording functions called in the application. In the MeLa library, the recording functions are annotated with the amount of data they record for satellite transmission. For functions recording arrays, the amount of data is dependent of the array size passed as a parameter of the function. The probability annotations in the MeLa code are also used for the computation.

Battery life time The lifetime of the instrument LT is related to the energy contained in the battery E_{bat}, the mean energy consumption of a dive E_{dive} and the mean duration of a dive T_{dive} :

$$LT = E_{bat}/E_{dive} * T_{dive} \tag{3.4}$$

The energy contained in the battery (*i.e.*, the battery capacity) is known and the computation of the mean duration of a dive has been introduced previously. The mean energy consumption of a dive is equal to the sum of the energy consumed during each of the stages (i.e., descent, parking, ascent and surface).

For each stage, the energy consumed E_{stage} is the sum of the energy consumed by the actuators, the sensors, the acquisition board and the satellite communication devices. Naturally, when the float is underwater the satellite communication devices are switched off and does not consume any energy.

$$E_{stage} = E_{act} + E_{sens} + E_{board} + E_{com} \tag{3.5}$$

The energy consumed by the actuators E_{act} depends on the mission step. Most of the energy of the descent is consumed by operating a valve at the surface. Once the float is deep enough, the pressure of the water is high enough to transfer oil from the outer bladder to the inner reservoir with almost no energy required from the pump. Thus we assume that the energy consumption of the descent is a small constant value. For the parking we consider that the energy consumption is null even if there is occasionally some small depth correction. Most of the energy is consumed during the ascent because the pump has to push oil in the outer bladder where the pressure can reach several hundred of bars. We use a quadratic relation between the power consumption and the depth of the float, and add a constant value for the power needed to fill the bladder at the surface. The linear relation is a simplified model, we plan to improve it by taking account of the motor efficiency and behavior of the float during the ascent, and then validate it with experimental data.

The energy consumed by the sensors E_{sens} is the product of their activation time with their energy consumption. The activation of a sensor can be intermittent if it is used by a short acquisition mode. Moreover the same sensor can be used by several applications. Thus an algorithm computes the activation time of the sensors. For example, if two applications, A and B, use the same sensor, one for 2 minutes every 5 minutes and the other for 1 minute every 3 minutes, the algorithms compute the pattern of table 3.2 which repeats itself every 15 minutes. Here, the sensor is used 9 minutes over a period of 15 minutes, that is 60 % of the time. If the stage during which this sensor is activated has a mean duration of 10 hours, the activation time of the sensor is 6 hours.

The energy consumed by the acquisition board E_{board} is the power consumption of the board multiplied by the mission step duration. This simplified model assumes that the acquisition board power consumption is constant. It is a conservative model because the sleeping modes, activated when the board does not have any data to process, are not taken into account.

The energy consumed by satellite communication is the number of bytes to transmit Tx_{bytes} divided by the average speed of the data transmission Tx_{speed}, that gives the du-

Table 3.2 – Activation pattern of a sensor used by two applications allowing to compute the energy consumption of the sensor. The first row is the time, the three other rows are the activation state of the sensor, with 1 for the activated state and 0 for the disabled state.

Time in minutes	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
Sensor used by app A	1	1	0	0	0	1	1	0	0	0	1	1	0	0	0	...
Sensor used by app B	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	...
Total sensor usage	1	1	0	1	0	1	1	0	0	1	1	1	1	0	0	...

ration of the transmission, and multiplied by the power consumption of the transmission device P_{sat} :

$$E_{sat} = Tx_{bytes}/Tx_{speed} * P_{sat} \tag{3.6}$$

3.3 Developing with MeLa

A workflow to develop applications with the MeLa language is illustrated in figure 3.6. The presented workflow allows to verify that the applications do not exceed the limits of the instrument during the development process, before to have a functional application. The programming of the instrument is only done once, at the very end.

The first step is to define the duration and depth of a dive, the acquisition mode and the coordinator. For each application, the developer has to choose between a continuous or a short acquisition mode. Continuous acquisition modes are more adapted to detect sporadic signals, like those from earthquakes, because they process data without stopping. Short acquisition modes are more suitable for monitoring events that evolve slowly, like temperature or wind, because they have a reduced impact on the battery lifetime of the floats since the sensor is switched off between each data packet. A good practice for continuous acquisition modes is to have a detection part, with a short processing time, in the real-time sequence and a discrimination (classification) part, which often have a long processing time, in a processing sequence. Short acquisition modes only have a processing part, they are not intended for real time detection.

Once the acquisition mode is chosen, the developer defines the sampling frequency and writes a first version of the application, with its main functions. The application does not need to be fully functional in a first step ; for example, filter parameters do not yet need to be chosen. Only the information used by models to verify that the limits of the instrument are not exceeded is necessary : the length of the arrays, of the Fourier transforms, the functions used to process the data and also the probabilities of the conditional branches. The probabilities do not have to be exact, but should be conservative to obtain a safe estimation of the battery life time and the cost of satellite transmission.

If the limits of the instrument are not exceeded, the application can be composed with another application to verify that they will be both able to execute on the same instrument. The dive depth and maximum duration must be the same, and if the same sensor is used at the same time by two applications, the configuration of the sensor must

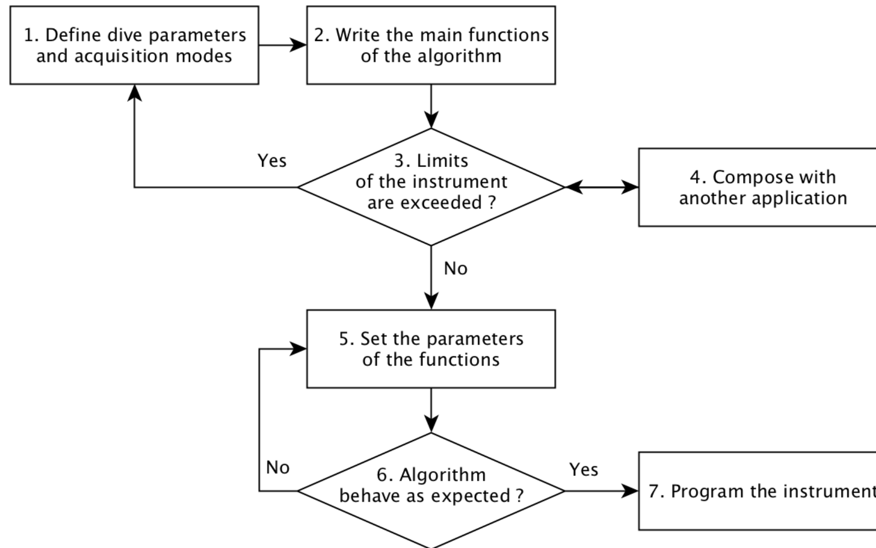


Figure 3.6 – Development workflow with MeLa.

be the same (*i.e.*, sampling frequency). If they are different an error is raised to force the developers to find a compromise. Once they are composed, it is necessary to verify again that the limits of the instrument are not exceeded. Most of the mechanisms used to execute the acquisition modes concurrently are managed by the embedded software code, this include the scheduling of tasks and the exchange of data from sensors to processing tasks.

The applications can be executed on a laptop (*i.e.*, simulation). Simulation is complementary to static analysis, it focuses on the behavior of the applications instead of the instrument limitations. It allows a developer to correctly set the parameters of an application without having to program a real instrument, for example she can verify that an application records as many earthquakes as expected and adjust parameters to improve the performances. Currently, the simulation handles only the processing of data, it does not fully simulate the behavior of the instrument.

The simulation code is generated from the MeLa code with the same library of functions as for the instrument. Most of the function implementations are exactly the same for the simulation and the instrument. It ensures that simulation gives results close to those that will be obtained on a float. Some differences exist because the DSP of the Cortex M4 is not available in a personal laptop. Emulating the float processor would allow to be even closer to the results of the instrument. During the simulation process, probability values can also be refined.

The code for the instrument is generated from the application. It can be compiled without any modification. Since verification about the limits of the instrument and simulation have been done during the development process, thanks to the MeLa capabilities, the applications can be deployed without requiring additional tests.

3.4 Experiments

We describe two detailed examples of real-life applications illustrating the capabilities of the MeLa language. The first example is the seismic detection algorithm implemented in the original Mermaid floats and the second one is an algorithm developed to detect D-calls of blue whales. We specify the algorithms and discuss the results of the analysis and simulation.

3.4.1 Detection of earthquakes

3.4.1.1 Scientific context

Seismic waves emitted by earthquakes are used by seismologists to map the interior of the earth. The speed of propagation of seismic waves, especially compressional (P) and shear (S) waves, is dependent on the temperature inside the Earth. When an earth-quake occurs, measurements of the travel time of the seismic waves allow to image cold subducting oceanic lithosphere and hot mantle plumes under volcanic islands such as Hawaii. In order to obtain a good and uniform image resolution, measurements all around the earth are needed, including in the oceanic regions that represent 70% of the surface of the globe. Because of the absence of seismographs in marine areas, scientists have developed the MERMAID floats. These are equipped with a hydrophone (i.e., an underwater microphone) that can record P and occasionally S waves as acoustic waves transmitted from the ocean floor into the water column. Recently, we have shown how even a small network of Mermaids was able to image a mantle plume beneath the Galapagos Islands [Nolet et al., 2019]. Another experiment is currently underway in the Pacific with 49 Mermaid floats deployed [3]. The Southern University of Science and Technology will also launch 10 Mermaids in the South China Sea in November 2020 and 5 in the Indian Ocean next year (Yongshun John Chen, personal communication 2020).

3.4.1.2 The seismic detection algorithm

The seismic wave detection algorithm is presented in detail in Sukhovich et al. [Sukhovich et al., 2015]. We give here a brief and simplified overview of this algorithm. The first step of the algorithm is the detection of an increase in the signal amplitude. The signal is filtered by a high pass filter to suppress the micro seismic noise at frequencies below 1 Hz. A *Short Term Average over Long Term Average* (STA/LTA) is used to detect an elevation of the absolute signal amplitude. In this example the STA/LTA ratio is the mean of the last 10 seconds of the signal over the mean of the last 100 seconds. When the result of the STA/LTA ratio exceeds a threshold, the discrimination part of the algorithm is triggered and decides if the signal is a seismic wave.

The discrimination algorithm computes the wavelet transform of the signal, equivalent to a bank of six bandpass filters. The six frequency bands are averaged over time and a

3. EarthScope Oceans website :
`http://geoweb.princeton.edu/people/simons/earthscopeoceans/`

normalization process is done between the noise part, before the trigger, and the signal part, after the trigger. This leads to a representation of the signal similar to a power spectrum, with six values for each frequency band. A criterion is computed from the distance between the measured powers and the center of six reference distributions. The *Signal over Noise Ratio* (SNR) is also computed. Both values are used in a decision to trigger the recording of the signal and eventually the ascent of the float to get a precise position with the GPS at surface.

3.4.1.3 Implementation with MeLa

The implementation with MeLa requires functions like STA/LTA, triggers, wavelets transforms and cumulative distributions. Implementing a specialized algorithm may require the involvement of an embedded software expert to write specific functions in C language in order to add them in the MeLa library. The MeLa language does not offer the full flexibility of a generic programming language in order to ensure the reliability and efficiency of the applications, and to permit the analysis of applications. But the current library of functions (table A.2) is already generic enough to be of use in many different applications. The MeLa code of the seismic application is accessible on Github, see Supplementary Materials section.

Once the seismic application has been implemented with the MeLa language, the analysis tool allows to verify that the limits of the instrument are not exceeded. Table 3.3 shows that the processor is used only 0.1% of the time. Indeed the time between each packet of data is long compared to the time required to process them. The autonomy of the float depends on the frequency of the ascent requests ; the estimated autonomy was found to be 5 years if the algorithm records 4 earthquakes and triggers one ascent per week (2.9 years if it records 10 earthquakes and triggers 10 ascents per week). The estimated amount of data transmitted per month was found to be respectively 708 kB and 915 kB. The 708 kB compares well with the Mermaids floats currently operating in the Pacific which transmit 400 kB per month with a compression algorithm that divides the size of data by 2. The processor usage of 0.1% is compared to the maximum allowed processor usage that is not necessarily 100% if several applications must be executed at the same time (as defined by the Liu and Layland theorem).

We have tested a version of the algorithm based on floating point numbers instead of integers to filter the micro seismic noise, process the STA/LTA and compute the wavelet transform. With floating point numbers the processor is used 0.17% of the time, higher than the integer implementation but still very low. It shows that choosing a floating point implementation is possible and may be preferable because it gives flexibility to design the high pass filter that removes micro seismic noise. We have also tried another version of the algorithm based on a Fourier transform instead of the numeric filter. It was found early that using a Fourier transform (1024 samples processed every 512 samples) increases the processor usage to 0.7%. This demonstrates that using Fourier transforms in real time is possible but uses more processor time than the original algorithm.

TABLE 3.3 – Analysis results given to the developers by MeLa.

Maximum processor usage during PARK: Valid: 0,1 % < 100 % Autonomy: 5 years Transmission per cycle: 142,6 kB Transmission per month: 694,39 kB	Energy consumption during DESCENT: 343mWh Processor consumption: 143mWh Actuator consumption: 200mWh Energy consumption during PARK: 8979,7mWh Processor consumption: 1315,7mWh HydrophoneBF: 7664mWh Actuator consumption: 0mWh Energy consumption during ASCENT: 4303,6mWh Processor consumption: 53,6mWh Actuator consumption: 4250mWh Energy consumption during SURFACE: 1032mWh Processor consumption: 10,3mWh Actuator consumption: 0mWh Transmission consumption: 1021,7mWh

3.4.1.4 Evaluation of the algorithm

The algorithm has been tested on a laptop with the simulation code generated from the MeLa language. To feed the algorithm we have used 10 months of continuous recording from a Mermaid recovered in August 2019. We compared the results of the simulation with the events sent through satellite by the Mermaid before its recovery. All 12 earthquakes detected by the Mermaid are also detected by the algorithm implemented in MeLa. Three additional non seismic events have also been detected, indicating slight differences in the implementations. Nevertheless, we conclude that the MeLa language can be used to implement advanced algorithms.

3.4.2 Detection of blue whales

3.4.2.1 Scientific context

Whales have become an important topic of study among marine biologists and scientists as they play a very crucial role in the health of the ocean ecosystem. They participate in the food chain by absorbing krill [Hildyard., Editor] and help to capture carbon from the atmosphere by rejecting nutrients that stimulate the growth of phytoplankton [Roman J, 2010]. Despite these important roles, whales are endangered worldwide. In particular, during the 20th century, the blue whale was an important whaling target. Nowadays, like other large whales, blue whales are threatened by other human activities (e.g. climate change impact on krill, ship strikes, fishing gears, toxic substances). Their protection and conservation requires a better understanding of their spatial distribution, migration, of their social structure, and how they communicate with one another. Long term acoustic monitoring at a global scale would help such studies. As all data cannot be sent by satellite, the processing such as counting the whale calls [Marques et al., 2009], must be done on the instrument.

3.4.2.2 The blue whale detection algorithm

Blue whales emit different sounds called A, B, C and D calls related to their social behaviors [Mcdonald et al., 2006]. We have developed an algorithm that detects the occurrence of D-calls and records their occurrence dates. The spectrograms of D-calls have a very specific shape. This is a narrow band signal that sweeps typically from 80 to 20 Hz as shown in figure 3.7. The algorithm detects this shape in real time. First, the algorithm computes the spectrum S of the signal with a Fourier transform of 64 samples and a window overlap of 50% (32 samples). After removing noise at low frequencies, below 20 Hz, the algorithm searches for the six highest values of the spectrum and computes a ratio between the two highest over the fifth and sixth highest $(max1(S) + max2(S))/(max5(S) + max6(S))$. This is done for successive time windows resulting in the curve shown in figure 3.8. The computed ratio is high only when a signal with a narrow frequency band exists. However, the ratio is not always very stable, especially if the signal is weak. We therefore compute the STA/LTA average that smoothes the curve as shown in figure 3.9. If the value of the STA/LTA exceeds a trigger value, the ratio is put inside a buffer to be used in the next step of the algorithm. The frequency at which is found the maximum amplitude of the spectra (figure 3.10) is also kept in memory.

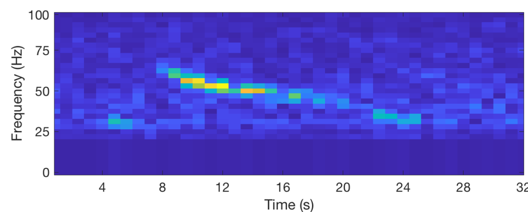


Figure 3.7 – Spectrogram of a blue whale D-call with low frequencies removed.

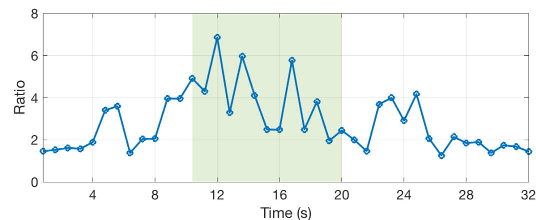


Figure 3.8 – Ratio of spectrum amplitudes for each spectrogram window.

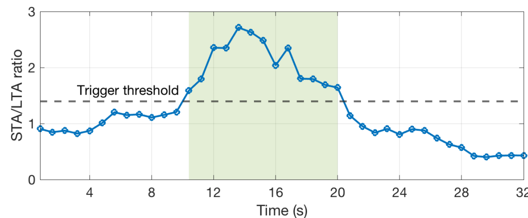


Figure 3.9 – STA/LTA computed from the ratio figure 3.8.

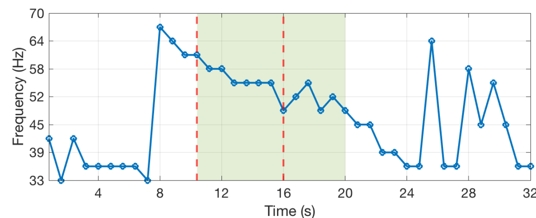


Figure 3.10 – Frequencies corresponding to the maximum amplitudes of the spectrogram.

After the value of the STA/LTA drops under the trigger threshold, the two curves are used to discriminate D-calls from other noises. Only the part of the curves between the trigger and the detrigger (green highlight) are in memory at this time. In order to remove potentially wrong values at the end or at the beginning of the curve, the time window is truncated. We use the frequency curve on figure 3.10 and select only the part between the maximum and minimum value (first minimum) as shown with the two dashed red

lines. Then, we count the number of times the frequency changes downward, upward, or keeps the same value (the frequency can only take 32 values corresponding to the frequency bins of the Fourier transform). For the highlighted part of the figure 3.10 the frequency goes downward 3 times and keep the same values 5 times. Finally several tests are done to check that the signal corresponds to a blue whale D-call and if they are all valid, the date of the detection is recorded. The tests have been defined empirically and are as follow :

- The length of the detection must be above 4 successive windows (green part of figure 3.9).
- The mean value of the ratio must be above 2.5
- The number of times the frequency goes downward between two successive windows must be more than 3 times the number of times the frequency goes upward.
- The number of times the frequency goes downward must be more than 2.
- The number of times the frequency goes downward must be more than 0.25 times the number of times the frequency stays stable.
- The maximum frequency must be above 40 Hz.
- The maximum frequency must not change by more than 20 Hz between within 2 points (0.8 seconds).

The algorithm could be improved by keeping values before the trigger that is a little late compared to the D-call arrival. Moreover, the discrimination process could be optimized with machine learning and input such as the features listed above or the ratio (figure 3.8) and frequency curves (figure 3.10).

3.4.2.3 Implementation with MeLa

The algorithm has been implemented with the MeLa language. The code is accessible on Github, see Supplementary Materials section. For this application the analysis estimates a processor usage of 2%. The autonomy of the float is found to be 4.6 years, this is less than the seismic detection application because the power consumption of the sensor is higher, due to a higher sampling frequency. The estimated amount of data transmitted each month is only 14 kB. This is much less than the seismic application because only timestamps are sent through satellite communication. However the probability of recording a blue whale D-call is estimated to be much higher with 5 records per hour. If 20 seconds of sounds were recorded for each detection, the amount of data to transmit every month would be 56047 kB. Transmitting such amount of data increases the costs of satellite transmission and reduces the life time of the float to 0.7 years.

Once a first version of the algorithm is ready we can compose it with the seismic application. A first error appears because the maximum duration of the dive and depth are not equal for the two applications, thus we defined both to 10 days and 1500 meters depth. A second error occurs because the two applications share the same sensor but at different sampling frequency. A solution could be decimate the signal but this is not yet implemented in the language. Or we could adapt the seismic algorithm to support higher sampling frequency. Instead we decided to use the two outputs of the hydrophone, one for low frequencies and one for high frequencies. However using the two outputs of the

hydrophone has a noticeable effect on the power consumption. Instead of an autonomy of 5 and 4.6 years for the two separated applications, MeLa computes an autonomy of 2.8 years when the two applications are composed, to be installed on the same instrument. The processor usage is still very low (*i.e.*, 2%) because neither applications require a lot of processing time.

At this point the algorithm has not been finalized, since several coefficients still have to be defined or refined, for example, the STA/LTA length or the tests done to validate that the signal is a blue whale D-call. Simulating the algorithm on a personal laptop with experimental data allowed to finalize the algorithm without programming a real instrument. The finalized algorithm is the one described in the previous section.

3.4.2.4 Evaluation of the algorithm

Contrary to the seismic detection algorithm, the blue whales detection algorithm has never been evaluated before. Thus, we compare its performances with state-of-the-art algorithms and datasets from the *Detection, Classification, Localization, Density Estimation* (DCLDE) community.

Evaluation protocol We used the data from the DCLDE 2015 challenge that has been recorded with High-frequency Acoustic Recordings Packages deployed off the southern and central coast of California. The data spans all four seasons over the 2009-2013 period [4] but we used a 50h-long subset that have been annotated during a recent collaborative campaign [Nguyen Hong Duc et al., 2021]. The annotators have identified a total of 916 D-calls, plus 101 40 Hz annotated sound events. The high-frequency data have been decimated to 200 Hz bandwidth to feed the MeLa algorithm.

Furthermore, we used as performance metrics the *Precision* (*i.e.*, total number of detected calls) and *Recall* (*i.e.*, total number of annotated calls), using the python library *sed_eval*[5] [Mesaros et al., 2016] for their implementation. As we are interested in soft detection of sound events, i.e. without the estimation of D-calls duration, we only used the onset time in our evaluation metrics with a large time margin of 6s from the reference time onset, within which the estimated onset needs to fit in so that a detection is counted as being correct.

The detection performance of the MeLa algorithm was compared to a custom *Convolutional Neural Network* (CNN) - based model. CNNs are increasingly used in classification applications involving acoustics [Cakir et al., 2017, Mac Aodha et al., 2018, Sainath and Parada, 2015] and have recently revealed promising performance for marine mammal detection [Liu et al., 2018, Luo et al., 2019, Shiu et al., 2020, Wang et al., 2018]. We used the ResNet architecture, which is a deep neural network using skip connections or short-cuts to jump over some layers [Schaetti, 2018]. Only 18 layers were stacked to avoid overfitting as the training set is not very large. It was trained from scratch to handle the size of the mel-spectrogram [Volkmann et al., 1937] images (110 × 90 instead of the initial

4. See the dataset documentation at `http://cetus.ucsd.edu/dclde/datasetDocumentation.html`
5. `https://tut-arg.github.io/sed_eval/sound_event.html`

shape of 224 × 224). Each image is generated from 5s audio excerpts taken sequentially from the recordings. Our implemented version is based on existing open source codes [6].

Results The Resnet model has a precision of 82 % and a recall of 69 % for 765 overall detected events. The MeLa algorithm has a precision of 99 % and a recall of 55 % for 513 overall detected events. Even if the recall is lower for the MeLa algorithm, its precision is better which is an advantage if the algorithm is used in an alarm system to prevent ship collision. The main differences between the algorithms reside in the resources they use :

- For the execution time, it takes 12 seconds for Resnet to process a 5 minutes long recording whereas it takes only 15 milliseconds for the algorithm written in MeLa ; for one year of data, it is 14 days against 26 minutes. Moreover the CNN has been executed on GPU Geforce GTX1060 whereas the MeLa algorithm has been executed on a laptop.
- The network Resnet size is 134.4 MB whereas the programmable memory of the float has only 256 kB of space, the MeLa algorithm size is 410 kB if compiled for a laptop and 139 kB if compiled for the float (and 148 kB if compiled with the seismic detection algorithm).
- The GPU used in the evaluation has a power consumption of 116 Watts, which would consume the 4 kWatts.h of energy available on the MERMAID float just after a day and a half. If the MeLa algorithm is used on the GPU to process one year of data, it will consume 46 W.h instead of 39 kW.h for the CNN ; this is equivalent to the energy required by an electric car to travel respectively 300 meters and 260 kilometers (for a car consuming 15 kWh / 100 km).

The algorithms developed with MeLa are suited to program Mermaid floats but can also be used to process large amounts of data with low execution time and energy consumption, which also mean less environmental impact.

3.5 Discussion and conclusion

We have developed a programming language called MeLa dedicated to the Mermaid instrument, a multidisciplinary float that can monitor the oceanic environment with multiple sensors. The language is a DSL created using a *Model Driven Engineering* (MDE) approach [Kent, 2002, Mussbacher et al., 2014, Schmidt, 2006]. The language allows non-specialists of embedded systems to write reliable and efficient applications for the Mermaid instrument. It uses models to verify that the applications comply with the limited resources of the instrument and to compose (*i.e.*, combine) applications developed independently but to deploy in a same instrument. The code to execute the applications on a personal computer and the code for the instrument are generated from models using rules defined by embedded software experts.

Generic programming languages such as C, Java or Python do not have functionalities such as those offered by MeLa. Compared to MeLa, writing applications with those

6. `https://github.com/keras-team/keras-contrib/blob/master/keras_contrib/applications/resnet.py`

languages increases the risk of making errors that may compromise the integrity of the instrument, even for an embedded software expert. A software library or a framework such as Arduino [7] can reduce the risks with specialized functions that encapsulate low-level concerns (*e.g.*, a function that reads a sensor) and by defining a default architecture for the code (*e.g.*, with a setup and a loop function for Arduino). It helps the developers but the code is still written in a generic programming language for which the risk of errors is higher. Furthermore, those languages do not incorporate analysis capabilities included in MeLa because it requires platform-specific information (*e.g.*, energy consumption, execution time) that are not compatible with their generic aspect. Analysis tools exist but also need platform-specific information, for example the processor speed or the time required to read a sensor.

There exists a few DSLs comparable to MeLa such as CPAL [Navet and Fejoz, 2016], MAUVE [Gobillot et al., 2018] or Mbeddr [Voelter et al., 2012]. However they have mostly been designed for embedded software developers. They offer high-level programming abstractions but are still too close to embedded software concerns ; for example, in CPAL and MeLa the processor usage is computed from execution times that must be manually inserted into the code, whereas for MeLa this information is hidden in the library of functions. Moreover, they do not include a composition tool for combining several applications.

MeLa has been developed for the Mermaid float, but could also be used to program other instruments. One of those could be the AudioMoth [Hill et al., 2018, 2019, Prince et al., 2019], an autonomous acoustic monitoring device that can be programmed with applications, for example to detect cicadas or bat calls. More generally, the language could be used for most sensors used in the Internet of Things [Atzori et al., 2010, Kocakulak and Butun, 2017]. However, it is presently limited to sensors and cannot be used to program actuators or display devices.

Mela is also limited to program existing instruments, for which a model of operation can be created and measurements such as execution time or energy can be done. The development of embedded systems from scratch is not possible with MeLa, it would require many other features such as adding a large library of configurable software and hardware components to create a system, and also taking account of physical considerations (*e.g.*, pressure change and its effect on actuators power consumption for Mermaid).

Industry such as automotive or spatial industry have the budget to follow very strict development processes with international standards, quality insurance, static analysis and the like [Prause et al., 2016]. Such processes cannot be followed each time a scientist wants to add an application to the Mermaid, it would require budgets that many scientists do not have. Thus instead of following the classical development process that consists of developing a signal processing application in a language such as Matlab [8], refine it into embedded software code, integrate it in an embedded software architecture and test it on the embedded hardware, our approach allows developing applications in a single step, and by a non-specialist. Such an approach could also be used for other multidisciplinary

7. Arduino website : `https://www.arduino.cc/`
8. Matlab website : `https://fr.mathworks.com/`

instruments that require adaptability.

We plan to integrate several other sensors (*e.g.*, chemical, magnetic, optical) to the Mermaid instrument. Since the instrument can be deployed for several years (depending on the applications), we plan to add over the air programming capabilities in order to enable scientists to modify the software after deployment. The MeLa language itself can be improved with several features such as :

- Raising its level of abstraction, for example by allowing the developers to define an overlap between each packet of data which can be useful for the computation of spectrograms.
- Adding machine learning capabilities to automate, at least in part, the development of algorithms. Models, such as decision trees or neural networks, can be integrated and trained into an application. It is also possible to optimize the value of specific parameters with linear regression, for example a threshold in a condition.
- Computing RAM, flash and programmable memory usage that are very limited in embedded systems. It would help to prevent exceeding of their capacity.
- Applications sharing the same sensor with different sampling frequencies should be able to take advantage of a decimation filter. Such a filter must be steep enough to prevent aliasing and doing it automatically is challenging.
- Development tools to offer a better experience to developers and incite them to adopt the language. For example, a development environment with auto completion and highlighting of the code snippets that use the most of resources, or plotting functions for simulation.

We expect that MeLa, certainly after adding such features, will stimulate the creation of applications from multiple disciplines and will lead to significant cost savings for future programs to monitor the oceanic environment.

Références

Atzori, L., Iera, A., and Morabito, G. (2010). The internet of things : A survey. *Computer Networks*, 54(15) :2787 – 2805.

Bonnieux, S., Mosser, S., Blay-Fornarino, M., Hello, Y., and Nolet, G. (2019). Model driven programming of autonomous floats for multidisciplinary monitoring of the oceans. In *OCEANS 2019 - Marseille*, pages 1–10, Marseille, France. IEEE.

Cakir, E., Adavanne, S., Parascandolo, G., Drossos, K., and Virtanen, T. (2017). Convolutional recurrent neural networks for bird audio detection. *2017 25th European Signal Processing Conference (EUSIPCO)*, pages 1744–1748.

Davis, R. E., Regier, L. A., Dufour, J., and Webb, D. C. (1992). The Autonomous Lagrangian Circulation Explorer (ALACE). *Journal of Atmospheric and Oceanic Technology*, 9(3) :264–285.

Devi, G. K., Ganasri, B., and Dwarakish, G. (2015). Applications of remote sensing in satellite oceanography : A review. *Aquatic Procedia*, 4 :579 – 584. International Conference on Water Resources, Coastal and Ocean Engineering (ICWRCOE'15).

Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional, 1st edition.

Gobillot, N., Lesire, C., and Doose, D. (2018). A design and analysis methodology for component-based real-time architectures of autonomous systems. *Journal of Intelligent and Robotic Systems*, pages 1–16.

Gould, W. J. (2005). From swallow floats to argo—the development of neutrally buoyant floats. *Deep Sea Research Part II : Topical Studies in Oceanography*, 52(3) :529 – 543. Direct observations of oceanic flow : A tribute to Walter Zenk.

Hello, Y., Royer, J. Y., Rivet, D., Charvis, P., Yegikyan, M., and Philippe, O. (2019). New versatile autonomous platforms for long-term geophysical monitoring in the ocean. In *OCEANS 2019 - Marseille*, pages 1–8.

Hildyard.(Editor), A. (2001). *Endangered Wildlife and Plants of the World, Volume 12*. Marshall Cavendish Corp.

Hill, A. P., Prince, P., Piña Covarrubias, E., Doncaster, C. P., Snaddon, J. L., and Rogers, A. (2018). Audiomoth : Evaluation of a smart open acoustic device for monitoring biodiversity and the environment. *Methods in Ecology and Evolution*, 9(5) :1199–1211.

Hill, A. P., Prince, P., Snaddon, J. L., Doncaster, C. P., and Rogers, A. (2019). Audiomoth : A low-cost acoustic device for monitoring biodiversity and the environment. *HardwareX*, 6 :e00073.

Kent, S. (2002). Model driven engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 286–298, London, UK, UK. Springer-Verlag.

Kocakulak, M. and Butun, I. (2017). An overview of wireless sensor networks towards internet of things. In *2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 1–6.

Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20 :46–61.

Liu, S., Liu, M., Wang, M., Ma, T., and Qing, X. (2018). Classification of cetacean whistles based on convolutional neural network. In *2018 10th International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 1–5.

Luo, W., Yang, W., and Zhang, Y. (2019). Convolutional neural network for detecting odontocete echolocation clicks. *The Journal of the Acoustical Society of America*, 145 :EL7–EL12.

Mac Aodha, O., Gibb, R., Barlow, K. E., Browning, E., Firman, M., Freeman, R., Harder, B., Kinsey, L., Mead, G. R., Newson, S. E., Pandourski, I., Parsons, S., Russ, J., Szodoray-Paradi, A., Szodoray-Paradi, F., Tilova, E., Girolami, M., Brostow, G., and Jones, K. E. (2018). Bat detective—deep learning tools for bat acoustic signal detection. *PLOS Computational Biology*, 14(3) :1–19.

Manley, J. and Willcox, S. (2010). The wave glider : A new concept for deploying ocean instrumentation. *IEEE Instrumentation Measurement Magazine*, 13(6) :8–13.

Marcelli, M., Piermattei, V., and Zappalà, G. (2011). Advances in low cost marine technologies. *WIT Transactions on Modelling and Simulation*, 51 :497–507.

Marques, T. A., Thomas, L., Ward, J., DiMarzio, N., and Tyack, P. L. (2009). Estimating cetacean population density using fixed passive acoustic sensors : An example with blainville's beaked whales. *The Journal of the Acoustical Society of America*, 125(4) :1982–1994.

Matsumoto, H., Jones, C., Klinck, H., Mellinger, D. K., Dziak, R. P., and Meinig, C. a. (2013). Tracking beaked whales with a passive acoustic profiler float. *The Journal of the Acoustical Society of America*, 133(2) :731–740.

Mcdonald, M., Hildebrand, J., and Mesnick, S. (2006). Biogeographic characterisation of blue whale song worldwide : Using song to identify populations. *Journal of Cetacean Research and Management*, 8.

Meindl, A. (1996). Guide to moored buoys and other ocean data acquisition systems. *WMO & IOC Data Buoy Cooperation Panel*.

Mesaros, A., Heittola, T., and Virtanen, T. (2016). Metrics for polyphonic sound event detection. *Applied Sciences*, 6 :162.

Miles, R. and Hamilton, K. (2006). *Learning UML 2.0*. O'Reilly Media.

Mussbacher, G., Amyot, D., Breu, R., Bruel, J.-M., Cheng, B. H. C., Collet, P., Combemale, B., France, R. B., Heldal, R., Hill, J. H., Kienzle, J., Schöttle, M., Steimann, F., Stikkolorum, D. R., and Whittle, J. (2014). The relevance of model-driven engineering thirty years from now. In *MoDELS*.

Navet, N. and Fejoz, L. (2016). Cpal : high-level abstractions for safe embedded systems. In *DSM@SPLASH*.

Nguyen Hong Duc, P., Torterotot, M., Samaran, F., White, P. R., Gérard, O., Adam, O., and Cazau, D. (2021). Assessing inter-annotator agreement from collaborative annotation campaign in marine bioacoustics. *Ecological Informatics*, 61 :101185.

Nolet, G., Hello, Y., van der Lee, S., Bonnieux, S., Ruiz, M. J. C., Pazmino, N. A., Deschamps, A., Regnier, M. M., Font, Y., Chen, Y. J., and Simons, F. J. (2019). Imaging the Galápagos mantle plume with an unconventional application of floating seismometers. In *Scientific Reports*.

Parr, T. J. and Quong, R. W. (1995). Antlr : A predicated- ll(k) parser generator. *Softw., Pract. Exper.*, 25 :789–810.

Pierce, B. C. and Turner, D. N. (2000). Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1) :1–44.

Prause, C. R., Bibus, M., Dietrich, C., and Jobi, W. (2016). Software product assurance at the German space agency. *Journal of Software : Evolution and Process*, 28(9) :744–761.

Prince, P., Hill, A., Piña-Covarrubias, E., Doncaster, C., Snaddon, J., and Rogers, A. (2019). Deploying acoustic detection algorithms on low-cost, open-source acoustic sensors for environmental monitoring. *Sensors*, 19 :553.

Riser, S. C., Nystuen, J. A., and Rogers, A. (2008). Monsoon effects in the bay of bengal inferred from profiling float-based measurements of wind speed and rainfall. *Limnol. Oceanogr.*, 53 :2080–2093.

Roemmich, D., Johnson, G., Riser, S., Davis, R., Gilson, J., Owens, W., Garzoli, S., Schmid, C., and Ignaszewski, M. (2009). The Argo Program : Observing the Global Ocean with Profiling Floats. *Oceanography*, 22(2) :34–43.

Roman J, M. J. (2010). The whale pump : marine mammals enhance primary productivity in a coastal basin. *PLoS One*.

Sainath, T. N. and Parada, C. (2015). Convolutional neural networks for small-footprint keyword spotting. *INTERSPEECH*.

Schaetti, N. (2018). Character-based convolutional neural network andresnet18 for twitter author profiling. In *Notebook for PAN at CLEF 2018*.

Schmidt, D. C. (2006). Guest editor's introduction : Model-driven engineering. *Computer*, 39 :25–31.

Sha, L., Klein, M., and Goodenough, J. (1991). Rate monotonic analysis for real-time systems. Technical Report CMU/SEI-91-TR-006, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Shiu, Y., Palmer, K., Roch, M., Fleishman, E., Liu, X., Nosal, E.-M., Helble, T., Cholewiak, D., Gillespie, D., and Klinck, H. (2020). Deep neural networks for automated detection of marine mammal species. *Scientific Reports*, 10 :607.

Sukhovich, A., Bonnieux, S., Hello, Y., Irisson, J.-O., Simons, F. J., and Nolet, G. (2015). Seismic monitoring in the oceans by autonomous floats. *Nature Communications*, 6(8027) :1–6.

Venkatesan, R., Ramesh, K., Kishor, A., Vedachalam, N., and Atmanand, M. A. (2018). Best practices for the ocean moored observatories. *Frontiers in Marine Science*, 5 :469.

Voelter, M., Ratiu, D., Schaetz, B., and Kolb, B. (2012). Mbeddr : An Extensible C-Based Programming Language and IDE for Embedded Systems. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications : Software for Humanity*, SPLASH '12, page 121–140, New York, NY, USA. Association for Computing Machinery.

Volkmann, J., Stevens, S., and Newman, E. (1937). A scale for the measurement of the psychological magnitude pitch. *J. Acoust. Soc. Am.*, 8 :208.

Wang, D., Zhang, L., Lu, Z., and Xu, K. (2018). Large-scale whale call classification using deep convolutional neural network architectures. *2018 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC)*, pages 1–5.

Wüst, G. (1964). The major deep-sea expeditions and research vessels 1873–1960 : A contribution to the history of oceanography. *Progress in Oceanography*, 2 :1 – 52.

Yang, J., Riser, S., Nystuen, J., Asher, W., and Jessup, A. (2015). Regional rainfall measurements using the passive aquatic listener during the spurs field campaign. *Oceanography*, 28.

Chapitre 4

Perspectives et conclusion

Contents

4.1 Les perspectives pour MeLa 76
4.1.1 Augmenter le niveau d'abstraction 76
4.1.2 Automatiser le développement d'algorithmes 76
4.1.3 Améliorer le processus de développement 77
4.1.4 Améliorer les capacité d'analyse, de composition d'applications et de génération de code 77
4.1.5 Étendre l'utilisation de MeLa à la programmation d'autres instruments 78
4.2 Les perspectives pour l'instrument 78
4.2.1 Étendre les capacités du flotteur 78
4.2.2 Déploiement des applications 79
4.3 Conclusion 79
Références 81
. 82

4.1 Les perspectives pour MeLa

4.1.1 Augmenter le niveau d'abstraction

Le langage MeLa peut être amélioré en élevant le niveau d'abstraction du langage, c'est-à-dire en diminuant le niveau de détail avec lequel peuvent être écrites les applications. Un plus haut niveau d'abstraction permettrait aux développeurs d'écrire des applications plus rapidement en diminuant le niveau d'expertise requis pour la programmation, mais aussi pour la création d'algorithmes de traitement du signal.

Par exemple, le traitement acoustique des données nécessite souvent de calculer un spectrogramme et pour cela il est usuel que chaque fenêtre temporelle utilisée dans le calcul d'un spectre (une ligne verticale du spectrogramme) recouvre une partie de la fenêtre précédente et de la suivante d'un certain nombre d'échantillons. Le recouvrement des données peut être fait avec des buffers et des tableaux dans la version actuelle de MeLa. Cependant, il serait plus efficace de le définir en une seule ligne de code tel que `overlap: 10 samples`, ou avec un pourcentage tel que `overlap: 10 %`.

Finalement, on peut imaginer que le développeur n'ai qu'à donner des spécifications de haut niveau telles que `record 2 minutes of data when an earthquake is detected`. Le problème avec ce type de spécification est que pour générer du code il faut que l'algorithme pour détecter un tremblement de terre, ou une baleine, ou quoi que ce soit d'autre, soit prédéfini, ce qui n'est pas le cas. Des approches existent pour générer des algorithmes automatiquement, au moins en partie ; ces approches sont décrites dans la sous-section suivante.

Un autre exemple pour augmenter le niveau d'abstraction est d'intégrer des unités physiques, telles que l'amplitude des ondes acoustiques en Pascal, la fréquence en Hertz ou le temps en secondes au lieu d'un nombre d'échantillons. Pour les développeurs, cela permettrait de travailler avec des quantités plus parlantes que des nombres sans unité. La conversion des unités pourrait être gérée par le langage, par exemple en convertissant des pieds en mètres et inversement ; c'est ce type d'erreur qui a provoqué la perte de la sonde Mars Climate Orbiter [1]. Par ailleurs, l'ajout d'unités physiques dans le langage permettrait de faire d'autres vérifications, par exemple pour empêcher que deux nombres soient additionnés s'ils n'ont pas la même unité.

4.1.2 Automatiser le développement d'algorithmes

Le développement des algorithmes peut être automatisé, au moins en partie, avec des techniques d'apprentissage automatique. L'apprentissage automatique utilise des données annotées pour trouver les meilleurs paramètres d'un modèle qui peut être un réseau de neurones, un arbre de décision, etc. De tels modèles peuvent être intégrés dans les applications s'ils sont suffisamment petits pour la mémoire de la carte d'acquisition. Le modèle peut également être l'application elle-même pour laquelle des paramètres spécifiques désignés par le développeur doivent être optimisés, par exemple en cherchant la meilleure valeur possible pour un seuil de détection. L'application elle-même peut

1. NASA website : `https://solarsystem.nasa.gov/missions/mars-climate-orbiter/in-depth/`

également être générée automatiquement, par exemple la programmation génétique est une technique qui génère des programmes optimisés pour un problème spécifique en utilisant des mécanismes évolutifs [Zhang et al., 2005].

4.1.3 Améliorer le processus de développement

L'interface utilisateur est importante pour améliorer l'expérience des développeurs et leur productivité. Par exemple la mise en surbrillance des parties du code qui utilisent le plus de ressources permettrait d'identifier instantanément les problèmes d'utilisation excessive du processeur ou de consommation d'énergie.

La simulation est une étape très importante pour finaliser les algorithmes. La simulation des applications se limite aujourd'hui à reproduire le comportement des algorithmes du flotteur, mais des fonctionnalités spécifiques pourraient être ajoutées, en particulier des fonctions de tracé de figures (*i.e.*, plotting functions).

Par ailleurs la simulation des applications pourrait être utilisée pour mesurer les probabilités d'exécution des différentes branches de l'algorithme plutôt que de laisser le développeur les définir. Les données utilisées pour alimenter l'algorithme doivent en revanche correspondre à celles attendues par les flotteurs, mais le problème se pose aussi lorsque l'algorithme est développé. Idéalement, il faudrait pouvoir fournir avec le langage une bibliothèque de données enregistrées en continu par quelques flotteurs à différents endroits du globe.

4.1.4 Améliorer les capacité d'analyse, de composition d'applications et de génération de code

Différentes capacités d'analyse peuvent être intégrées au langage. Parmi ces capacités il sera important de prendre en compte la quantité de mémoire utilisée en fonction des différents types de mémoire ; la mémoire RAM contenant les variables des applications, la mémoire Flash (*i.e.*, carte SD) contenant les données enregistrées et la mémoire programmable contenant le programme exécutable.

Il est toujours possible d'améliorer la fidélité des modèles d'analyse vis-à-vis de la réalité. La création des modèles d'analyse demande cependant un investissement en temps. Il serait intéressant d'automatiser la création de ces modèles, par exemple pour trouver le modèle calculant le temps d'exécution d'une fonction qui dépend de la taille du tableau passé en paramètre. La génération de modèle à partir de code source a par exemple été étudiée avec le framework MoDisco [Béziers la Fosse et al., 2018] même si dans ce cas précis le modèle généré est statique, il n'intègre pas la notion de dépendance à des paramètres.

La composition des applications peut aussi être améliorée. Dans le chapitre précédent, nous avons présenté la composition de deux applications, pour la détection de séismes et de baleines, en supposant qu'elles utilisent deux capteurs différents. Les deux applications pourraient utiliser le même capteur, mais un filtre de décimation est nécessaire pour adapter la fréquence d'échantillonnage au besoin de chaque application. Cependant, la pente du filtre de décimation doit être suffisamment forte pour prévenir le phénomène

de repliement de spectre (*i.e.*, conversion de hautes fréquences en basse fréquence si le critère de Shanon n'est pas respecté). Pour le langage MeLa, il faudrait être capable de générer ce filtre automatiquement à partir de données acoustiques.

Le code généré pour l'instrument pourrait être optimisé en fonction des besoins des applications. Par exemple, la fréquence du processeur pourrait être ajustée pour optimiser la consommation d'énergie ou bien la méthode d'enregistrement des données sur la carte SD pourrait se faire avec une fonction non bloquante plutôt que bloquante. Cela requiert en revanche l'utilisation d'un modèle de plateforme pouvant être configuré de différentes façons, d'un modèle d'analyse de performance (*e.g.*, temps d'exécution) capable de prendre en compte ces différentes possibilités et d'un algorithme permettant de choisir une configuration optimale parmi un ensemble de configurations possibles [Lazreg et al., 2019].

4.1.5 Étendre l'utilisation de MeLa à la programmation d'autres instruments

MeLa a été créé pour la programmation des flotteurs Mermaid, mais il est envisageable de l'utiliser pour programmer d'autres instruments. Par exemple, la carte d'acquisition AudioMoth [Hill et al., 2018, 2019, Prince et al., 2019] utilisée, entre autres, pour détecter les sons émis des chauves-souris. En général, le langage peut être utilisé pour la plupart des capteurs employés dans le domaine de l'Internet des Objets [Atzori et al., 2010, Kocakulak and Butun, 2017]. Pour étendre l'utilisation du langage à d'autres instruments, ou plus généralement d'autres systèmes embarqués, il faudrait lui ajouter la capacité de gérer des dispositifs d'affichage et des actionneurs. Cependant, certaines fonctionnalités du langage de programmation dépendent du système à programmer. A titre d'exemple, la définition d'une profondeur et d'une durée de plongée est spécifique aux flotteurs Mermaid. Il faudrait que le langage puisse s'adapter en fonction du système à programmer en composant plusieurs langages, par exemple les modes d'acquisitions peuvent être communs à tous les systèmes de capteurs et la gestion des plongées (durée et profondeur), spécifique aux flotteurs. En plus de composer les langages, il faudrait être capable de composer les modèles d'analyse et de génération de code.

4.2 Les perspectives pour l'instrument

4.2.1 Étendre les capacités du flotteur

Le modèle de fonctionnement du flotteur est actuellement relativement simple avec une phase de descente et de parking et une de remontée, mais il est envisageable de réaliser des profils plus complexes avec plusieurs phases de remontées et de descentes, ou encore empêcher la remontée du flotteur en surface si de la glace est détectée. Avec une telle complexité dans le pilotage du flotteur, des conflits entre les différentes applications pourraient survenir. Un moyen simple de les résoudre serait de définir des priorités pour chaque application.

4.2.2 Déploiement des applications

Étant donné que les applications peuvent évoluer en fonction des premiers résultats obtenus après un déploiement, ou en fonction de l'emplacement du flotteur (*e.g.*, le paysage acoustique peut être différent suivant la position du flotteur), il est nécessaire de pouvoir reprogrammer le flotteur à distance. Reprogrammer les flotteurs via un système de communication très contraint comme Iridium peut avoir un impact sur les batteries de l'instrument si l'opération est effectuée trop souvent et que les programmes à installer ont une taille conséquente (vis-à-vis des capacités de transmission par satellite). Idéalement il faudrait pouvoir ne reprogrammer que partiellement l'instrument avec les parties modifiées des applications, comme avec un protocole de programmation adapté [Kim and Joo, 2009] ou bien en utilisant un langage interprété (*e.g.*, python), en supposant que l'interpréteur n'ait pas besoin d'être mis à jour.

4.3 Conclusion

Les océans sont d'une importance capitale que ce soit pour le climat, la biodiversité ou les activités humaines, en tant que source de nourriture ou d'inspiration pour la création de nouveaux médicaments et de nouvelles technologies. Les scientifiques étudient les océans afin de mieux les comprendre pour prévenir les risques associés, tels que les tsunamis, et aider à la mise en place de politiques de préservation du climat et de la biodiversité. Le flotteur Mermaid est un instrument qui a été créé pour la surveillance des océans et l'étude de l'intérieur de la Terre (par la tomographie sismique), il peut être équipé de différents capteurs. Les données de ces capteurs, et en particulier les données acoustiques, doivent être traitées sur l'instrument, car les capacités de transmissions de données de l'instrument sont limitées et il est parfois nécessaire que l'instrument réagisse en temps réel en fonction des données acquises.

Les algorithmes de traitements de données sont écrits par des scientifiques. Le processus de développement usuel est que ces applications soient ensuite transmises à une entreprise qui intégrera ces algorithmes dans l'instrument. L'instrument Mermaid étant pluridisciplinaire, plusieurs scientifiques de différentes spécialités (*e.g.*, météorologie, biologie, géosciences) doivent pouvoir intégrer différentes applications de traitement de données dans l'instrument. Il est difficile d'adopter une telle approche, car le moindre changement du logiciel requiert de faire appel à l'entreprise et de suivre un processus de développement complexe.

Ce travail de thèse nous a permis de développer un langage dédié à la programmation des instruments Mermaid. Ce langage a été conçu pour être utilisé par des scientifiques qui ne sont pas experts en systèmes embarqués. Il prend en compte les ressources limitées de l'instrument au cours du développement logiciel, ce qui permet de savoir très tôt dans le processus de développement si une application sera capable de fonctionner correctement sur l'instrument sans impacter significativement la durée de vie de ses batteries. Un outil de simulation des applications permet de vérifier leur bon fonctionnement sur un ordinateur personnel avant la programmation de l'instrument. Un outil de composition

permet d'incorporer plusieurs applications développées indépendamment par plusieurs scientifiques pour programmer un même instrument. Les règles définissant la transformation du code MeLa en code pour la programmation de l'instrument ont été définies par des spécialistes en systèmes embarqués. Ces règles, tout comme celles imposées par le langage MeLa (*e.g.*, programmation avec des modes d'acquisitions), permettent de garantir que le code est fiable (*e.g.*, sans bugs) et efficace (*e.g.*, sans utilisation excessive des ressources de l'instrument). Le langage MeLa a été testé dans un premier temps sur une carte Arduino, puis deux applications pour la détection de séismes et de baleines bleues ont été implémentées. L'approche à donc été validée avec des algorithmes répondant à des besoins scientifiques réels.

Nous espérons que ce langage permettra de faciliter les initiatives scientifiques de surveillance des océans pour les étudier et mieux les protéger. Plusieurs perspectives d'amélioration du langage ont été présentées dans le paragraphe précédent afin de rendre ce langage encore plus attrayant. Par ailleurs, MeLa pourrait potentiellement évoluer pour être utilisé pour la programmation de nombreux systèmes d'acquisition de données pluridisciplinaires.

Références

Atzori, L., Iera, A., and Morabito, G. (2010). The internet of things : A survey. *Computer Networks*, 54(15) :2787 – 2805.

Béziers la Fosse, T., Mottu, J.-M., Tisi, M., and Sunyé, G. (2018). Characterizing a Source Code Model with Energy Measurements. In *Workshop on Measurement and Metrics for Green and Sustainable Software Systems (MeGSuS)*, Oulu, Finland.

Hill, A. P., Prince, P., Piña Covarrubias, E., Doncaster, C. P., Snaddon, J. L., and Rogers, A. (2018). Audiomoth : Evaluation of a smart open acoustic device for monitoring biodiversity and the environment. *Methods in Ecology and Evolution*, 9(5) :1199–1211.

Hill, A. P., Prince, P., Snaddon, J. L., Doncaster, C. P., and Rogers, A. (2019). Audiomoth : A low-cost acoustic device for monitoring biodiversity and the environment. *HardwareX*, 6 :e00073.

Kim, B. W. and Joo, S. (2009). A New Commissioning and Deployment Method for Wireless Sensor Networks. In *2009 Third International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, pages 232–237, Sliema, Malta.

Kocakulak, M. and Butun, I. (2017). An overview of wireless sensor networks towards internet of things. In *2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 1–6.

Lazreg, S., Cordy, M., Collet, P., Heymans, P., and Mosser, S. (2019). Multifaceted automated analyses for variability-intensive embedded systems. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 854–865. IEEE Press.

Prince, P., Hill, A., Piña-Covarrubias, E., Doncaster, C., Snaddon, J., and Rogers, A. (2019). Deploying acoustic detection algorithms on low-cost, open-source acoustic sensors for environmental monitoring. *Sensors*, 19 :553.

Zhang, L., B. Jack, L., and Nandi, A. K. (2005). Fault detection using genetic programming. *Mechanical Systems and Signal Processing*, 19(2) :271–289.

Bibliographie

Atzori, L., Iera, A., and Morabito, G. (2010). The internet of things : A survey. *Computer Networks*, 54(15) :2787 – 2805.

Barnes, C. R., Best, M. M. R., and Zielinski, A. (2008). The NEPTUNE Canada regional cabled ocean observatory. *Sea Technology*, 49(7) :10–14.

Baumgartner, M. F., Stafford, K. M., and Latha, G. (2018). Near real-time underwater passive acoustic monitoring of natural and anthropogenic sounds. In Venkatesan, R., Tandon, A., D'Asaro, E., and Atmanand, M. A., editors, *Observing the Oceans in Real Time*, pages 203–226. Springer International Publishing, Cham.

Berry, G. and Gonthier, G. (1992). The Esterel synchronous programming language : design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152.

Béziers la Fosse, T., Mottu, J.-M., Tisi, M., and Sunyé, G. (2018). Characterizing a Source Code Model with Energy Measurements. In *Workshop on Measurement and Metrics for Green and Sustainable Software Systems (MeGSuS)*, Oulu, Finland.

Bingham, B., Kraus, N., Howe, B., Freitag, L., Ball, K., Koski, P., and Gallimore, E. (2012). Passive and active acoustics using an autonomous wave glider. *Journal of Field Robotics*, 29(6) :911–923.

Boehme, L., Lovell, P., Biuw, M., Roquet, F., Nicholson, J., Thorpe, S. E., Meredith, M. P., and Fedak, M. (2009). Technical Note : Animal-borne CTD-Satellite Relay Data Loggers for real-time oceanographic data collection. *Ocean Science*, 5(4) :685–695.

Bonnieux, S., Mosser, S., Blay-Fornarino, M., Hello, Y., and Nolet, G. (2019). Model driven programming of autonomous floats for multidisciplinary monitoring of the oceans. In *OCEANS 2019 - Marseille*, pages 1–10, Marseille, France. IEEE.

Bouysse, P. (2014). Geological map of the world, explanatory note, 3rd revised edition at the 1 :35 000 000 scale. Commission For The Geological Map Of The World.

Cakir, E., Adavanne, S., Parascandolo, G., Drossos, K., and Virtanen, T. (2017). Convolutional recurrent neural networks for bird audio detection. *2017 25th European Signal Processing Conference (EUSIPCO)*, pages 1744–1748.

Calel, R., Chapman, S. C., Stainforth, D. A., and Watkins, N. W. (2020). Temperature variability implies greater economic damages from climate change. *Nature Communications*, 11(5028) :1–5.

Davis, R. E., Regier, L. A., Dufour, J., and Webb, D. C. (1992). The Autonomous Lagrangian Circulation Explorer (ALACE). *Journal of Atmospheric and Oceanic Technology*, 9(3) :264–285.

Dawe, T. C., Bird, L., Talkovic, M., Brekke, K., Osborne, D. J., and Etchemendy, S. (2005). Operational Support of Regional Cabled Observatories The MARS Facility. In *Proceedings of OCEANS 2005 MTS/IEEE*, pages 1–6.

DeAntoni, J. and Mallet, F. (2012). Timesquare : treat your models with logical time. In *TOOLS*.

Delange, J. (2017). *AADL In Practice*. Reblochon Development Company.

D'Eu, J., Royer, J., and Perrot, J. (2012). Long-term autonomous hydrophones for large-scale hydroacoustic monitoring of the oceans. In *2012 Oceans - Yeosu*, pages 1–6.

Devi, G. K., Ganasri, B., and Dwarakish, G. (2015). Applications of remote sensing in satellite oceanography : A review. *Aquatic Procedia*, 4 :579 – 584. International Conference on Water Resources, Coastal and Ocean Engineering (ICWRCOE'15).

Feiler, P., Gluch, D., and Hudak, J. (2006). The architecture analysis & design language (aadl) : An introduction.

Ferdinand, C. and Heckmann, R. (2004). ait : Worst-case execution time prediction by static program analysis. In Jacquart, R., editor, *Building the Information Society*, pages 377–383, Boston, MA. Springer US.

Feuillet, N., Jorry, S., Crawford, W. C., Deplus, C., Thinon, I., Jacques, E., Saurel, J. M., Lemoine, A., Paquet, F., Daniel, R., Gaillot, A., Satriano, C., Peltier, A., Aiken, C., Foix, O., Kowalski, P., Laurent, A., Beauducel, F., Grandin, R., Ballu, V., Bernard, P., Donval, J. P., Géli, L., Gomez, J., Pelleau, P., Guyader, V., Rinnert,

E., Besançon, S., Bertil, D., Lemarchand, A., and Vanderwoerd, J. (2019). Birth of a large volcano offshore Mayotte through lithosphere-scale rifting. In *AGU Fall Meeting Abstracts*, volume 2019, pages V52D–01.

Fish, F. E. (2020). Advantages of aquatic animals as models for bio-inspired drones over present AUV technology. *Bioinspiration & Biomimetics*, 15(2) :025001.

Fletcher, B., Bowen, A., Yoerger, D., and Whitcomb, L. (2009). Journey to the challenger deep : 50 years later with the nereus hybrid remotely operated vehicle. *Marine Technology Society Journal*, 43 :65–76.

Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional, 1st edition.

Fretwell, P. T., Staniland, I. J., and Forcada, J. (2014). Whales from space : Counting southern right whales by satellite. *PLOS ONE*, 9(2) :1–9.

Friedlingstein, P., Jones, M. W., O'Sullivan, M., Andrew, R. M., Hauck, J., Peters, G. P., Peters, W., Pongratz, J., Sitch, S., Le Quéré, C., Bakker, D. C. E., Canadell, J. G., Ciais, P., Jackson, R. B., Anthoni, P., Barbero, L., Bastos, A., Bastrikov, V., Becker, M., Bopp, L., Buitenhuis, E., Chandra, N., Chevallier, F., Chini, L. P., Currie, K. I., Feely, R. A., Gehlen, M., Gilfillan, D., Gkritzalis, T., Goll, D. S., Gruber, N., Gutekunst, S., Harris, I., Haverd, V., Houghton, R. A., Hurtt, G., Ilyina, T., Jain, A. K., Joetzjer, E., Kaplan, J. O., Kato, E., Klein Goldewijk, K., Korsbakken, J. I., Landschützer, P., Lauvset, S. K., Lefèvre, N., Lenton, A., Lienert, S., Lombardozzi, D., Marland, G., McGuire, P. C., Melton, J. R., Metzl, N., Munro, D. R., Nabel, J. E. M. S., Nakaoka, S.-I., Neill, C., Omar, A. M., Ono, T., Peregon, A., Pierrot, D., Poulter, B., Rehder, G., Resplandy, L., Robertson, E., Rödenbeck, C., Séférian, R., Schwinger, J., Smith, N., Tans, P. P., Tian, H., Tilbrook, B., Tubiello, F. N., van der Werf, G. R., Wiltshire, A. J., and Zaehle, S. (2019). Global Carbon Budget 2019. *Earth System Science Data*, 11(4) :1783–1838.

Gaur, P. and Tahiliani, M. P. (2015). Operating systems for iot devices : A critical survey. *2015 IEEE Region 10 Symposium*, pages 33–36.

Gentemann, C. L., Scott, J. P., Mazzini, P. L. F., Pianca, C., Akella, S., Minnett, P. J., Cornillon, P., Fox-Kemper, B., Cetinić, I., Chin, T. M., Gomez-Valdes, J., Vazquez-Cuervo, J., Tsontos, V., Yu, L., Jenkins, R., De Halleux, S., Peacock, D., and Cohen, N. (2020). Saildrone : Adaptively Sampling the Marine Environment. *Bulletin of the American Meteorological Society*, 101(6) :E744–E762.

Gobillot, N., Lesire, C., and Doose, D. (2018). A design and analysis methodology for component-based real-time architectures of autonomous systems. *Journal of Intelligent and Robotic Systems*, pages 1–16.

Gomaa, H. (2016a). *Real-Time Software Design for Embedded Systems*. Cambridge University Press.

Gomaa, H. (2016b). *Real-Time Software Design for Embedded Systems*. Cambridge University Press.

Gould, W. J. (2005). From swallow floats to argo—the development of neutrally buoyant floats. *Deep Sea Research Part II : Topical Studies in Oceanography*, 52(3) :529 – 543. Direct observations of oceanic flow : A tribute to Walter Zenk.

Grolleau, E., Hugues, J., Yassine, O., and Henri, B. (2018). *Introduction aux systèmes embarqués temps réel : Conception et mise en oeuvre*. Dunod.

Group, O. M. (2011). Uml profile for marte : Modeling and analysis of real-time embedded systems.

Guinotte, J. and Fabry, V. (2008). Ocean acidification and its potential effects on marine ecosystems. *Annals of the New York Academy of Sciences*, 1134 :320–42.

Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991). The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320.

Halpern, B. S., Walbridge, S., Selkoe, K. A., Kappel, C. V., Micheli, F., D'Agrosa, C., Bruno, J. F., Casey, K. S., Ebert, C., Fox, H. E., Fujita, R., Heinemann, D., Lenihan, H. S., Madin, E. M. P., Perry, M. T., Selig, E. R., Spalding, M., Steneck, R., and Watson, R. (2008). A global map of human impact on marine ecosystems. *Science*, 319(5865) :948–952.

Hello, Y. and Nolet, G. (2020). Floating Seismographs (MERMAIDS). In Gupta, H. K., editor, *Encyclopedia of Solid Earth Geophysics*, pages 1–6, Cham, Switzerland. Springer.

Hello, Y., Royer, J. Y., Rivet, D., Charvis, P., Yegikyan, M., and Philippe, O. (2019). New versatile autonomous platforms for long-term geophysical monitoring in the ocean. In *OCEANS 2019 - Marseille*, pages 1–8.

Hildyard.(Editor), A. (2001). *Endangered Wildlife and Plants of the World, Volume 12*. Marshall Cavendish Corp.

Hill, A. P., Prince, P., Piña Covarrubias, E., Doncaster, C. P., Snaddon, J. L., and Rogers, A. (2018). Audiomoth : Evaluation of a smart open acoustic device for monitoring biodiversity and the environment. *Methods in Ecology and Evolution*, 9(5) :1199–1211.

Hill, A. P., Prince, P., Snaddon, J. L., Doncaster, C. P., and Rogers, A. (2019). Audiomoth : A low-cost acoustic device for monitoring biodiversity and the environment. *HardwareX*, 6 :e00073.

Hine, R., Willcox, S., Hine, G., and Richardson, T. (2009). The Wave Glider : A Wave-Powered autonomous marine vehicle. In *OCEANS 2009*, pages 1–6.

Huang, J., Huang, J., Liu, X., Li, C., Ding, L., and Yu, H. (2018). The global oxygen budget and its future projection. *Science Bulletin*, 63(18) :1180–1186.

Iyenghar, P. and Pulvermüller, E. (2018). A model-driven workflow for energy-aware scheduling analysis of iot-enabled use cases. *IEEE Internet of Things Journal*, 5 :4914–4925.

Joint, I. and Groom, S. B. (2000). Estimation of phytoplankton production from space : current status and future

potential of satellite remote sensing. *Journal of Experimental Marine Biology and Ecology*, 250(1) :233–255.

Joubert, C., Nolet, G., Bonnieux, S., Deschamps, A., Dessa, J.-X., and Hello, Y. (2016). P-Delays from Floating Seismometers (MERMAID), Part I : Data Processing. *Seismological Research Letters*, 87(1) :73–80.

Kakousis, K., Paspallis, N., and Papadopoulos, G. A. (2010). A survey of software adaptation in mobile and ubiquitous computing. *Enterprise Information Systems*, 4(4) :355–389.

Kawaguchi, K., Kaneko, S., Nishida, T., and Komine, T. (2015). *Construction of the DONET real-time seafloor observatory for earthquakes and tsunami monitoring*, pages 211–228. Springer, Berlin, Heidelberg.

Kent, S. (2002). Model driven engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 286–298, London, UK, UK. Springer-Verlag.

Kim, B. W. and Joo, S. (2009). A New Commissioning and Deployment Method for Wireless Sensor Networks. In *2009 Third International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, pages 232–237, Sliema, Malta.

Kocakulak, M. and Butun, I. (2017). An overview of wireless sensor networks towards internet of things. In *2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 1–6.

la Fosse, T. B., Mottu, J.-M., Tisi, M., Rocheteau, J., and Sunyé, G. (2018). Characterizing a source code model with energy measurements. In *MeGSuS@ESEM*.

Lane, N. D., Bhattacharya, S., Mathur, A., Georgiev, P., Forlivesi, C., and Kawsar, F. (2017). Squeezing deep learning into mobile and embedded devices. *IEEE Pervasive Computing*, 16(3) :82–88.

Larsen, K. G., Pettersson, P., and Yi, W. (1997). Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1 :134–152.

Lazreg, S., Cordy, M., Collet, P., Heymans, P., and Mosser, S. (2019). Multifaceted automated analyses for variability-intensive embedded systems. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 854–865. IEEE Press.

Lefevre, D., Zakardkjian, B., and Embarcio, D. (2018). Unique observatories for sea science and particle astrophysics : The EMSO-Antares and EMSO-Western Ionian nodes in the Mediterranean Sea. In *8th Very Large Volume Neutrino Telescope Workshop*, volume 207, page 09004, Dubna, Russia.

Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20 :46–61.

Liu, S., Liu, M., Wang, M., Ma, T., and Qing, X. (2018). Classification of cetacean whistles based on convolutional neural network. In *2018 10th International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 1–5.

Lorincz, K., Chen, B.-r., Waterman, J., Werner-Allen, G., and Welsh, M. (2008). Resource aware programming in the pixie os. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 211–224, New York, NY, USA. ACM.

Louboutin, M., Lange, M., Luporini, F., Kukreja, N., Witte, P. A., Herrmann, F. J., Velesko, P., and Gorman, G. J. (2019). Devito (v3.1.0) : an embedded domain-specific language for finite differences and geophysical exploration. *Geoscientific Model Development*, 12(3) :1165–1187.

Luo, W., Yang, W., and Zhang, Y. (2019). Convolutional neural network for detecting odontocete echolocation clicks. *The Journal of the Acoustical Society of America*, 145 :EL7–EL12.

Ma, B. B. and Nystuen, J. A. (2005). Passive acoustic detection and measurement of rainfall at sea. *Journal of Atmospheric and Oceanic Technology*, 22(8) :1225 – 1248.

Mac Aodha, O., Gibb, R., Barlow, K. E., Browning, E., Firman, M., Freeman, R., Harder, B., Kinsey, L., Mead, G. R., Newson, S. E., Pandourski, I., Parsons, S., Russ, J., Szodoray-Paradi, A., Szodoray-Paradi, F., Tilova, E., Girolami, M., Brostow, G., and Jones, K. E. (2018). Bat detective—deep learning tools for bat acoustic signal detection. *PLOS Computational Biology*, 14(3) :1–19.

Malve, H. (2016). Exploring the ocean for new drug developments : Marine pharmacology. *Journal of Pharmacy And Bioallied Sciences*, 8(2) :83–91.

Manley, J. and Willcox, S. (2010a). The Wave Glider : A persistent platform for ocean science. In *OCEANS'10 IEEE SYDNEY*, pages 1–5.

Manley, J. and Willcox, S. (2010b). The wave glider : A new concept for deploying ocean instrumentation. *IEEE Instrumentation Measurement Magazine*, 13(6) :8–13.

Marcelli, M., Piermattei, V., and Zappalà, G. (2011). Advances in low cost marine technologies. *WIT Transactions on Modelling and Simulation*, 51 :497–507.

Marques, T. A., Thomas, L., Ward, J., DiMarzio, N., and Tyack, P. L. (2009). Estimating cetacean population density using fixed passive acoustic sensors : An example with blainville's beaked whales. *The Journal of the Acoustical Society of America*, 125(4) :1982–1994.

Matsumoto, H., Jones, C., Klinck, H., Mellinger, D. K., Dziak, R. P., and Meinig, C. a. (2013a). Tracking beaked whales with a passive acoustic profiler float. *The Journal of the Acoustical Society of America*, 133(2) :731–740.

Matsumoto, H., Jones, C. R., Klinck, H., Mellinger, D. K., Dziak, R. P., and Meinig, C. (2013b). Tracking beaked whales with a passive acoustic profiler float. *The Journal of the Acoustical Society of America*, 133 2 :731–40.

Mazzullo, A., Stutzmann, E., Montagner, J.-P., Kiselev, S., Maurya, S., Barruol, G., and Sigloch, K. (2017). Anisotropic Tomography Around La Réunion Island From Rayleigh Waves. *Journal of Geophysical Research :*

Solid Earth, 122(11) :9132–9148.

Mcdonald, M., Hildebrand, J., and Mesnick, S. (2006). Biogeographic characterisation of blue whale song worldwide : Using song to identify populations. *Journal of Cetacean Research and Management*, 8.

McPhaden, M. J., Ando, K., Bourlès, B., Freitag, H. P., Lumpkin, R., Masumoto, Y., Murty, V. S. N., Nobre, P., Ravichandran, M., Vialard, J., Vousden, D., and Yu, W. (2009). The global tropical moored buoy array. In *OceanObs'09 : Sustained Ocean Observations and Information for Society*, volume 2, Venice, Italy.

Meindl, A. (1996). Guide to moored buoys and other ocean data acquisition systems. *WMO & IOC Data Buoy Cooperation Panel*.

Meiner, A. (2010). Integrated maritime policy for the European Union — consolidating coastal and marine information to support maritime spatial planning. *Journal of Coastal Conservation*, 14(1) :1–11.

Mesaros, A., Heittola, T., and Virtanen, T. (2016). Metrics for polyphonic sound event detection. *Applied Sciences*, 6 :162.

Miles, R. and Hamilton, K. (2006). *Learning UML 2.0*. O'Reilly Media.

Minnett, P., Alvera-Azcárate, A., Chin, T., Corlett, G., Gentemann, C., Karagali, I., Li, X., Marsouin, A., Marullo, S., Maturi, E., Santoleri, R., Saux Picart, S., Steele, M., and Vazquez-Cuervo, J. (2019). Half a century of satellite remote sensing of sea-surface temperature. *Remote Sensing of Environment*, 233 :111366.

Mussbacher, G., Amyot, D., Breu, R., Bruel, J.-M., Cheng, B. H. C., Collet, P., Combemale, B., France, R. B., Heldal, R., Hill, J. H., Kienzle, J., Schöttle, M., Steimann, F., Stikkolorum, D. R., and Whittle, J. (2014). The relevance of model-driven engineering thirty years from now. In *MoDELS*.

Navet, N. and Fejoz, L. (2016). Cpal : high-level abstractions for safe embedded systems. In *DSM@SPLASH*.

Neumann, B., Vafeidis, A. T., Zimmermann, J., and Nicholls, R. J. (2015). Future coastal population growth and exposure to sea-level rise and coastal flooding - a global assessment. *PLOS ONE*, 10(3) :1–34.

Nguyen Hong Duc, P., Torterotot, M., Samaran, F., White, P. R., Gérard, O., Adam, O., and Cazau, D. (2021). Assessing inter-annotator agreement from collaborative annotation campaign in marine bioacoustics. *Ecological Informatics*, 61 :101185.

Nolet, G., Hello, Y., Lee, S. v. d., Bonnieux, S., Ruiz, M. C., Pazmino, N. A., Deschamps, A., Regnier, M. M., Font, Y., Chen, Y. J., and Simons, F. J. (2019a). Imaging the Galápagos mantle plume with an unconventional application of floating seismometers. *Scientific Reports*, 9(1326) :1–12.

Nolet, G., Hello, Y., van der Lee, S., Bonnieux, S., Ruiz, M. J. C., Pazmino, N. A., Deschamps, A., Regnier, M. M., Font, Y., Chen, Y. J., and Simons, F. J. (2019b). Imaging the Galápagos mantle plume with an unconventional application of floating seismometers. In *Scientific Reports*.

Palter, J. B. (2015). The Role of the Gulf Stream in European Climate. *Annual Review of Marine Science*, 7(1) :113–137.

Parr, T. J. and Quong, R. W. (1995). Antlr : A predicated- ll(k) parser generator. *Softw., Pract. Exper.*, 25 :789–810.

Pelcat, M., Desnos, K., Heulot, J., Guy, C., Nezan, J., and Aridhi, S. (2014). Preesm : A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. In *2014 6th European Embedded Design in Education and Research Conference (EDERC)*, pages 36–40.

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. a. P., and Saraiva, J. a. (2017). Energy efficiency across programming languages : How do energy, time, and memory relate ? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, page 256–267, New York, NY, USA. Association for Computing Machinery.

Pierce, B. C. and Turner, D. N. (2000). Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1) :1–44.

Pilone, D. and Pitman, N. (2005). *UML 2.0 in a Nutshell*. O'Reilly Media.

Pinder, D. (2001). Offshore oil and gas : global resource knowledge and technological change. *Ocean & Coastal Management*, 44(9) :579–600.

Prause, C. R., Bibus, M., Dietrich, C., and Jobi, W. (2016). Software product assurance at the German space agency. *Journal of Software : Evolution and Process*, 28(9) :744–761.

Prince, P., Hill, A., Piña-Covarrubias, E., Doncaster, C., Snaddon, J., and Rogers, A. (2019). Deploying acoustic detection algorithms on low-cost, open-source acoustic sensors for environmental monitoring. *Sensors*, 19 :553.

Riser, S. C., Nystuen, J. A., and Rogers, A. (2008). Monsoon effects in the bay of bengal inferred from profiling float-based measurements of wind speed and rainfall. *Limnol. Oceanogr.*, 53 :2080–2093.

Riser, S. C., Swift, D., and Drucker, R. (2018). Profiling Floats in SOCCOM : Technical Capabilities for Studying the Southern Ocean. *Journal of Geophysical Research (Oceans)*, 123(6) :4055–4073.

Rodrigues da Silva, A. (2015). Model-driven engineering : A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43 :139–155.

Roemmich, D., Johnson, G., Riser, S., Davis, R., Gilson, J., Owens, W., Garzoli, S., Schmid, C., and Ignaszewski, M. (2009). The Argo Program : Observing the Global Ocean with Profiling Floats. *Oceanography*, 22(2) :34–43.

Rolland, L. M., Occhipinti, G., Lognonné, P., and Loevenbruck, A. (2010). Ionospheric gravity waves detected offshore Hawaii after tsunamis. *Geophysical Research Letters*, 37(17).

Roman J, M. J. (2010). The whale pump : marine mammals enhance primary productivity in a coastal basin. *PLoS One*.

Rudnick, D. L., Davis, R. E., Eriksen, C. C., Fratantoni, D. M., and Perry, M. J. (2004). Underwater Gliders for Ocean Research. *Marine Technology Society Journal*, 38(2) :73–84.

Sainath, T. N. and Parada, C. (2015). Convolutional neural networks for small-footprint keyword spotting. *INTERSPEECH*.

Schaetti, N. (2018). Character-based convolutional neural network andresnet18 for twitter author profiling. In *Notebook for PAN at CLEF 2018*.

Schmidt, D. C. (2006). Guest editor's introduction : Model-driven engineering. *Computer*, 39 :25–31.

Selić, B. and Gérard, S. (2014). *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE*. Elsevier.

Sha, L., Klein, M., and Goodenough, J. (1991). Rate monotonic analysis for real-time systems. Technical Report CMU/SEI-91-TR-006, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Shi, W., Cao, J., Zhang, Q., Li, Y., and Xu, L. (2016). Edge Computing : Vision and Challenges. *IEEE Internet of Things Journal*, 3(5) :637–646.

Shiu, Y., Palmer, K., Roch, M., Fleishman, E., Liu, X., Nosal, E.-M., Helble, T., Cholewiak, D., Gillespie, D., and Klinck, H. (2020). Deep neural networks for automated detection of marine mammal species. *Scientific Reports*, 10 :607.

Sigler, M. (2014). The Effects of Plastic Pollution on Aquatic Wildlife : Current Situations and Future Solutions. *Water, Air, & Soil Pollution*, 225(11) :2184.

Sigloch, K. (2013). Short Cruise Report for Research Cruise M101 on RV "METEOR", RHUM-RUM project, Oct-Dec 2013. RHUM-RUM – Seismological Imaging of a mantle plume under La Réunion, western Indian Ocean. Technical report, Universität Hamburg.

Simon, J. D., Simons, F. J., and Nolet, G. (2014). Data and Analysis Methods of the Son-O-Mermaid and MERMAID Experiments. In *AGU Fall Meeting Abstracts*, pages S13B–4448.

Simons, F. J., Nolet, G., Georgief, P., Babcock, J. M., Regier, L. A., and Davis, R. E. (2009). On the potential of recording earthquakes for global seismic tomography by low-cost autonomous instruments in the oceans. *Journal of Geophysical Research : Solid Earth*, 114(B5).

Sindermann, C. J. (1996). *Ocean Pollution : Effects on Living Resources and Humans*. CRC Press.

Song, L., Li, Y., Wang, J., Wang, F., Hu, S., Liu, C., Diao, X., and Guan, C. (2018). Tropical Meridional Overturning Circulation Observed by Subsurface Moorings in the Western Pacific. *Scientific Reports*, 8(7632) :1–8.

Sorber, J., Kostadinov, A., Garber, M., Brennan, M., Corner, M. D., and Berger, E. D. (2007). Eon : A language and runtime system for perpetual systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 161–174, New York, NY, USA. ACM.

Sørensen, M. B., Spada, M., Babeyko, A., Wiemer, S., and Grünthal, G. (2012). Probabilistic tsunami hazard in the Mediterranean Sea. *Journal of Geophysical Research : Solid Earth*, 117(B1).

Sukhovich, A., Bonnieux, S., Hello, Y., Irisson, J.-O., Simons, F. J., and Nolet, G. (2015a). Seismic monitoring in the oceans by autonomous floats. *Nature Communications*, 6(8027) :1–6.

Sukhovich, A., Bonnieux, S., Hello, Y., Irisson, J. O., Simons, F. J., and Nolet, G. (2015b). Seismic monitoring in the oceans by autonomous floats. In *Nature communications*.

Teng, C., Cucullu, S., McArthur, S., Kohler, C., Burnett, B., and Bernard, L. (2009). Buoy vandalism experienced by noaa national data buoy center. In *OCEANS 2009*, pages 1–8.

Venkatesan, R., Ramesh, K., Kishor, A., Vedachalam, N., and Atmanand, M. A. (2018). Best practices for the ocean moored observatories. *Frontiers in Marine Science*, 5 :469.

Voelter, M., Ratiu, D., Schaetz, B., and Kolb, B. (2012). Mbeddr : An Extensible C-Based Programming Language and IDE for Embedded Systems. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications : Software for Humanity*, SPLASH '12, page 121–140, New York, NY, USA. Association for Computing Machinery.

Volkmann, J., Stevens, S., and Newman, E. (1937). A scale for the measurement of the psychological magnitude pitch. *J. Acoust. Soc. Am.*, 8 :208.

Wang, D., Zhang, L., Lu, Z., and Xu, K. (2018). Large-scale whale call classification using deep convolutional neural network architectures. *2018 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC)*, pages 1–5.

White, E. (2011). *Making Embedded Systems : Design Patterns for Great Software*. O'Reilly Media, Inc.

Woodside, C. M., Franks, G., and Petriu, D. C. (2007). The future of software performance engineering. *Future of Software Engineering (FOSE '07)*, pages 171–187.

Wüst, G. (1964). The major deep-sea expeditions and research vessels 1873–1960 : A contribution to the history of oceanography. *Progress in Oceanography*, 2 :1 – 52.

Yang, J., Riser, S., Nystuen, J., Asher, W., and Jessup, A. (2015). Regional rainfall measurements using the passive aquatic listener during the spurs field campaign. *Oceanography*, 28.

Zhang, L., B. Jack, L., and Nandi, A. K. (2005). Fault detection using genetic programming. *Mechanical Systems and Signal Processing*, 19(2) :271–289.

Annexes

Annexe A

Short list of MeLa data types and functions

Table A.1 – Data types

```
# Values
Bool, Int, Float
ComplexInt, ComplexFloat
String

# Arrays
ArrayInt, ArrayFloat
ArrayComplexInt, ArrayComplexFloat

# Buffers
BufferInt, BufferFloat

# Specific types for processing
FFTInt, FFTFloat,
IIRInt, IIRFloat,
FIRInt, FIRFloat,
CDF24Int, CDF24Float,
StaLtaInt, StaLtaFloat,
TriggerInt, TriggerFloat,
DistributionInt, DistributionFloat

# Utils
File
```

Table A.2 – Functions

```
# Array functions
put(); Put a value in an array
get(); Get a value from an array
copy(); Copy an array to another array
toComplex(); Convert an integer or a float to a complex number
toFloat(); Convert an integer to a float
toInt(); Convert a float to an integer
select(); Select a portion of an array to work on
unselect(); Work on the whole array
clear(); Delete all values in the array

# Buffer functions
push(); Add a value to the end of the buffer
toArray(); Copy the content of the buffer to an array

# Processing functions
fft(); Compute a Fourier transform
cdf24(); Compute a CDF24 wavelet transform
cdf24ScalesPower(); Compute the power of each scale of a CDF24
iir(); Infinite impulse response filter
fir(); Finite impulse response filter
stalta(); Short term over long term average
trigger(); Return true for high or low value, or rising or falling edge,
           compared to a threshold
cumulativeDistribution(); Compute the cumulative distribution
max(); Find the maximum value and its index in an array
min(); Find the minimum value and its index in an array
mean(); Compute the mean value of an array
sum(); Sum all the elements of an array
energy(); Compute the energy, defined as the sum of the squared values of
          an array
rms(); Root mean square
stdDev(); Standard deviation
var(); Variance
abs(); Absolute value
add(); Add elements of an array with a value or add two arrays
sub(); Subtract elements of an array with a value or subtract of two arrays
mult(); Multiply elements of an array with a value or divide two arrays
div(); Divide elements of an array with a value or divide two arrays
diff(); Forward finite difference (derivative approximation)
dotProduct(); Dot product of two arrays
negate(); Negates each value of an array
conv(); Convolution between two array
corr(); Correlation between two array
sqrt(); Square root
cos(); Cosinus
sin(); Sinus
log10(); Common logarithm
pow(); Power of n
magnitude(); Magnitude of a complex number

# Utility functions
ascentRequest(); Request the ascent of the float
getTimestamp(); Get the current date with a resolution of 1 second
record(); Record a value, an array, a buffer or a string in a file
getSampleIndex(); Only for simulation. Get the index of current
                  sample readed in the input file
```

Annexe B

MeLa v1.0 reference manual

Contents

B.1 Language description 92
B.1.1 Mission configuration 92
B.1.2 Coordinator 92
B.1.3 Acquisition modes 92
B.1.4 Instructions 93
B.2 Constants 94
B.3 Data types 95
B.3.1 Boolean 95
B.3.2 Integer 95
B.3.3 Float 95
B.3.4 Complex numbers 95
B.3.5 Arrays 95
B.3.6 Buffers 96
B.3.7 FFT 96
B.3.8 IIR 97
B.3.9 FIR 97
B.3.10 CDF24 98
B.3.11 STALTA 98
B.3.12 Trigger 99
B.3.13 Distribution 99
B.3.14 File 100
B.4 Array and Buffer functions 100
B.4.1 clear 100
B.4.2 copy 101
B.4.3 get 101
B.4.4 put 102
B.4.5 push 102
B.4.6 select 103
B.4.7 unselect 103

B.5 Math functions **104**
B.5.1 abs 104
B.5.2 add 104
B.5.3 cdf24 105
B.5.4 cdf24ScalesPower 105
B.5.5 conv 106
B.5.5.1 corr 106
B.5.5.2 cos 107
B.5.6 cumulativeDistribution 107
B.5.7 diff 108
B.5.8 div 108
B.5.9 dotProduct 109
B.5.10 energy 109
B.5.11 fft 110
B.5.12 fir 110
B.5.13 iir 111
B.5.14 log10 111
B.5.15 magnitude 111
B.5.16 max 112
B.5.17 mean 112
B.5.18 min 113
B.5.19 mult 113
B.5.20 negate 114
B.5.21 pow 114
B.5.22 rms 114
B.5.23 sin 115
B.5.24 sqrt 115
B.5.25 stalta 116
B.5.26 stdDev 116
B.5.27 sub 116
B.5.28 sum 117
B.5.29 trigger 117
B.5.30 var 118
B.6 Utility functions **118**
B.6.1 ascentRequest 118
B.6.2 convert 119
B.6.3 getSampleIndex 119
B.6.4 getTimestamp 120
B.6.5 record 120

B.1 Language description

A MeLa application is composed of three main parts :

- The mission configuration to define basic parameters of the instrument.
- The coordinator to define when to execute the acquisition mode(s).
- The acquisition mode(s) to acquire and process the sensors data. There are two types of acquisition modes, the `ContinuousAcqMode` that processes data without stopping, and the `ShortAcqMode` that processes only one packet of data.

```
Mission:
    ...

Coordinator:
    ...

ContinuousAcqMode acq1:
        ...

ShortAcqMode acq2:
        ...
```

B.1.1 Mission configuration

The mission configuration allows to define the depth of the dive, and the maximum time that the instrument can pass at this depth.

```
Mission:
    ParkTime: 14400 minutes;
    ParkDepth: 1500 meters;
```

B.1.2 Coordinator

The coordinator defines when to execute an acquisition mode during the steps of a dive (i.e., descent, park and ascent). For `ShortAcqMode`, a time interval between each execution must be defined. This is not needed for `ContinuousAcqMode` that never stops during the step in which it is executed.

```
Coordinator:
    DescentAcqModes: acq1;
    ParkAcqModes: acq1, acq2 every 10 minutes;
    AscentAcqModes: acq2 every 10 minutes;
```

B.1.3 Acquisition modes

A `ContinuousAcqMode` processes data in a streamed way, without stopping. It is more adapted to monitor sporadic events (i.e., that appen from time to time), but it can use a lot of processor time, especially if the sampling of the sensor is high. A `ContinuousAcqMode` is divided in several parts :

- The `Input` part allows defining the input sensor to use and its configuration, for example it can be the hydrophone with a sampling frequency of 200 Hz.
- The `Variables` part allows defining the variables used for data processing. A list of data types is given in section B.3.
- The `RealTimeSequence` part contains instructions to process the data in real time. This sequence is executed in a loop each time time a packet of data is sent by the sensor. It gives the guarantee that all the data will be processed, without missing a sample, such that the data cannot be truncated. Only one `RealTimeSequence` can be defined.
- The `ProcessingSequence` part is optional but can be used for instructions with an execution time too long for the `RealTimeSequence`. During the execution of this sequence some data sent by the sensor can be missed. It is possible to define several `ProcessingSequence` that can be called from the `RealTimeSequence` or from another `ProcessingSequence`.

```
ContinuousAcqMode acq1:

Input:
    HydrophoneBF(40);

Variables:
    Int i;
    ArrayFloat array(10);

RealTimeSequence seq1:
    ...
endseq;

ProcessingSequence seq2:
    ...
endseq;
```

A `ProcessingAcqMode` has a very similar structure, the only difference is that it does not contain a `RealTimeSequence` because only one packet of data is processed.

B.1.4 Instructions

Instructions are called inside the sequences of instructions (i.e., the RealTimeSequence or ProcessingSequence). The instructions can be :

- An operation, such as `c = a + b`. Only two operands are accepted in the current version of MeLa. The operands must be integers or floats. The possible operators are `+`, `-`, `*` and `/`.
- A function call, such as `mean(array, r);` which computes the mean values of the array and puts the result into the `r` variable.
- A `if` condition, such `if a > 10 && b > c`. The comparison operators are `<`, `<=`, `>`, `>=`, `==`, `&&`, `||`. A condition must also contain a probability, such as `@probability = 1 per hour`. These probabilities are used by MeLa to compute the battery

lifetime of the float and the amount of data transmitted each month. The possible time units are `sec`, `min`, `hour`, `day` and `week`.

- A `for` loop, such as `for i, v in array`. Each loop iteration reads an element of the array from the first to the last, the index of the current element is put in the `i` variable, and its value in the `v` variable.

```
RealTimeSequence seq1:

  /* Add two variables */
  c = a + b;

  /* If variables exceed a value */
  if a > 10 && b > c:
    @probability = 1 per hour
    */ Compute the mean of the array */
    mean(array, r);
  endif;

  /* Add each element of the array to the a variable */
  for i, v in array:
    a = a + v;
  endfor;

endseq;
```

B.2 Constants

Constants can be used to set the parameters of variables or functions. There are three types of constants that are integers, floating point numbers and strings.

Integer constants are defined as real numbers such as `8467` or `-16`. They are encoded on 32 bits. The largest possible value of a 32 bits integer is 2147483647, the smallest positive is 1 and the smallest negative is -2147483648.

Floating point numbers must be defined with a fractional part such as `8467.54` or `-16.45532`. The fractional part must be defined even if it is null to be recognized as a floating point number by the language, for example `10.0`, otherwise the number will be considered as an integer. It is also possible to define floating point numbers under an exponential form such as `+6.840015400e-01`. Floating point numbers are also encoded on 32 bits. The largest possible value of a floating point number is 3.4028235×10^{38}, the smallest positive is 1.175494×10^{-38} and the smallest negative is $-3.4028235 \times 10^{38}$.

Strings are written between quotation marks, such as `"This is a string"`. MeLa does not include functions to manipulate strings, however they can be used as separator for data recorded on files.

B.3 Data types

All data types are presented with the following pattern :

```
DataType variableName;
```

or if parameters can be specified :

```
DataType variableName(parameter1, parameter2, ...);
```

B.3.1 Boolean

Boolean variables are declared as :

```
Bool b;
```

B.3.2 Integer

Integer variables are declared as :

```
Int i;
```

B.3.3 Float

Floating point variables are declared as :

```
Float f;
```

B.3.4 Complex numbers

Complex numbers contain an imaginary part and a real part, they are mainly used to compute Fast Fourier Transform. They can be either integers of floating point numbers.

```
ComplexInt ci;
ComplexFloat cf;
```

B.3.5 Arrays

Arrays can be defined for integers, floating points and complex numbers.

```
ArrayInt ai(length, initvs);
ArrayFloat af(length, initvs);
ArrayComplexInt aci(length);
ArrayComplexFloat acf(length);
```

Parameters :

1. `length` is the length of the array.
2. `initvs` are optional parameters to initialize the values of the array. The number of values must be equal to the length of the array. Arrays of complex numbers cannot be initialized in the current version of MeLa.

Examples :

```
ArrayInt ai1(3);
ArrayInt ai2(10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
ArrayFloat af1(8);
ArrayFloat af2(5, 1.0, 2.0, 3.0, 4.0, 5.0);
ArrayComplexInt aci(100);
ArrayComplexFloat acf(20);
```

B.3.6 Buffers

Buffers are similar to arrays but it is possible to append data at the end of a buffer whereas it is not for an array. If the buffer is full the older data are overwritten, this is the principle of a circular buffer in which data can be appended indefinitely.

```
BufferInt ai(length);
BufferFloat bf(length);
```

Parameter :

1. `length` is the length of the buffer.

Examples :

```
BufferInt bi(10);
BufferFloat bf(10);
```

B.3.7 FFT

This data type allows to declare variables that are necessary to compute Fast Fourier Transform (FFT). The FFT can be either integer or floating points arithmetic. The function to process the FFT is presented in section B.5.11.

```
FFTInt x(npow2);
FFTFloat x(npow2);
```

Parameter :

1. `npow2` is the size of the FFT. It must be an integer with a power of 2 value comprised between 32 and 4096 (*i.e.*, 32, 64, 128, 256, 512, 1024, 2048 or 4096).

Examples :

```
FFTInt fi(128);
FFTFloat ff(1024);
```

B.3.8 IIR

this data type allows to declare variables necessary for integer Infinite Impulse Response filters (IIR). The IIR can be either integer or floating points arithmetic.

```
IIRInt x(norder, dorder, ncoeffs, dcoeffs);
IIRFloat x(norder, dorder, ncoeffs, dcoeffs);
```

Parameters :

1. `norder` is the number of coefficients at the numerator.
2. `dorder` is the number of coefficients at the denominator.
3. `ncoeffs` are the coefficients of the numerator, the number of values must be equal to the numerator number of coefficients.
4. `dcoeffs` are the coefficients of the denominator, the number of values must be equal to the denominator number of coefficients.

Examples :

```
IIRInt ii(2, 5, 1, -2, 1, -2, 3, -4, 5);
```

corresponds to the filter $H(z) = \frac{y(z)}{x(z)} = \frac{1-2z^{-1}}{1-2z^{-1}+3z^{-2}-4z^{-3}+5z^{-4}}$

```
IIRFloat if(2, 5, 1.0, -0.2, 1.0, -0.2, 0.3, -0.4, 0.5);
```

corresponds to the filter $H(z) = \frac{y(z)}{x(z)} = \frac{1-0.2z^{-1}}{1-0.2z^{-1}+0.3z^{-2}-0.4z^{-3}+0.5z^{-4}}$

B.3.9 FIR

Finite Impulse Response filters (FIR) are defined as IIR but do not have a denominator.

```
FIRInt x(norder, ncoeffs);
FIRFloat x(norder, ncoeffs);
```

Parameters :

1. `norder` is the number of coefficients.
2. `ncoeffs` are the coefficients, the number of values must be equal to the number of coefficients.

Examples :

```
FIRInt fi(2, 1, -2);
FIRFloat ff(2, 0.1, -0.2);
```

corresponds to the filters $H(z) = \frac{y(z)}{x(z)} = 1 - 2z^{-1}$ and $H(z) = 0.1 - 0.2z^{-1}$

B.3.10 CDF24

This datatype allows to declare variables necessary to compute a CFD24. The CDF24 is a specific implementation of a wavelet transform originally used in the Mermaid algorithm [1]. A wavelet transform is basically equivalent to a bank of band pass filter.

```
CDF24Int x(nscales, nsamples);
CDF24Float x(nscales, nsamples);
```

Parameters :

1. `nscales` is the number of scales of the transform (*i.e.*, number of frequency bands).
2. `nsamples` is the number of samples to process.

Examples :

```
CDF24Int cdi(6, 12000);
CDF24Float cdf(6, 12000);
```

B.3.11 STALTA

This datatype allows to declare variables necessary for the Short Term Average over Long Term Average (STA/LTA). The STA/LTA algorithm is commonly found for seismic processing.

```
StaLtaInt x(staLenght, ltaLenght, ltaDelay, scaling);
StaLtaFloat x(staLenght, ltaLenght, ltaDelay);
```

Parameters :

1. `staLenght` is the length of the short term average.
2. `ltaLenght` is the length of the long term average.
3. `ltaDelay` is a delay for the beginning of the LTA.
4. `scaling` is a scaling factor for the ratio. It allows to obtain a better precision when integer arithmetic is used and the ratio of the average is close to 1. For example, instead of a ratio passing from 0 to 1 without intermediate values, a scaling factor of 10 allows to pass from 0 to 10 with all intermediate integer values.

Examples :

```
StaLtaInt sti(400, 4000, 400, 1000);
StaLtaFloat stf(400, 4000, 400);
```

In these examples, the STA has a length of 400 samples, the LTA has a length of 4000 samples and is delayed of 400 samples (*i.e.*, the LTA average the samples between the index 400 and 4399). The ratio of the integer implementation is multiplied by 1000.

1. Sukhovich et al., (2011), "Automatic discrimination of underwater acoustic signals generated by teleseismic P-waves : A probabilistic approach", Geophys. Res. Lett., 38 :L18605

B.3.12 Trigger

This datatype allows to declare variables necessary for the trigger algorithm. The trigger is used to detect changes of a signal compared to to a threshold value. The function that processes the trigger (section B.5.29) return a boolean value depending of the parameters defined here.

```
TriggerInt x(mode, threshold, delay, minInterval);
TriggerFloat x(mode, threshold, delay, minInterval);
```

Parameters :

1. `mode` is the mode of activation of the trigger, there are four possible modes :
   (a) `HIGH` : the trigger will return `true` if the signal is above the threshold value.
   (b) `LOW` : the trigger will return `true` if the signal is under the threshold value.
   (c) `RISING_EDGE` : the trigger will return `true` if the signal passes above the threshold value, but only for the instant when it passes from a lower to a higher level.
   (d) `FALLING_EDGE` : the trigger will return `true` if the signal passes under the threshold value, but only the instant when it passes from a higher to a lower level.
2. `threshold` is the threshold to compare with the signal.
3. `delay` is a delay to activate the trigger. It can be used to wait a number of `delay` samples after a signal passes over a threshold in order to have more data to process.
4. `minInterval` is the minimum sample numbers between each trigger, it can be used to ignore successive triggers close to each others.

Examples :

```
TriggerInt ti(RISING_EDGE, 2500, 4000, 10000);
TriggerFloat tf(HIGH, 2.5, 4000, 10000);
```

In the first example, the trigger will return `true` if the signal **pass** above a value of `2500`. In the second example, the trigger will return `true` if the signal **is** above a value of `2.5`. For both examples, the trigger is always delayed of 4000 samples and there cannot be less than 10000 samples between each trigger.

B.3.13 Distribution

This type of data is used by the function that computes the cumulative distribution (section B.5.6).

```
DistributionInt x(length, xvalues, yvalues);
DistributionFloat x(length, xvalues, yvalues);
```

Parameters :

1. `length` is the length of the distribution.

2. `xvalues` is the x-axis values, the number of values must be equal to the length of the distribution.
3. `yvalues` is the y-axis values, the number of values must be equal to the length of the distribution.

Examples :

```
DistributionInt di(5,
                   1, 2, 3, 4, 5,
                   3, 6, 9, 8, 2);
DistributionFloat df(5,
                     1.0, 1.1, 1.2, 1.3, 1.4,
                     129.0, 326.0, 446.0, 290.0, 230.0);
```

In these examples, both distributions contains 5 numbers, where the x values are given in the second line of each distribution and the y values are given in the third line of each distribution.

B.3.14 File

To declare a file for recording data. The files cannot be read. All data recorded in a file are transmitted trough satellite.

```
File f;
```

No parameters are required.

B.4 Array and Buffer functions

B.4.1 clear

Set all values of an array or a buffer to 0.

```
clear(ArrayInt array);
clear(ArrayFloat array);
clear(ArrayComplexInt array);
clear(ArrayComplexFloat array);
clear(BufferInt buffer);
clear(BufferFloat buffer);
```

Parameters :

1. `array` is the array to clear.

Example :

```
ArrayInt a(5);
put(a, 0, 56);
put(a, 1, 1896);
put(a, 2, 89);
put(a, 3, 67);
```

```
put(a, 4, 156);
clear(a);
```

In this example, some values are put in the `a` array and then it is cleared which means that the array will be filled with zero values.

B.4.2 copy

Copies the content of array1 starting from index1 and for the specified length to the array2 starting from index2.

```
copy(ArrayInt src, Int isrc,
            ArrayInt dst, Int idst, Int len);
copy(ArrayFloat src, Int isrc,
            ArrayInt dst, Int idst, Int len);
copy(ArrayComplexInt src, Int isrc,
            ArrayInt dst, Int idst, Int len);
copy(ArrayComplexFloat src, Int isrc,
            ArrayInt dst, Int idst, Int len);
```

Parameters :

1. `src` is the array to copy.
2. `isrc` is the index from which to start the copy.
3. `dst` is the array in which to realize the copy.
4. `idst` is the index from which to start the copy.
5. `len` is the length to copy.

Example :

```
ArrayInt a(10);
ArrayInt b(10);
ArrayInt c(20);
copy(a, 0, c, 0, 10);
copy(b, 5, c, 10, 5);
```

In this example, the first call to the `copy` function copies the content of `a` in the first half part of `c` and the second call to the `copy` function copies half of the content of 'b' in the second half part of `c`.

B.4.3 get

Get a value in an array at a specified index.

```
get(ArrayInt array, Int index, Int value);
get(ArrayFloat array, Int index, Float value);
get(ArrayComplexInt array, Int index, ComplexInt value);
get(ArrayComplexFloat array, Int index, ComplexFloat value);
```

Parameters :

1. `array` is the array in which to get a value.
2. `index` is the index where to get the value.
3. `value` is the variable that will contain the value.

Example :

```
ArrayFloat a(10);
Float v;
get(a, 5, v);
```

In this example, the value at index 5 of the `a` array is put into the `v` variable.

B.4.4 put

Put a value in an array at a specified index.

```
put(ArrayInt array, Int index, Int value);
put(ArrayFloat array, Int index, Float value);
put(ArrayComplexInt array, Int index, ComplexInt value);
put(ArrayComplexFloat array, Int index, ComplexFloat value);
```

Parameters :

1. `array` is the array in which to put a value.
2. `index` is the index where to put the value.
3. `value` is the value to put in the array.

Example :

```
ArrayFloat a(10);
Float v;
put(a, 5, v);
put(a, 8, 1.6);
```

In this example, an array of floating point numbers called `a` and a variable called v are declared. The value of `v` is put at index 5 of the array and a value of `1.6` is put at index 8 of the array.

B.4.5 push

Add a value to the end of the buffer.

```
push(BufferInt buffer, Int value);
push(BufferFloat buffer, Float value);
push(BufferInt buffer, ArrayInt value);
push(BufferFloat buffer, ArrayFloat value);
```

Parameters :

1. `buffer` is the buffer to fill.

2. `value` is the value used to fill the buffer (it can be an array).

Example :

```
BufferInt b(100);
ArrayInt a(10);

/* Add values of the array to the end of the buffer */
push(b, a);
```

In this example, the 10 values contained in the `a` array are put into the `b` buffer.

B.4.6 select

Select a specific portion of an array to work on. Once a portion of the array is selected all the following operation are done on the selected portion of the array, until it is unselected using the 'unselect' function.

```
select(ArrayInt array, Int index1, Int index2);
select(ArrayFloat array, Int index1, Int index2);
select(ArrayComplexInt array, Int index1, Int index2);
select(ArrayComplexFloat array, Int index1, Int index2);
```

Parameters :

1. `array` is the array to select.
2. `index1` is the index where to start the selected portion.
3. `index2` is the index where to stop the selected portion.

Example :

```
ArrayInt a(100);
select(a, 50, 69);
```

In this example, we select a portion of the `a` array between index 50 and 69 (included), the length of the selected portion is 20.

B.4.7 unselect

Cancel the effect of the select command.

```
unselect(ArrayInt array);
unselect(ArrayFloat array);
unselect(ArrayComplexInt array);
unselect(ArrayComplexFloat array);
```

Parameters :

1. `array` is the array to unselect.

Example :

```
ArrayInt a(10);
unselect(a, 2, 5);
unselect(a);
```

In this example, we first selected a portion of the `a` array and unselected it to come back to its normal size.

B.5 Math functions

B.5.1 abs

Absolute value.

```
abs(Int input, Int output);
abs(Float input, Float output);
abs(ArrayInt input, ArrayInt output);
abs(ArrayFloat input, ArrayFloat output);
```

Parameters :

1. `input` is the input value.
2. `output` is the absolute value of the input value.

Example :

```
ArrayInt a(10);
ArrayInt b(10);

/* Compute the absolute values of a and put the result in b */
abs(a, b);
```

B.5.2 add

Add elements of an array (with a value) or add two arrays.

```
add(ArrayInt input1, ArrayInt input2, ArrayInt output);
add(ArrayFloat input1, ArrayFloat input2, ArrayFloat output);
TODO: add with integer or float
```

Parameters :

1. `input1` is the left operand.
2. `input2` is the right operand.
3. `output` is result of the addition of the two array (element by element).

Example :

```
ArrayInt a(10);
ArrayInt b(10);
ArrayInt c(10);
```

```
/* Add each element of a and b and put the result in c */
abs(a, b, c);
```

B.5.3 cdf24

Compute a CDF24 wavelet transform. Parameters of the CDF24 must be defined in the variable `cdf24v`. Input data are overwritten by the results of the transform.

```
cdf24(CDF24Int cdf24v, ArrayInt array);
cdf24(CDF24Float cdf24v, ArrayFloat array);
```

Parameters :

1. `cdf24v` is the variable which contain parameters to process the wavelet transform.
2. `array` is the array to process.

Example :

```
CDF24Int cdi(6, 12000);
ArrayInt a(12000);

/* Compute the CDF24 wavelet transform of the array a */
cdf24(cdi, a);
```

B.5.4 cdf24ScalesPower

Compute the power of each scale of the CDF24 from array1. The index1 and index2 values allow to select a subset of the CDF24. The results are put in array2 which must have a size at least equal to the number of computed scales.

```
cdf24ScalesPower(CDF24Int cdf24v, ArrayInt array1,
                 Int index1, Int index2, ArrayInt array2);
cdf24ScalesPower(CDF24Float cdf24v, ArrayFloat array1,
                 Int index1, Int index2, ArrayFloat array2);
```

Parameters :

1. `cdf24v` is the variable which contains parameters of the wavelet transform.
2. `array1` is the array containing the wavelet transform.
3. `index1` is the index at which the processing of the power must start.
4. `index2` is the index at which the processing of the power must end.
5. `array2` is the array containing the power for each scale.

Example :

```
CDF24Int cdi(6, 12000);
ArrayInt a(12000);
ArrayInt r(6);

/* Compute the power on the first half part of the signal */
cdf24(cdi, a);
cdf24ScalesPower(cdi, a, 0, 5999, r);
```

In this example, the CDF24 wavelet of the `a` array is processed, then the power of each scale contained in the `a` array is computed and the result is put in the `r` array. Only the first half part of the signal (from index 0 to index 5999) is used to compute the power.

B.5.5 conv

Convolution between two arrays.

```
conv(ArrayInt input1, ArrayInt input2, ArrayInt output);
conv(ArrayFloat input1, ArrayFloat input2, ArrayFloat output);
```

Parameters :

1. `input1` is the left operand.
2. `input2` is the right operand.
3. `output` is the result of the convolution between `input1` and `input2`.

Example :

```
ArrayInt a(100);
ArrayInt b(100);
ArrayInt c(100);

/* Compute the convolution of array a and b
   and put the result in c */
conv(a, b, c);
```

B.5.5.1 corr

Correlation between two arrays.

```
corr(ArrayInt input1, ArrayInt input2, ArrayInt output);
corr(ArrayFloat input1, ArrayFloat input2, ArrayFloat output);
```

Parameters :

1. `input1` is the left operand.
2. `input2` is the right operand.
3. `output` is the result of the correlation between `input1` and `input2`.

Example :

```
ArrayInt a(100);
ArrayInt b(100);
ArrayInt c(100);

/* Compute the correlation of array a and b
   and put the result in c */
conv(a, b, c);
```

B.5.5.2 cos

Cosine.

```
cos(ArrayInt input, ArrayInt output);
cos(ArrayFloat input, ArrayFloat output);
```

Parameters :

1. `input` is the input array.
2. `input` is the result of the cosine operation.

Example :

```
ArrayInt a(100);
ArrayInt b(100);

/* Compute the cosine of array a
   and put the result in b */
cos(a, b);
```

B.5.6 cumulativeDistribution

Compute the cumulative distribution. The distribution must be defined when declaring the variable `distributionv`. This function sums the y-values of the distribution until to reach the limit that is compared to the x-values of the distribution.

```
cumulativeDistribution(DistributionInt distributionv,
                       Int limit, Int result);
cumulativeDistribution(DistributionFloat distributionv,
                       Float limit, Float result);
```

Parameters :

1. `distributionv` is the distribution variable (section B.3.13).
2. `limit` is the limit until which is computed the cumulative distribution.
3. `result` is the result of the cumulative distribution.

Example :

```
DistributionInt dist(100);
Int lim;
Int res;

/* Compute the cumulative distribution until
   to reach the lim value */
cumulativeDistribution(dist, lim, res);
```

B.5.7 diff

Forward finite difference (derivative approximation). The equation used for this algorithm is $output[i] = input[i+1] - input[i]$ and for the last element of the array $output[i] = input[i]$ (but this last element should not exist since $i+1$ did not).

```
diff(ArrayInt input, ArrayInt output);
diff(ArrayFloat input, ArrayFloat output);
```

Parameters :

1. `input` is the input array.
2. `output` is the result of the forward finite difference.

Example :

```
ArrayInt data(100);
ArrayInt res(100);

/* Compute the forward finite difference
   of data and put the result in res */
diff(data, res);
```

B.5.8 div

Divide elements of an array with a value, $a[i]/v$, or divide two arrays element by element, $a1[i]/a2[i]$.

```
div(ArrayInt input1, Int input2, ArrayInt output);
div(ArrayFloat input1, Float input2, ArrayFloat output);
div(ArrayInt input1, ArrayInt input2, ArrayInt output);
div(ArrayFloat input1, ArrayFloat input2, ArrayFloat output);
```

Parameters :

1. `input1` is the array to divide.
2. `input2` is the array or value used to divide `input1`.
3. `output` is the result of the division.

Example :

```
ArrayFloat in1(100);
ArrayFloat in2(100);
ArrayFloat res(100);

/* Divide each element of the array by 20 */
div(in1, 20.0, res);

/* Divide in1 by in2 element by element */
div(in1, in2, res);
```

B.5.9 dotProduct

Dot product of two arrays.

```
dotProduct(ArrayInt input1, ArrayInt input2, Float output);
dotProduct(ArrayFloat input1, ArrayFloat input2,
                                              Float output);
```

Parameters :

1. `input1` is the first array.
2. `input2` is the second array.
3. `output` is the result of the dot product, always returned in a float variable.

Example :

```
ArrayInt in1(100);
ArrayInt in2(100);
Float res;

/* Dot product of in1 and in2, the result is put in
   a float variable */
dotProduct(in1, in2, res);
```

B.5.10 energy

Compute the energy, defined as the sum of the squared values of an array.

```
energy(ArrayInt input, Int output);
energy(ArrayFloat input, Float output);
```

Parameters :

1. `input` is the array to process.
2. `output` is the result.

Example :

```
ArrayFloat in(100);
ArrayFloat res(100);

/* Compute energy of in */
energy(in, res);
```

B.5.11 fft

Compute a Fast Fourier Fourier transform. Parameters of the FFT must be defined in the variable 'fftv'. Input data are overwritten by the results of the transform.

```
fft(FFTInt fftv, ArrayComplexInt array);
fft(FFTFloat fftv, ArrayComplexFloat array);
```

Parameters :

1. `fftv` is the variable containing parameters to process the FFT.
2. `array` is the array to process.

Example :

```
FFTInt fi(128);
ArrayComplexInt ai(128);

/* Compute the fft of the complex array ai */
fft(fi, ai);
```

B.5.12 fir

Finite impulse response filter. The coefficients of the filter must be set in the variable `firv`.

```
fir(FIRInt firv, ArrayInt input, ArrayInt output);
fir(FIRFloat firv, ArrayFloat input, ArrayFloat output);
```

Parameters :

1. `firv` is the variable containing the filter parameters.
2. `input` is the data to process.
3. `output` is the processed data.

Example :

```
FIRFLoat firv(2, 0.1, -0.2);
ArrayFloat in(100);
ArrayFloat res(100);

/* Compute the fft of the complex array ai */
fir(firv, in, res);
```

B.5.13 iir

Infinite impulse response filter. The coefficients of the filter must be set in the variable 'iirv'. The 'array1' must contain the input data, the 'array2' contains the filtered data.

```
iir(IIRInt iirv, ArrayInt array1, ArrayInt array2);
iir(IIRFloat iirv, ArrayFloat array1, ArrayFloat array2);
```

Parameters :

1. `iirv` is the variable containing the filter parameters.
2. `input` is the array to process.
3. `output` is the result of the filter.

Example :

```
IIRInt ii(2, 5,
          1, -2,
          1, -2, 3, -4, 5);
ArrayInt a(100);
ArrayInt b(100);

/* Filter the signal contained in a and put the result in b */
iir(ii, a, b);
```

B.5.14 log10

Common logarithm.

```
log10(Float input, Float output);
log10(ArrayFloat input, ArrayFloat output);
```

Example :

```
ArrayFloat in(5, 1.0, 2.0, 3.0, 4.0, 5.0);
ArrayFloat res(5);

/* Compute the common logarithm of a */
log10(in, res);
```

B.5.15 magnitude

Magnitude of a complex number.

```
magnitude(ArrayComplexInt input, ArrayInt output)
magnitude(ArrayComplexFloat input, ArrayFloat output)
```

Parameters :

1. `input` is the complex array to process.
2. `ouptut` is the magnitude of the input.

Example :

```
ArrayComplexInt in(100);
ArrayInt res(100);

/* Compute the magnitude of the complex number */
magnitude(in, res);
```

B.5.16 max

Find the maximum value and its index in an array.

```
max(ArrayInt array, Int value, Int index);
max(ArrayFloat array, Float value, Int index);
```

Parameters :

1. `array` is the array to process.
2. `value` is the maximum value found in the array.
3. `index` is the index of the array at which the maximum value is found.

Example :

```
ArrayInt in(100);
Int maxVal;
Int iMaxVal;

/* Search the maximum value of the array and put the value
   in maxVal and the index in iMaxVal */
max(in, maxVal, iMaxVal);
```

B.5.17 mean

Compute the mean value of an array.

```
mean(ArrayInt array, Int result);
mean(ArrayFloat array, Float result);
```

Parameters :

1. `array` is the array.
2. `result` is the mean of the array.

Example :

```
ArrayInt a(100);
Int mean;

/* Compute the mean of a */
mean(a, mean);
```

B.5.18 min

Find the minimum value and its index in an array.

```
min(ArrayInt array, Int value, Int index);
min(ArrayFloat array, Float value, Int index);
```

Parameters :

1. `array` is the array to process.
2. `value` is the minimum value found in the array.
3. `index` is the index of the array at which the minimum value is found.

Example :

```
ArrayInt in(100);
Int minVal;
Int iMinVal;

/* Search the minimum value of the array and put the value
   in minVal and the index in iMinVal */
max(in, minVal, iMinVal);
```

B.5.19 mult

Multiply elements of an array with a value or divide two arrays.

```
mult(ArrayInt input1, Int input2, ArrayInt output);
mult(ArrayFloat input1, Float input2, ArrayFloat output);
mult(ArrayInt input1, ArrayInt input2, ArrayInt output);
mult(ArrayFloat input1, ArrayFloat input2, ArrayFloat output);
```

Parameters :

1. `input1` is the first operand of the multiplication.
2. `input2` is the second operand of the multiplication.
3. `output` is the result of the multiplication.

Example :

```
ArrayInt in1(100);
ArrayInt in2(100);
Int res;

/* Multiply each element of the in1 array by 534 */
mult(in1, 534, res);

/* Multiply the in1 and in2 array element by element */
mult(in1, in2, res);
```

B.5.20 negate

Negates each value of an array.

```
negate(ArrayInt input, ArrayInt output);
negate(ArrayFloat input, ArrayFloat output);
```

Parameters :

1. `input` is the array to negate.
2. `output` is the negative of array.

Example :

```
ArrayInt input(100);
ArrayInt output(100);

/* Negate the input array */
negate(input, output);
```

B.5.21 pow

Power of n.

```
pow(Float input, Float power, Float output);
pow(ArrayFloat input, Float power, ArrayFloat output);
```

Parameters :

1. `input` is the base.
2. `power` is the exponent.
3. `output` is the result.

Example :

```
ArrayFloat in(100);
ArrayFloat res(100);

/* Compute the power of 12 of each element of the in array */
pow(in, 12, res);
```

B.5.22 rms

Root mean square.

```
rms(ArrayInt input, Int output);
rms(ArrayFloat input, Float output);
```

Parameters :

1. `input` is the array to process.
2. `input` is the root mean square of the array.

Example :

```
ArrayFloat in(100);
Float res;

/* Compute the root mean square of the array */
rms(in, res);
```

B.5.23 sin

Sine.

```
sin(ArrayInt input, ArrayInt output);
sin(ArrayFloat input, ArrayFloat output);
```

Parameters :

1. `input` is the array to process.
2. `output` is the sine of each element of the array.

Example :

```
ArrayFloat in(100);
ArrayFloat res(100);

/* Compute sine of each element of the in array */
sin(in, res);
```

B.5.24 sqrt

Square root.

```
sqrt(ArrayInt input, ArrayInt output);
sqrt(ArrayFloat input, ArrayFloat output);
```

Parameters :

1. `input` is the array to process.
2. `output` is the square root of each element of the array.

Example :

```
ArrayFloat in(100);
ArrayFloat res(100);

/* Compute square root of each element of the in array */
sqrt(in, res);
```

B.5.25 stalta

Short term over long term average. The coefficients of the STA/LTA must be set in the variable `staltav`.

```
stalta(StaLtaInt staltav, Int input, Int output);
stalta(StaLtaFloat staltav, Float input, Float output);
stalta(StaLtaInt staltav, ArrayInt input, ArrayInt output);
stalta(StaLtaFloat staltav, ArrayFloat input,
                                        ArrayFloat output);
```

Parameters :

1. `staltav` is the variable containing the STA/LTA parameters (section B.3.11).
2. `input` is the input data of the STA/LTA algorithm.
3. `output` is the result of the STA/LTA algorithm.

Example :

```
StaLtaInt sti (400, 4000, 400, 1000);
ArrayInt in(100);
ArrayInt res(100);

/* Compute STLA/LTA from data of the in array */
stalta(sti, in, res);
```

B.5.26 stdDev

Standard deviation.

```
stdDev(ArrayInt input, Int result);
stdDev(ArrayFloat input, Float result);
```

Parameters :

1. `input` is the array to process.
2. `result` is the standard deviation computed from the elements of the array.

Example :

```
ArrayInt in(100);
Int res;

/* Compute standard deviation from the in array */
stdDev(in, res);
```

B.5.27 sub

Subtract elements of an array (with a value) or subtract the elements of two arrays.

```
sub(ArrayInt input1, ArrayInt input2, ArrayInt output);
sub(ArrayFloat input1, ArrayFloat input2, ArrayFloat output);
```

Parameters :

1. `input1` is the first operand of the subtraction.
2. `input2` is the second operand of the subtraction.
3. `output` is the result of the subtraction.

Example :

```
ArrayInt in1(100);
ArrayInt in2(100);
ArrayInt res(100);

/* Subtract each element of in1 by 128 */
sub(in1, 128, res);

/* Subtract in2 by in1 element by element */
sub(in1, in2, res);
```

B.5.28 sum

Sum all the elements of an array.

```
sum(ArrayInt array, Int result);
sum(ArrayFloat array, Float result);
```

Parameters :

1. `array` is the array to process.
2. `result` is the sum of all elements of the array.

Example :

```
ArrayInt in(100);
Int res;

/* Sum all of the elements of the in array */
sum(in, res);
```

B.5.29 trigger

Return true for a high or low value, or a rising or falling edge, compared to a threshold. The parameters of the trigger must be defined when declaring the variable `triggerv`. The input can be array, in this case the output is set to true if at least one value in the array has produce a trigger.

```
trigger(TriggerInt triggerv, Int input, Bool output);
trigger(TriggerFloat triggerv, Float input, Bool output);
trigger(TriggerInt triggerv, ArrayInt input, Bool output);
trigger(TriggerFloat triggerv, ArrayFloat input, Bool output);
```

Parameters :

1. `triggerv` is the variable containing the trigger parameters (section B.3.12).
2. `input` is the input data of the trigger algorithm.
3. `output` is the result of the trigger algorithm.

Example :

```
TriggerInt triggerv(RISING EDGE, 2500, 4000, 10000);
ArrayInt in(100);
Bool res;

/* Put true in the res variable if at least one element
   of the in array raised above the threshold defined
   in the triggerv variable */
trigger(triggerv, in, res);
```

B.5.30 var

Variance.

```
var(ArrayInt array, Int result);
var(ArrayFloat array, Float result);
```

Parameters :

1. `array` is the array to process.
2. `result` is the variance of the elements of the array.

Example :

```
ArrayInt in(100);
Int res;

/* Compute the variance of the in array */
var(in, res);
```

B.6 Utility functions

B.6.1 ascentRequest

Request the ascent of the float. When this function is called, and if the float is at the park step, all the acquisition modes are stopped and the float enter in the ascent step and execute the acquisition modes defined in the mission configuration for the ascent.

```
ascentRequest();
```

No parameters.

Example :

```
ascentRequest();
```

B.6.2 convert

Copy data of a specific type of data to an other type of data.

```
convert(ArrayInt input, ArrayFloat output);
convert(ArrayInt input, ArrayComplexInt output);
convert(ArrayFloat input, ArrayInt output);
convert(ArrayFloat input, ArrayComplexFloat output);
convert(BufferInt input, ArrayInt output);
convert(BufferFloat input, ArrayFloat output);
```

Parameters :

1. `input` is the variable to convert.
2. `output` is the converted variable.

Example :

```
ArrayInt intNumber(100);
ArrayComplexInt complexNb(100);
ArrayFloat floatNumber(100);

/* Convert the integer array into a complex integer array.
   The array to convert is copied into the real part of
   the complex array */
convert(intNumber, complexNb);

/* Convert the integer array into a floating point
   number array */
convert(intNumber, floatNumber);
```

B.6.3 getSampleIndex

This function is only valuable for simulation. It get the index of sample currently read in input data file used for the simulation.

```
getSampleIndex(Int sampeIndex)
```

Parameter :

1. `sampeIndex` is the index of the sample.

Example :

```
Int sampleIndex;
getSampleIndex(sampleIndex);
```

B.6.4 getTimestamp

Get the current date with a resolution of 1 second.

```
getTimestamp(Int timestamp)
```

Parameter :

1. `timestamp` is the current date.

Example :

```
Int timestamp;
getTimestamp(timestamp);
```

B.6.5 record

Record a value, an array, a buffer or a string in a file.

```
record(File file, Int data)
record(File file, Float data)
record(File file, ArrayInt data)
record(File file, ArrayFloat data)
record(File file, BufferInt data)
record(File file, BufferFloat data)
record(File file, String data)
```

Parameter :

1. `file` is the file in which to record the data.
2. `data` is the data to record.

Example :

```
File f;
Int timestamp;
ArrayFloat array;

/* Record the current date and the data in array */
record(f, timestamp);
record(f, array);
```

Annexe C

MeLa tutorial

Contents

C.1 Continuous recording 122
C.2 Detection algorithm 124
C.3 Short acquisition 127
C.4 Composition 127

C.1 Continuous recording

The first application continuously records data from the hydrophone. Create a file called `TutoApp.mela` in the `MeLaApps` directory. In the `Compomaid.java` file, write the file name in `String app1_name = "TutoApp";` and leave the second app name blank `String app1_string = "";` (lines 70, 71).

In the `TutoApp.mela` file, define the park duration and the depth of the mission, for example :

```
Mission:
    ParkTime: 14400 minutes;
    ParkDepth: 1000 meters;
```

Define the coordinator with an acquisition mode called `ContinuousRecord` executed only during the park stage :

```
Coordinator:
    ParkAcqModes: ContinuousRecord;
```

Define the acquisition mode, the sensor to use, its sampling frequency and the variable that will receive the data from the sensor. The acquisition mode must be a `ContinuousAcqMode` since we want to record all data from the sensor. We use the low frequency hydrophone `HydrophoneBF` with a sampling frequency of 200 Hz and the `x` variable, an array that can contain only one sample.

```
ContinuousAcqMode ContinuousRecord:

Input:
     sensor: HydrophoneBF(200);
     data: x(1);
```

Next, define the file in which to record the data from the hydrophone :

```
Variables:
     File f;
```

A continuous acquisition mode must contain a `RealTimeSequence` that is able to handle all the data from the sensor. In this sequence we call the record function to record the hydrophone data in the file.

```
RealTimeSequence detection:
     record(f, x);
endseq;
```

Finally close the acquisition mode with :

```
endmode;
```

The complete code of the application is :

```
Mission:
    ParkTime: 14400 minutes;
    ParkDepth: 1000 meters;
```

```
Coordinator:
  ParkAcqModes: ContinuousRecord;

ContinuousAcqMode ContinuousRecord:

Input:
    sensor: HydrophoneBF(200);
    data: x(1);

Variables:
    File f;

RealTimeSequence detection:
    record(f, x);
endseq;

endmode;
```

Launch the MeLa compiler, it will return the following analysis results :

```
* Verification of execution time:

Maximum processor usage during PARK:
  Error: 10000 % > 100 %


* Estimation of energy consumption:

Autonomy: 0 years
Energy consumption during DESCENT: 295,3mWh
  Processor consumption: 95,3mWh
  Actuator consumption: 200mWh

Energy consumption during PARK: 16872mWh
  Processor consumption: 2472mWh
  HydrophoneBF: 14400mWh
  Actuator consumption: 0mWh

Energy consumption during ASCENT: 3035,7mWh
  Processor consumption: 35,7mWh
  Actuator consumption: 3000mWh

Energy consumption during SURFACE: 4837510,3mWh
  Processor consumption: 10,3mWh
  Actuator consumption: 0mWh
  Transmission consumption: 4837500mWh


* Amount of data transmitted by satellite:
```

```
Transmission per cycle: 675000 kB
Transmission per month: 1915522,56 kB
```

The first thing is that the processor usage is far too high. A way to correct the problem would be to record the data in a `ProcessingSequence` instead of the `RealTimeSequence` but this would not gives us the guarantee that all the data are recorded conducting to a corrupted signal.

A better solution is to increase the size of the `x` variable, this would reduce the period of execution of the real time sequence, giving more time to the recording function to be executed. Indeed the recording function has an execution time almost independent of the amount of data to record.

Define the size of the `x` variable to 10000. The processor usage is now only 1%. However the power consumption is still high. We can see that most of the power is consumed at surface because all the recorded data are transmitted by satellite. Instead of recording all the data it is possible to record only some detected signal. Thus, our next objective is to write a simple detection algorithm.

C.2 Detection algorithm

The principle of this detection algorithm is to detect an increase of amplitude in the acoustic signal and to record 100 seconds before and 200 seconds after this increase.

The sampling frequency of the hydrophone is set to 20 Hz for this application, the lowest is the sampling frequency, the lowest will be the amount of recorded data. We will process the data by packets of 1 seconds, which means that we will be able to detect the beginning of a seismic wave every second. Choosing a smaller packet size could increase the processor usage (but not necessary) and big blocks of data would not allow us to identify the beginning of a seismic wave. The input part of the acquisition mode must be now :

```
Input:
    sensor: HydrophoneBF(20);
    data: x(20);
```

In order to record 100 seconds before and 200 seconds after the amplitude increase, we need a buffer that will keep 300 seconds of data in memory. For a sampling frequency of 20 Hz, the buffer has to be able to contain 6000 samples. Each data packet must be appended to the buffer with the push function. The content of the acquisition mode must be now :

```
Input:
    sensor: HydrophoneBF(20);
    data: x(20);

Variables:
    /* Buffer containing 300 seconds of data,
       or 6000 samples at 20 Hz */
```

```
    BufferInt last5Minutes(6000);

RealTimeSequence detection:
    push(last5Minutes, x);
endseq;
```

Now we use an STA/LTA algorithm to detect an increase of amplitude in the signal. The result of the STA/LTA gives a value close to 1 when the signal is stable but this value increases when there is an elevation of the acoustic signal that reaches the Short Term Average. Considering that the duration of an earthquake is of several hundred of seconds, we want to compute the short term average over 10 seconds of data, which is 200 samples. For the long term average we choose a length of 100 seconds that does not include the 10 seconds of the short term average. These parameters will be refined later. Since the average is computed with integers we have to define a scaling factor, we choose it to be 10 which is sufficient to see changes of the ratio at a resolution of 0.1. The content of the acquisition mode must be now :

```
Variables:
    File f;
    /* Buffer containing 300 seconds of data,
       or 6000 samples at 20 Hz */
    BufferInt last5Minutes(6000);

    /* STALTA variables */
    StaLtaInt staltaInstance(200, 2000, 200, 1000);
    ArrayInt staltaResult(200);

RealTimeSequence detection:
    push(last5Minutes, x);
    stalta(staltaInstance, x, staltaResult);
endseq;
```

The result of the STA/LTA is used to trigger the recording of the signal, however it is not sufficient to compare the STA/LTA result with a threshold value since it would record the signal several times while the threshold is exceeded. The signal must trigger the recording only one time. Thus we define a `TriggerInt` variable called `triggerInstance` with the `RISING_EDGE` trigger mode. We want to activate the trigger on a rising edge with a threshold of `2000` (equivalent to 2 when we consider the scaling factor of 1000 of the STA/LTA). Since we want to wait 150 seconds after the trigger to get the whole seismic signal, we define the trigger delay to 3000 samples. We also define a minimum time between each trigger such that two records must be spaced by at least 60 seconds (1200 samples).

```
/* Trigger variable */
TriggerInt triggerInstance(RISING_EDGE, 2000, 3000, 1200);
```

The `trigger` function must be put in the real time sequence after the `stalta` function. Its parameters must include the `triggerInstance` variable, the `staltaResult` variable and a boolean variable that will be set to true each time the trigger is activate. The

recording function is put in a condition that test the boolean variable. The condition must be annotated with a probability, in this case `1 per day`, that is used by the application to compute the amount of data that will be transmitted trough satellite. The recording function can not be put in the real time sequence because its execution time is too long. Instead, we put it in a processing sequence for which functions with long execution time are allowed (at the cost of potentially missing some samples). The final application is described below :

```
Mission:
   ParkTime: 14400 minutes;
   ParkDepth: 1000 meters;


Coordinator:
   ParkAcqModes: DetectionRecord;


ContinuousAcqMode DetectionRecord:

Input:
    sensor: HydrophoneBF(20);
    data: x(20);

Variables:
    File f;
    /* Buffer containing 300 seconds of data,
       or 6000 samples at 20 Hz */
    BufferInt last5Minutes(6000);

    /* STALTA variables */
    StaLtaInt staltaInstance(200, 2000, 200, 1000);
    ArrayInt staltaResult(200);

    /* Trigger variable */
    TriggerInt triggerInstance(RISING_EDGE, 2000, 3000, 1200);
    Bool trigRes;

RealTimeSequence detection:
    push(last5Minutes, x);
    stalta(staltaInstance, x, staltaResult);
    trigger(triggerInstance, staltaResult, trigRes);
    if trigRes:
        @probability = 1 per day
        call recordSeq
    endif;
endseq;

ProcessingSequence recordSeq:
    record(f, last5Minutes);
```

```
endseq;
endmode;
```

We do not show the simulation part allowing to set the parameters of the application.

C.3 Short acquisition

Instead of using a detection algorithm, one may want to record only some small snippets of the signal. For this we can use a short acquisition mode. The time interval at which the acquisition mode is executed must be set in the coordinator. Each time we record 10 minutes of data (200 samples).

```
Mission:
   ParkTime: 14400 minutes;
   ParkDepth: 1000 meters;

Coordinator:
   DescentAcqModes: ShortRecord every 60 minutes;
   ParkAcqModes: ShortRecord every 1 hour;
   AscentAcqModes: ShortRecord every 60 minutes;



ShortAcqMode ShortRecord:

Input:
    sensor: HydrophoneBF(200);
    data: x(600);

Variables:
    File f;

ProcessingSequence recordSeq:
    record(f, x);
endseq;

endmode;
```

C.4 Composition

In order to execute both applications on the same instrument we can use the composition tool that will take two MeLa files or just copy both applications in the same file and edit the coordinator part to start the acquisition modes at the appropriate times. The corresponding MeLa application will be :

```
Mission:
   ParkTime: 14400 minutes;
```

```
    ParkDepth: 1000 meters;

Coordinator:
    DescentAcqModes: ShortRecord every 60 minutes;
    ParkAcqModes: ContinuousRecord
                  ShortRecord every 1 hour;
    AscentAcqModes: ShortRecord every 60 minutes;

ShortAcqMode ShortRecord:
      ...
endmode;

ContinuousAcqMode ContinuousRecord:
      ...
endmode;
```