

N°d'ordre NNT : 2018LYSEI018

THESE de DOCTORAT DE L'UNIVERSITE DE LYON
opérée au sein de
L'INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE LYON

Ecole Doctorale 512
INFORMATIQUE ET MATHÉMATIQUES

Spécialité/ discipline de doctorat :
Informatique

Soutenue publiquement le 15/03/2018, par :
ȘTEFAN-GABRIEL CHITIC

Middleware and programming models for multi-robot systems

Devant le jury composé de :

Abderrafiaa KOUKAM	Professeur	Université de Technologie de Belfort-Montbéliard	Rapporteur
Philippe LALANDA	Professeur	Université Joseph Fourier, Saint-Martin-d'Hères	Rapporteur
Noury BOURAQADI	Professeur	IMT Lille Douai	Président de jury
Stéphanie CHOLLET	Maître de conférences	Grenoble INP - Esisar, Valence	Examinatrice
Olivier SIMONIN	Professeur	INSA Lyon	Directeur de thèse
Julien PONGE	Maître de conférences	INSA Lyon	Co-directeur de thèse

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Département FEDORA – INSA Lyon - Ecoles Doctorales – Quinquennal 2016-2020

SIGLE	ÉCOLE DOCTORALE	NOM ET COORDONNEES DU RESPONSABLE
CHIMIE	**CHIMIE DE LYON** **http://www.edchimie-lyon.fr** Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage **secretariat@edchimie-lyon.fr** INSA : R. GOURDON	M. Stéphane DANIELE Institut de recherches sur la catalyse et l'environnement de Lyon IRCELYON-UMR 5256 Équipe CDFA 2 Avenue Albert EINSTEIN 69 626 Villeurbanne CEDEX **directeur@edchimie-lyon.fr**
E.E.A.	**ÉLECTRONIQUE, ÉLECTROTECHNIQUE, AUTOMATIQUE** **http://edeea.ec-lyon.fr** Sec. : M.C. HAVGOUDOUKIAN **ecole-doctorale.eea@ec-lyon.fr**	M. Gérard SCORLETTI École Centrale de Lyon 36 Avenue Guy DE COLLONGUE 69 134 Écully Tél : 04.72.18.60.97 Fax 04.78.43.37.17 **gerard.scorletti@ec-lyon.fr**
E2M2	**ÉVOLUTION, ÉCOSYSTÈME, MICROBIOLOGIE, MODÉLISATION** **http://e2m2.universite-lyon.fr** Sec. : Sylvie ROBERJOT Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 INSA : H. CHARLES **secretariat.e2m2@univ-lyon1.fr**	M. Fabrice CORDEY CNRS UMR 5276 Lab. de géologie de Lyon Université Claude Bernard Lyon 1 Bât. Géode 2 Rue Raphaël DUBOIS 69 622 Villeurbanne CEDEX Tél : 06.07.53.89.13 **cordey@univ-lyon1.fr**
EDISS	**INTERDISCIPLINAIRE SCIENCES-SANTÉ** **http://www.ediss-lyon.fr** Sec. : Sylvie ROBERJOT Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 INSA : M. LAGARDE **secretariat.ediss@univ-lyon1.fr**	Mme Emmanuelle CANET-SOULAS INSERM U1060, CarMeN lab, Univ. Lyon 1 Bâtiment IMBL 11 Avenue Jean CAPELLE INSA de Lyon 69 621 Villeurbanne Tél : 04.72.68.49.09 Fax : 04.72.68.49.16 **emmanuelle.canet@univ-lyon1.fr**
INFOMATHS	**INFORMATIQUE ET MATHÉMATIQUES** **http://edinfomaths.universite-lyon.fr** Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage Tél : 04.72.43.80.46 Fax : 04.72.43.16.87 **infomaths@univ-lyon1.fr**	M. Luca ZAMBONI Bât. Braconnier 43 Boulevard du 11 novembre 1918 69 622 Villeurbanne CEDEX Tél : 04.26.23.45.52 **zamboni@maths.univ-lyon1.fr**
Matériaux	**MATÉRIAUX DE LYON** **http://ed34.universite-lyon.fr** Sec. : Marion COMBE Tél : 04.72.43.71.70 Fax : 04.72.43.87.12 Bât. Direction **ed.materiaux@insa-lyon.fr**	M. Jean-Yves BUFFIÈRE INSA de Lyon MATEIS - Bât. Saint-Exupéry 7 Avenue Jean CAPELLE 69 621 Villeurbanne CEDEX Tél : 04.72.43.71.70 Fax : 04.72.43.85.28 **jean-yves.buffiere@insa-lyon.fr**
MEGA	**MÉCANIQUE, ÉNERGÉTIQUE, GÉNIE CIVIL, ACOUSTIQUE** **http://edmega.universite-lyon.fr** Sec. : Marion COMBE Tél : 04.72.43.71.70 Fax : 04.72.43.87.12 Bât. Direction **mega@insa-lyon.fr**	M. Philippe BOISSE INSA de Lyon Laboratoire LAMCOS Bâtiment Jacquard 25 bis Avenue Jean CAPELLE 69 621 Villeurbanne CEDEX Tél : 04.72.43.71.70 Fax : 04.72.43.72.37 **philippe.boisse@insa-lyon.fr**
ScSo	**ScSo*** **http://ed483.univ-lyon2.fr** Sec. : Viviane POLSINELLI Brigitte DUBOIS INSA : J.Y. TOUSSAINT Tél : 04.78.69.72.76 **viviane.polsinelli@univ-lyon2.fr**	M. Christian MONTES Université Lyon 2 86 Rue Pasteur 69 365 Lyon CEDEX 07 **christian.montes@univ-lyon2.fr**

*ScSo : Histoire, Géographie, Aménagement, Urbanisme, Archéologie, Science politique, Sociologie, Anthropologie

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

To Anda, my faithfully, lovely wife.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Acknowledgements

The journey behind this PhD thesis started in 2013 when I was a technical student at CERN. After discovering the previous year, the working environment in one of the largest software companies, my stay at CERN connected me with the research and academia world. I was curious to discover this new world that was, step by step, revealing to me during my internship. As a result, I have decided to continue my master's degree with a PhD. First, I have contacted my co-supervisor, Dr. Julien Ponge with whom I have already worked on various projects during my bachelors and master's degree. He then mange to find my thesis supervisor, Prof. Olivier Simonin and together to plan and start this magnificent journey, my PhD thesis.

I would like first to thank my supervisor Prof. Olivier Simonin for his good guidance and support in a robotic research community. Having just a software engineer background, his advices helped me understand and apply robotic related concepts. It has been a great pleasure to work with him, both from a professional and a personal point of view. I would also like to thank deeply my co-supervisor Dr. Julien Ponge for all the time we worked together, for his support, technical debates and guidance. It was a pleasure and an honor to have him as a colleague and as a friend. Special thanks for the period when he supported me and give me the power to re-find my motivation to finish this PhD.

Secondly, I would like to thank to Prof. Fabrice Valois for his precious administrative and moral support. It was a pleasure to have him as teacher in my master's degree, as a laboratory head, and as a colleague.

Many thanks to all the colleague and CITI stuff for their guidance and support. Thank you! It was a pleasure to work with you.

Many thanks go to my reviewers and examiners of my PhD defense: Philippe Lalanda, Abderrafiaa Koukam, Noury Bouraqadi and Stéphanie Chollet. It has been a great honor to have you all as part of the examination committee!

The biggest and most profound thanks go to my wife, Anda-Catalina Chelba-Chitic for her deeply love, hard working in motivating me finish this PhD and profound support. I know that during these years of PhD, you have made sacrifices and I am thankful from all my heart that you not only accepted them but also endorsed them with all your love. Thanks, my love!

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

And finally let me switch to Romanian, my native language in order to thank to my parents.

In final, mulţumirle cele mai călduroase sunt pentru părinţii mei, Gheorghe-Constantin Chitic şi Maria Chitic pentru simplu fapt că au fost alături de mine la bine şi la rau. Probabil nu aş fi ajuns aşa de departe in studiile şi cariera mea fără încurajările şi educatia care mi le-au oferit. Această lucrare le este dedicată şi lor. Vă mulţumesc!

Saint Genis Pouilly, 21 December 2017, on my 29th *anniversary* S. G. C.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Abstract

Despite many years of work in robotics, there is still a lack of established software architecture and middleware for multi-robot systems. A robotic middleware should be designed to abstract the low-level hardware architecture, facilitate communication and integration of new software. This PhD thesis is focusing on middleware for multi-robot system and how we can improve existing frameworks for fleet purposes by adding multi-robot coordination services, development and massive deployment tools. We expect robots to be increasingly useful as they can take advantage of data pushed from other external devices in their decision making instead of just reacting to their local environment (sensors, cooperating robots in a fleet, etc.).

This thesis first evaluates one of the most recent middleware for mobile robot(s), Robot operating system (ROS) and continues with a state of the art about the commonly used middleware in robotics. Based on the conclusions, we propose an original contribution in the multi-robot context, called SDfR (Service discovery for Robots), a service discovery mechanism for Robots. The main goal is to propose a mechanism that allows highly mobile robots to keep track of the reachable peers inside a fleet while using an ad-hoc infrastructure. Another objective is to propose a network configuration negotiation protocol. Due to the mobility of robots, classical peer to peer network configuration techniques are not suitable. SDfR is a highly dynamic, adaptive and scalable protocol adapted from Simple Service Discovery Protocol (SSDP). We conducted a set of experiments, using a fleet of Turtlebot robots, to measure and show that the overhead of SDfR is limited.

The last part of the thesis focuses on programming model based on timed automata. This type of programming has the benefits of having a model that can be verified and simulated before deploying the application on real robots. In order to enrich and facilitate the development of robotic applications, a new programming model based on timed automata state machines is proposed, called ROSMDB (Robot Operating system Model Driven Behavior). It provides model checking at development phase and at runtime. This contribution is composed of several components: a graphical interface to create models based on timed automata, an integrated model checker based on UPPAAL and a code skeleton generator. Moreover, a ROS specific framework is proposed to verify the correctness of the execution of the models and to trigger alerts. Finally, we conduct two experiments: one with a fleet of Parrot drones and second with Turtlebots in order to illustrates the proposed model and its ability to check properties.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Acknowledgements

Key words:
Dynamic middleware, Connected robots, reconfiguration, resilience, mobility, programming languages, multi-robot systems, robotic cloud, Robotic fleet, Service Discovery, Service oriented Architecture.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Résumé

Malgré de nombreuses années de travail en robotique, il existe toujours un manque d'architecture logicielle et de middleware stables pour les systèmes multi-robot. Un intergiciel robotique devrait être conçu pour faire abstraction de l'architecture matérielle de bas niveau, faciliter la communication et l'intégration de nouveaux logiciels. Cette thèse se concentre sur le middleware pour systèmes multi-robot et sur la façon dont nous pouvons améliorer les frameworks existantes dans un contexte multi-robot en ajoutant des services de coordination multi-robot, des outils de développement et de déploiement massif. Nous nous attendons à ce que les robots soient de plus en plus utiles car ils peuvent tirer profit des données provenant d'autres périphériques externes dans leur prise de décision au lieu de simplement réagir à leur environnement local (capteurs, robots coopérant dans une flotte, etc.).

Cette thèse évalue d'abord l'un des intergiciels les plus récents pour robot(s) mobile(s), Robot operating system (ROS), suivi par la suite d'un état de l'art sur les middlewares couramment utilisés en robotique. Basé sur les conclusions, nous proposons une contribution originale dans le contexte multi-robots, appelé SDfR (Service discovery for Robots), un mécanisme de découverte des services pour les robots. L'objectif principal est de proposer un mécanisme permettant aux robots de garder une trace des pairs accessibles à l'intérieur d'une flotte tout en utilisant une infrastructure ad-hoc. Un autre objectif est de proposer un protocole de négociation de configuration réseau. A cause de la mobilité des robots, les techniques classiques de configuration de réseau pair à pair ne conviennent pas. SDfR est un protocole hautement dynamique, adaptatif et évolutif adapté du protocole SSDP (Simple Service Discovery Protocol). Nous conduisons un ensemble d'expériences, en utilisant une flotte de robots Turtlebot, pour mesurer et montrer que le sur débit de SDfR est limité.

La dernière partie de la thèse se concentre sur un modèle de programmation basé sur un automate temporisé. Ce type de programmation a l'avantage d'avoir un modèle qui peut être vérifié et simulé avant de déployer l'application sur de vrais robots. Afin d'enrichir et de faciliter le développement d'applications robotiques, un nouveau modèle de programmation basé sur des automates à états temporisés est proposé, appelé ROSMDB (Robot Operating system Model Driven Behaviour). Il fournit une vérification de modèle lors de la phase de développement et lors de l'exécution. Cette contribution est composée de plusieurs composants : une interface graphique pour créer des modèles basés sur un automate temporisé, un vérificateur de modèle intégré basé sur UPPAAL et un générateur de squelette de code. De plus, un framework spécifique à ROS est proposé pour vérifier l'exactitude de l'exécution du

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

modèle et déclencher des alertes. Enfin, nous avons effectué deux expériences : une avec une flotte de drones Parrot et l'autre avec des Turtlebots afin d'illustre le modèle proposé et sa capacité à vérifier les propriétés.

`Mots clefs :`
Middleware dynamique, robots connectés, reconfiguration, résilience, mobilité, langages de programmation, systèmes multi-robots, cloud robotisé, flotte robotisée, service deécouverte, architecture orientée service.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Contents

Acknowledgements i

Abstract (English/Français) iii

List of figures xi

List of tables xv

I Introduction and background 1

1 Introduction 3
- 1.1 Context . . . 3
- 1.2 Key research issues . . . 3
- 1.3 Contributions Overview . . . 4
- 1.4 Outline . . . 5

2 Middleware for robotics 7
- 2.1 Introduction . . . 7
- 2.2 Middlewares in distributed systems . . . 8
  - 2.2.1 Classes of middlewares . . . 9
  - 2.2.2 Middlewares in robotics systems . . . 15
- 2.3 Challenges for middleware in robotics . . . 16
  - 2.3.1 Why a middleware for robotics ? . . . 17
  - 2.3.2 Infrastructure and communication . . . 17
  - 2.3.3 Application programming interfaces and services . . . 18
  - 2.3.4 Robotic applications as services . . . 19
  - 2.3.5 Limitations . . . 20
- 2.4 Existing middlewares . . . 20
- 2.5 Comparative criteria . . . 23
- 2.6 Middleware Comparison . . . 25
  - 2.6.1 Architecture . . . 25
  - 2.6.2 Infrastructure . . . 26
  - 2.6.3 Usage . . . 28
- 2.7 Conclusion . . . 30

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

3 Formalisms to design systems behavior **33**
3.1 Introduction . . . 33
3.2 Model driven development . . . 34
3.2.1 Model driven development characteristics . . . 35
3.2.2 Model driven development paradigm . . . 36
3.2.3 Model driven development in robotics . . . 37
3.2.4 Conclusion . . . 37
3.3 Classical formalism . . . 38
3.3.1 Finite state machine . . . 38
3.3.2 Petri nets . . . 38
3.3.3 Markov decision process . . . 39
3.3.4 Process algebras . . . 40
3.4 Timed automata . . . 40
3.4.1 Overview . . . 41
3.4.2 Classes of timed automata . . . 45
3.4.3 Software tools . . . 47
3.5 Conclusion . . . 49

II Model driven multi-robot applications development **51**

4 Service discovery for robots **53**
4.1 Objectives and motivation for fleet service discovery . . . 53
4.2 Limitation of existing service discovery protocols . . . 54
4.3 Definition of SDfR protocol . . . 56
4.3.1 SDfR as a derivate of SSDP . . . 57
4.3.2 Protocol model and description . . . 60
4.3.3 Implementation . . . 64
4.4 Evaluation of SDfR overhead with robots . . . 68
4.4.1 Experimental settings . . . 68
4.4.2 Functional validation . . . 70
4.5 Summary . . . 81

5 ROSMDB: Development methodology **83**
5.1 From component services to fleet applications . . . 84
5.1.1 Service oriented architecture as root for model based robotic software development . . . 84
5.1.2 Alternative approach for service oriented architecture . . . 87
5.2 Modeling component external interactions with timed automata . . . 88
5.2.1 Motivating example . . . 89
5.2.2 Timed automata . . . 91
5.2.3 Event recording timed automata . . . 93

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

5.3 Validating service compositions . . . 95
5.3.1 Applications, services and components . . . 96
5.3.2 Event recording automata composition . . . 99
5.3.3 Model validation . . . 103
5.4 The ROSMDB toolset . . . 105
5.4.1 Global overview of the environment . . . 106
5.4.2 Design . . . 107
5.4.3 Validation . . . 110
5.4.4 Code generation . . . 113
5.4.5 ROSMDB framework . . . 115
5.4.6 Summary of SDfR usage . . . 117
5.4.7 Fleet deployment . . . 118
5.4.8 Runtime validation feedback . . . 120
5.5 Summary . . . 123

6 ROSMDB: Experimentations **125**
6.1 Package delivery by drones swarm . . . 125
6.1.1 Flight synchronization based on N pole: Description . . . 126
6.1.2 Flight synchronization based on N pole: Models . . . 127
6.1.3 Flight synchronization based on N pole: Experimental Results . . . 130
6.2 Guest welcoming and management with intrusion detection system . . . 132
6.2.1 Random movement object search: Description . . . 136
6.2.2 Random movement object search: Model . . . 136
6.2.3 Random movement object search: Experimental results . . . 139
6.3 Summary . . . 142

7 Conclusion and perspectives **143**
7.1 Concluding remarks . . . 143
7.2 Perspectives beyond *Robot operating system Model Driven Behavior (ROSMDB)* and *Service Discovery for Robots (SDfR)* . . . 144
7.2.1 Short-term perspectives . . . 145
7.2.2 Long-term perspectives . . . 146

A Selected Middlewares descriptions **149**
A.1 Player/Stage . . . 149
A.2 Robot operating system . . . 150
A.3 Miro . . . 151
A.4 MRDS . . . 151
A.5 MARIE . . . 152
A.6 Orca . . . 152
A.7 Carmen . . . 152
A.8 Pyro . . . 153

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

B Model driven development in robotics **155**
B.1 RobotML . . . 155
B.2 *3-View Component Meta-Model for Model-Driven Robotic Software Development (V3CMM)* . . . 155
B.3 SmartSoft . . . 156
B.4 BRICS model . . . 156

C Product construction of examples **157**
C.1 Obstacle detection and avoidance navigation service . . . 157
C.1.1 Optical sensor component . . . 157
C.1.2 Navigation component . . . 158
C.1.3 Product construction of the service . . . 159
C.2 Fleet platooning service . . . 160
C.2.1 Leader component . . . 160
C.2.2 Networking component . . . 161
C.2.3 Platooning manager component . . . 161
C.2.4 Product construction of the service . . . 162
C.3 Fleet platooning robot with collision avoidance application . . . 164

D Flight synchronization based on N pole **165**
D.1 Commander application . . . 165
D.1.1 Take off manager service . . . 165
D.1.2 Lock North manager service . . . 168
D.1.3 Drone Movement Manager service . . . 170
D.1.4 Networking service . . . 172
D.2 Controller application . . . 174
D.2.1 Command sender service . . . 174

E Random movement object search **177**
E.1 Collision avoidance application . . . 177
E.1.1 Engine stopper service . . . 177
E.1.2 Avoidance service . . . 180
E.2 Object detection application . . . 182
E.2.1 Image analyzer service . . . 182
E.2.2 Mover service . . . 184

Bibliography **205**

Glossary **207**

Acronyms **209**

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

List of Figures

2.1 Middleware layer interaction 8
2.2 RPC progress 9
2.3 DOM architecture 10
2.4 MOM progress 11
2.5 Three phase commit protocol 12
2.6 SOA meta-model 13
2.7 Serial Bus communication architecture 15
2.8 Ad-hoc fleet infrastructure 19

3.1 Object Management Group modelling 37
3.2 Simple light controller 41
3.3 Example of a simple timed automata 41
3.4 An LTS example 44
3.5 Example of an event timed automata 46
3.6 Example of a robust automata 46
3.7 Example of a silent transition 47
3.8 Model checking principle 47
3.9 UPPALL screen-shot 49

4.1 SSDP and SDfR protocol timed diagram. 57
4.2 SSDP and SDfR differences 59
4.3 SDfR protocol timed diagram. 61
4.4 SDfR common header. 62
4.5 Service oriented architecture in SDfR. 64
4.6 SDfR service architecture. 65
4.7 Turtlebots in the experimentation hall. 70
4.8 Average request time for publishing a service. 71
4.9 Average request time for unpublishing a service. 72
4.10 Average request time for subscribing a consumer. 73
4.11 CPU usage 74
4.12 % of cpu usage for 6 robots with 70% publishers. 75
4.13 Memory usage 76
4.14 % of memory usage for 6 robots with 70% publishers. 77

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

4.15 TX usage . . . 78
4.16 Average number of kilobytes transmitted per robot in 5 minutes with 70% publishers using 6 robots. . . . 79
4.17 RX usage . . . 80
4.18 Average number of kilobytes received per robot in 5 minutes with 70% publishers using 6 robots. . . . 81

5.1 Proposed robotic application life cycle . . . 84
5.2 Computing speed developments . . . 86
5.3 Obstacle detection and avoidance . . . 89
5.4 Fleet platooning . . . 90
5.5 Simple collision avoidance modelling . . . 91
5.6 Simple collision avoidance timed automaton . . . 92
5.7 Simple collision avoidance Event Recording automata . . . 94
5.8 Event Recording Automaton operations . . . 95
5.9 Bottom-up approach of Event Recording automata composition . . . 96
5.10 Components of service: Obstacle detection and avoidance navigation . . . 97
5.11 Components of service: Fleet platooning . . . 98
5.12 Timed automata product construction example . . . 100
5.13 Obstacle detection and avoidance navigation as product . . . 101
5.14 Obstacle detection and avoidance navigation as reduce product . . . 102
5.15 Product of entire example application . . . 103
5.16 General architecture for ROSMDB . . . 106
5.17 File manager in ROSMDB . . . 107
5.18 Graphical user interface in ROSMDB . . . 109
5.19 UPPAAL - ROSMDB integration . . . 111
5.20 ROSMDB model-validation front-end . . . 112
5.21 User code interaction with ROSMDB . . . 115
5.22 ROSMDB framework . . . 116
5.23 SDfR metadata input . . . 118
5.24 Deployment interface in ROSMDB tool chain . . . 119
5.25 ROSMDB fleet deployment process . . . 120
5.26 ROSMDB trace feedback . . . 121
5.27 ROSMDB trace feedback . . . 122

6.1 Fleet of drones network . . . 127
6.2 Command sender Model . . . 128
6.3 Networking Model . . . 128
6.4 Take off Model . . . 128
6.5 Lock North Model . . . 129
6.6 Drone Movement Model . . . 129
6.7 Experimenting Flight synchronization with Parrot Drones . . . 130
6.8 Lock North messages inter-arrival time . . . 131

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

6.9 Drone movement messages inter-arrival time . . . 132
6.10 Refined Lock North Model executed on each drone assigned laptop . . . 133
6.11 Refined Drone Movement Model executed on each drone assigned laptop . . . 133
6.12 Guest handeling . . . 134
6.13 Intrusion detection . . . 135
6.14 Engine stopper Model . . . 137
6.15 Avoidance Model . . . 137
6.16 Image analyser Model . . . 138
6.17 Movement Model . . . 138
6.18 Experimenting Random movement object search with Turtlebots . . . 139
6.19 Engine stopper messages inter-arrival time . . . 139
6.20 Avoidance service messages inter-arrival time . . . 140
6.21 Image analyzer messages inter-arrival time . . . 141
6.22 Mover messages inter-arrival time . . . 141

C.1 Optical sensor component Event Recording Automaton . . . 157
C.2 Navigation component Event Recording Automaton . . . 158
C.3 Product construction of components for service: Obstacle detection and avoidance navigation . . . 159
C.4 Reduced product construction of components for service: Obstacle detection and avoidance navigation . . . 160
C.5 Leader component Event Recording Automaton . . . 160
C.6 Networking component Event Recording Automaton . . . 161
C.7 Platooning manager component Event Recording Automaton . . . 161
C.8 Partial Product construction of components for service: Fleet platooning . . . 162
C.9 Product construction of components for service: Fleet platooning . . . 163
C.10 Reduced product construction of components for service: Fleet platooning . . . 163
C.11 Product construction for entire robotic application . . . 164
C.12 Reduced Product construction for entire robotic application . . . 164

D.1 Take off manager: *ROSMDB* model screenshot . . . 165
D.2 Take off manager: *ROSMDB* model-checking screenshot . . . 166
D.3 Take off manager: feedback results example screenshot . . . 167
D.4 Lock North manager: *ROSMDB* model screenshot . . . 168
D.5 Lock North manager: *ROSMDB* model-checking screenshot . . . 168
D.6 Lock North manager: feedback results example screenshot . . . 169
D.7 Drone Movement Manager: *ROSMDB* model screenshot . . . 170
D.8 Drone Movement Manager: *ROSMDB* model-checking screenshot . . . 170
D.9 Drone Movement Manager: feedback results example screenshot . . . 171
D.10 Networking: *ROSMDB* model screenshot . . . 172
D.11 Networking: *ROSMDB* model-checking screenshot . . . 172
D.12 Networking: feedback results example screenshot . . . 173
D.13 Command sender: *ROSMDB* model screenshot . . . 174

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

D.14 Command sender: *ROSMDB* model-checking screenshot 174
D.15 Command sender: feedback results example screenshot 175

E.1 Engine stopper: *ROSMDB* model screenshot 177
E.2 Engine stopper: *ROSMDB* model-checking screenshot 178
E.3 Engine stopper: feedback results example screenshot 179
E.4 Avoidance: *ROSMDB* model screenshot 180
E.5 Avoidance: *ROSMDB* model-checking screenshot 180
E.6 Avoidance: feedback results example screenshot 181
E.7 Image analyzer: *ROSMDB* model screenshot 182
E.8 Image analyzer: *ROSMDB* model-checking screenshot 182
E.9 Image analyzer: feedback results example screenshot 183
E.10 Mover: *ROSMDB* model screenshot 184
E.11 Mover: *ROSMDB* model-checking screenshot 184
E.12 Mover: feedback results example screenshot 185

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

List of Tables

2.1 Architecture 26
2.2 Infrastructure 27
2.3 Usage 29

4.1 RESTful API for SDfR protocol. 66
4.2 Test-cases for static scenario. 69

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Part I
Introduction and background

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

1 Introduction

1.1 Context

One of the common practices while creating new robotic applications is to start with a model that later will become a software component. We think that the robotic application development can be improved by the interaction with software engineering. As an example, *Robot operating system (ROS)* is starting to be largely used nowadays in robotic application in both academia and industry. But, as shown later, *ROS* and the other existing middlewares need an extension in order to facilitate the development, deployment and run-time of an application designed for a fleet of robots.

Moreover, robotic applications are often developed from a behavioral model. These models could be validated using different techniques and formalisms like model checking. Techniques from software engineering field like *Model driven development (MDD)* could also be applied into multi-robot applications, allowing for a faster and better mapping of the model to the executed source code of the fleet application.

Furthermore, in the context of a robotic fleet application, the peers need to be able to communicate in order to share information and cooperate. Communication in multi-robot systems is a central and challenging issue, whether the architecture is centralized or decentralized.

1.2 Key research issues

The existing middlewares have improved the development of robotic applications. But there is still a gap between the software experience brought by the middleware (e.g. modularity, abstraction, scalability, etc.) and the commonly used practice in robotics development (e.g. model based behaviour, mission planning, etc.).

These middleware for robotics are designed and adapted for single robot applications. Their usage can be affected by expertise needed and their complexity. Furthermore, their applicability in multi-robot context is not adapted. Their usage can be extended to multi-robot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

applications by developing new components.

One of the concepts used in middlewares and sometimes in robotics development is modularity. The robotic software architectures could be designed using *Service Oriented Architecture (SOA)* in which modules/components become services. This allows the design of model that could be composed with others to generate (a) service(s). This granularity could increase the extensibility and the scalability of the new robotic application.

The first question that arises is how can an application whose behavior was designed using a model can be automatically verified based on defined properties before starting the source code development process? Next question is how can this model based application be applied to an existing middleware and which is(are) the adapted programming paradigm? Furthermore, can the process of development, deployment and run-time monitoring be simplified and automatized? Finally, can the run-time behavior be analyzed and compared to the original model?

1.3 Contributions Overview

In order to answer these question, we have first started to analyze the common element of all the life-cycle of an application: the communication inside the fleet. The focus of our research targets the distributed architecture where each peer can communicate with each other without a centralized infrastructure in an ad-hoc network.

We have noticed the absence of dedicated mechanisms that will allow the robotic applications to be aware of the near-by peers and what services they are offering. In order to answer this matter, we propose and study a service and neighbors discovery protocol that allows an ad-hoc fleet of robots to know at any time the reachable peer and what are the services/components on this peer.

Another objective of this research was to build an adapted programming model based on our observation of the commonly used practice of robotic development. We present a model based programming approach that offers properties validation based on timed automata models. This model checking is done at the conception phase. The tool we propose is also capable of translating the interaction between models of a robotic application to an automatically generated *ROS*[1] based code skeleton.

To simplify the process of development and fleet provisioning, we propose an automated mechanism capable of distributing new software components in an ad-hoc, highly mobile, fleet of robots. This allows robots to be capable of auto-provisioning by automatically discovering their peers and being able to self-install software modules and libraries used by the these peers in the fleet. Furthermore, robots benefit from self-profiling which allows them to set up and launch software components and services without the direct intervention of external

[1] Robot Operating System

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

components.

There is still a gap between the modelled application and the final software. Another objective of this work is to reduce this gap by proposing a *ROS* based framework that is checking the correctness of models interactions at run-time and offers adapted support for *ROS* based and fleet communications. The interactions and the model guard violations can be analyzed via the monitoring system we propose. The entire life-cycle can then be reiterated in order to refine the behavior of the application.

1.4 Outline

This thesis has been divided in the following two parts.

Introduction and background The present chapter (chapter 1) provided an introduction. Chapter 2 is an overview of the existing dedicated robotic middlewares. We also provide an extended comparison between the selected middleware and conclude on their applicability in multi-robot context. Finally, chapter 3 provides background knowledge on formalisms to design robotic behaviors. It focuses on timed automata formalisms and it introduces *Model driven development (MDD)*. This is especially interesting in the context of this work, as we make a different usage of timed automata and *MDD*.

Model driven multi-robot applications development Chapter 4 presents the challenges of service discovery in robotic fleet context and proposes a service discovery protocol called *Service Discovery for Robots (SDfR)* using *MDD* on a *Service Oriented Architecture (SOA)*. Chapter 5 presents our proposal for a complete tool-chain, called *Robot operating system Model Driven Behavior (ROSMDB)*, that uses timed automata to model the behavior of each services in a *SOA*, verifies the correctness of the model in the design phase by verifying liveness and reachability properties, provides a code skeleton and a dedicated framework for the development phase of the applications, integrates an automatically deployment system using *SDfR* for the robotic fleet and provides a trace collection and monitoring system for the run-time. A series of scenarios and experimental applications are presented to evaluate the proposed *ROSMDB* approach in chapter 6. Finally, chapter 7 opens perspectives for future work, and gives hints for applying the contributions of this work.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

2 Middleware for robotics

This chapter presents a summary of the existing classes of middleware, focusing on these applied to robotics and compares a limited number of selected middlewares for multi-robotic systems.

2.1 Introduction 7

2.2 Middlewares in distributed systems 8

2.3 Challenges for middleware in robotics 16

2.4 Existing middlewares 20

2.5 Comparative criteria 23

2.6 Middleware Comparison 25

2.7 Conclusion 30

This chapter gives some background knowledge on existing middleware commonly used in robotic applications. It first begins with a brief description of various families of middlewares used in general distributed systems and in robotics systems, then it defines, from our point of view, the challenges a multi-robot middleware may encounter. It then reviews some of the existing middleware for single robot application and study their applicability in a multi-robot context.

2.1 Introduction

Nowadays, distributed computing systems are everywhere. In this kind of systems, there is a great need for distributed elements to interact in order to share information or consume each

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

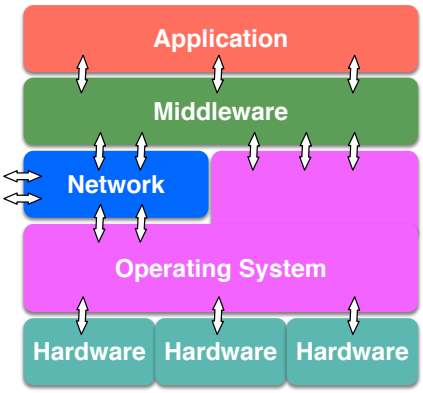


Figure 2.1 – Middleware layer interaction with the runtime stack

other functionalities. This trend is accelerated by the *"information technology of all forms becoming highly commodities (i.e. hardware and software artefacts are getting faster, cheaper, and better at a relatively predictable rate)"* and by the *"growing acceptance of a network-centered paradigm"* [Schantz and Schmidt, 2002]. Even if this approach of distributing components can be developed directly over the operation system and network layer, it will generate yet another layer of complexity that need to be managed. In order to avoid this, the new layer of complexity can be delegated to the appropriate families of middlewares.

We start with an overview of the families of middleware used in general distributed systems and later we will focus on the advantages of using middleware in robotics systems. The next section continues this discussion by presenting the challenges that a family of middlewares needs to solve in order to be used in robotics.

2.2 Middlewares in distributed systems

A family of middlewares is composed of software and tools sets that act as abstraction and integration layers between network, operating system and applications as shown in fig. 2.1. The middlewares are an important component in the process of developing, deploying and operating new software.

The main purposes of using a family of middlewares are:

- Facilitate the development and evolution of distributed systems
- Orchestrate the interconnection and communication of application components
- Allow the inter-operability, portability and integration of components using different technologies

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

2.2.1 Classes of middlewares

In general, the families of middleware can be regrouped [Qilin and Mintian, 2010] using their mechanism to communicate with distributed components in:

- *Remote Procedure Call (RPC)* and *Distributed Object Middleware (DOM)* middleware

  A *RPC* [Birrell and Nelson, 1981] represents an action triggered by a program in order to execute a subroutine in another process, usually on another network shared computer. This procedure is developed as it was a local subroutine call, without the complexity of the remote interaction. This allows to have the same functions whether the call is local or remote. It can be seen as a client-server architecture where the client is the process calling the execution of the subroutine and the server is the executor. The communication is realized via message-passing system [Waldo, 1998].

  A *RPC* middleware offers the following services [Issarny et al., 2007]: generating client and server stub[1], marshalling[2] / un-marshalling data and establishing synchronous communication. As show in fig. 2.2, a *RPC* is initiated by the client via its stub which sends a request message to a known remote server to execute a specified procedure. The remote server replies with the result of the execution, and the application continues its processing.

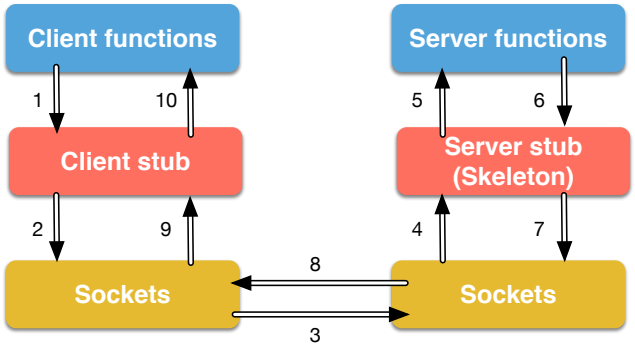


Figure 2.2 – Steps in executing a RPC

The biggest inconvenient for *RPC* is the possibility of the call to fail because of unpredictable network problems. The caller must deal with such failures without knowing whether the remote procedure was actually invoked. Idempotent procedures (those that have no additional effects if called more than once) are easily handled. Another problem is the limited use of parallelism via multiple threads since *RPCs* are synchronous [Qilin and Mintian, 2010].

[1] A object participating in distributed object communication acting as a proxy object. For the server side is also known as skeleton.

[2] The process of transforming the memory representation of an object to a data format suitable for storage or transmission, and it is typically used when data must be moved between different parts of a computer program or from one computer to another.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

DOMs is a middleware class that provides communication between a client *object* that executes an operation on a server *object* that resides on another host [Capra et al., 2001]. *DOMs* evolved more or less directly from the idea of *RPCs*. The main difference between those two is the tight link between *DOMs* and *Object-oriented programming (OOP)* [3].

DOMs offers a great interoperability between heterogeneous platforms and components. It provides an abstraction layer for remote objects whose methods can be invoked like the object is part of the same runtime as the requester. It allows all the benefits of *OOP* like inheritance, polymorphism or encapsulation to be applied over distributed objects across a network.

A *DOM* offers mechanisms [Issarny et al., 2007] to generate stubs for object interfaces, get and access references on remote objects and provides synchronous communication to invoke methods by marshalling/ un-marshalling the requests.

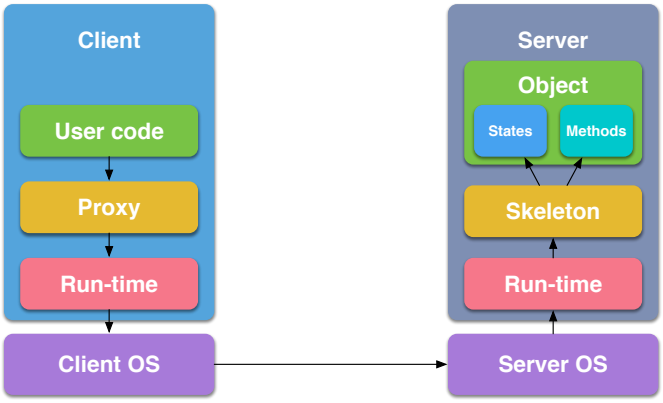


Figure 2.3 – DOM architecture

A typical *DOM* architecture is presented in fig. 2.3. A client object invokes methods on a proxy object residing on the same host. The proxy object marshals the request and sends the invocation request to a remote object server. The server runtime dispatches the request to an appropriate object skeleton who is responsible for un-marshalling the request and invoking the appropriate methods on a local object instance. The results are sent via the same procedure.

- *Message-Oriented Middleware (MOM)*

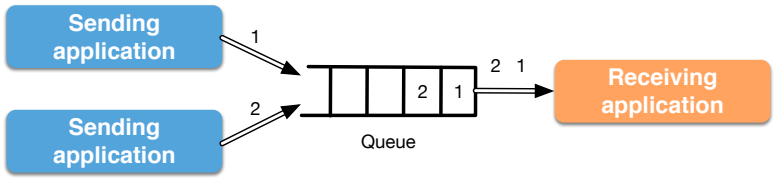
  A *MOM* represents a software layer that provides mechanisms for sending and receiving messages between distributed systems. It allows the integration of software modules that reside over heterogeneous platforms and it reduces the complexity of the client-server architecture. The middleware offers a distributed communication layer which provides application sand-boxing[4] from the heterogeneous network and operating

[3] *OOP* is a programming paradigm based on the concept of *objects*, which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.

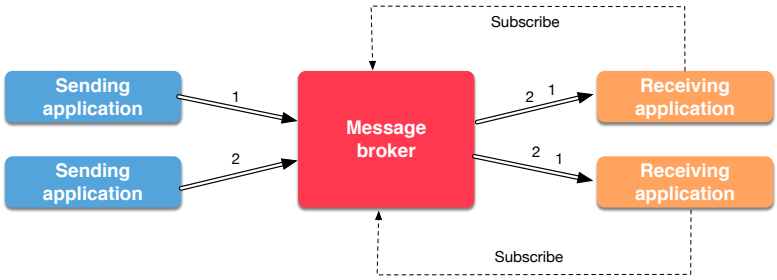
[4] Component isolation or sand-boxing is a security mechanism for separating running process. The code and data spaces are also separated for each process.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

system layers. Those *Application programming interfaces (APIs)* are typically provided by the MOM [Curry, 2004].



(a) Queue MOM



(b) Pub/Sub MOM.

Figure 2.4 – Steps in sending a message in a MOM

Even if all *MOMs* support a communication mechanism where clients send messages with their requests for a service execution to (a) server(s) across the network, which later responds with the result of the execution [Capra et al., 2001], they can be classified in two categories based on their mechanism to relay messages:

- Queue-based middlewares

  Queue-based *MOMs* are based on a one-to-one architecture as show in fig. 2.4a. The receiver application has a queue where it stocks all the messages received from all the client applications. It then processes those messages based on a *First in First out (FIFO)* or custom policy.

- Publish/Subscribe middlewares

  Publish/Subscribe *MOM* provides an architecture where the messages are routed by a central element: a message broker. Receivers need to subscribe prior to receiving messages. As show in fig. 2.4b, the main difference from a queue-based *MOM*, is the architecture that allows messages to be received by all the subscribed receivers. This allows m to n communications.

MOMs can also include features like message persistence and replication. They can also provide time-bound *Quality of service (QoS)* performance and increase the scalability and security of applications. They can guarantee durability, which is essential for some types of distributed system interactions. Furthermore, since the architecture is based on a client/server model, they support asynchronous communications [Capra et al., 2001, Issarny et al., 2007].

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

- *Transaction-Oriented Middleware (TOM)*

  A transaction represents a task usually executed within a distributed system that requires consistency and reliability [Capra et al., 2001]. A transaction is executed independently from other transactions in a reliable and coherent way. It usually represents any change in database. It ensures that the task is correctly recovered from failures and the database remains consistent even in case of an incomplete execution stop. Furthermore, a transaction guarantees the sand-boxing between applications accessing distributed components. In other words, a transaction is a mechanism of coordination among distributed systems that respects the *Atomicity, Consistency, Isolation, and Durability (ACID)* properties [Issarny et al., 2007].

  A *TOM* can provide the correctness of transaction operations within a distributed system: a client accumulates several tasks in a transaction which is route to a server via a network in a transparent way. The main downside of *TOMs* is the significant overhead generated by the respect of the *ACID* properties, and, as showed in [Issarny et al., 2007], it often offers unnecessary *QoS* guarantees.

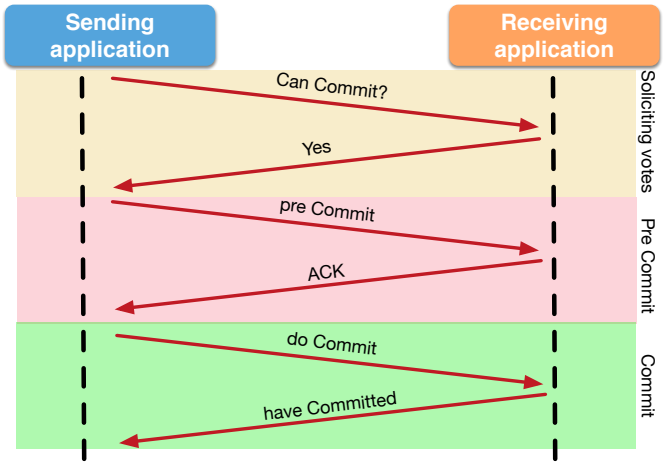


Figure 2.5 – Three phase commit protocol

TOMs supports both asynchronous and synchronous communication among heterogeneous hosts providing an easy mechanism for clients, servers and database management systems to cooperate and provides high reliability. If all the participants implement a two-phase-commit [Cotner et al., 1999] or a three-phase-commit protocol [Al-Houmaily and Samaras, 2009], as show in fig. 2.5, the *ACID* properties are maintained. In a three-phase-commit protocol, when a node receives a transaction request, the system enters a soliciting votes state. The peer sends request a decision to commit from the cohort nodes and moves to the waiting state. If there is a failure, timeout, or if the coordinator receives a negative message in the waiting state, the coordinator aborts the transaction and sends an abort message to all cohorts. If the coordinator succeeds in the pre-commit state, it will move to the commit state.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

- *Service Oriented Architecture (SOA), Service Oriented Middleware (SOM)* and *Enterprise Service Bus (ESB)*

  SOM is a class of middleware systems based on *SOA*. A *SOA* is an architectural pattern in software design in which application components provide services to other components via a communications protocol, typically over a network. The principles of service-orientation are independent of any vendor, product or technology [Papazoglou et al., 2007]. A service is defined as a loosely linked set of functionality that is self-contained. A service needs to implement at least one specific action like requesting the value of a sensor, updating a mission configuration or changing the environment settings.

  SOA is bounded on service-orientation by its fundamental design principle. The composition of the services is transparent for the user since each service should define an interface which abstracts the underlying complexity and its platform implementation [Channabasavaiah et al., 2003].

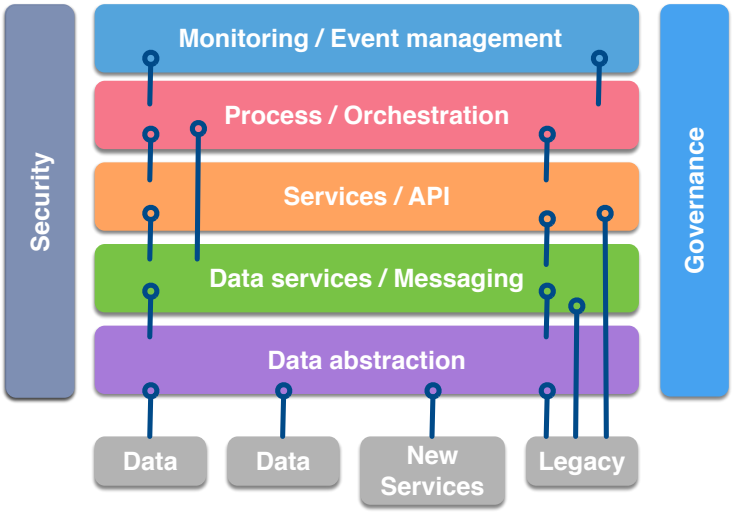


Figure 2.6 – SOA meta-model based on *The Linthicum Group, 2007*[5]

Figure 2.6 represents one view of *SOA*, working up from the data, to the data abstraction layer, to the data services, to the services, to the orchestration layers, and finally monitoring and event management. Both governance and security are linked to all layers. Depending on the requirements and functionality of each service, if this stack does not meet the specifications, each service is open to use any software stack as long as it provides an abstraction interface for the other peers to use it.

In a *SOA*, services need to communicate in order to exchange information and perform actions. Services need to implement protocols that specify how data is parsed and passes using metadata. The information in the metadata can describe the functionality

[5] http://semanticommunity.info/@api/deki/files/7640/=EA07_Keynote_Linthicum.pdf

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

of the service, as well as the mechanism to marshalling/ un-marshalling the information used by the service. *SOA* description metadata should comply with the following criteria:

- The metadata should be shared using a well-defined serialization format that allows other components the discovery and incorporation of the service, but also to maintain integrity and coherence. The metadata can be used by other services to dynamically discover the services without modifying the functional contract of a service.
- The metadata serialization type should be readable with a limited cost and effort.

The main advantage of *SOA* is the ability of combining an elastic number of functionality in order to create an ad-hoc application created entirely form existing services [Bell, 2010]. The larger the number of functionality implemented by a service implies a smaller number of services use, thus a fewer interfaces required to combine the services. However, the services need to implement a limited number of functionalities in order to maintain the granularity of each services and the easy reuse of them. Because each service interface has its overhead, the performance of the entire application is related to the number of services and their granularity.

SOM enhances *DOM* by the concept of services. A service is represented by a group of *objects* and their behavior. These *objects* offers an external interface in order to allow the services to be used form other distributed components. It also provides communications protocols between services.

SOM is composed of three main components: a service provider, a service requester and a registry. It allows support for service providers to deploy their components and further publish their presence to the registry. It usually includes a mechanism to discover the published services. *SOM* also provides an abstraction of the heterogeneity of the services. The communications can be established in both synchronous and asynchronous way.

ESB is another middleware that proposes a software architecture for communication between components in a *SOA* [Schmidt et al., 2005]. Its architecture and communication model is derived from the client-server paradigm and it facilities the agility and flexibility of the communication mechanisms between components. It is mostly used in the integration of heterogeneous enterprise software components.

Based on the bus paradigm which is found in many hardware architectures, *ESB* benefits of the concurrent and modular model-design of modern operating systems. *ESB* is used to structure and design the implementation of loosely coupled services that are deployed and run independently on network distributed architectures. The main difference between *ESB* and *SOM* is the absence of a central broker, making the *ESB* more flexible and scalable for enterprise-wide solutions.

As shown in fig. 2.7, *ESB* is capable of allowing heterogeneous components that communicate using various marshalling formats like *JavaScript Object Notation (JSON)* or

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

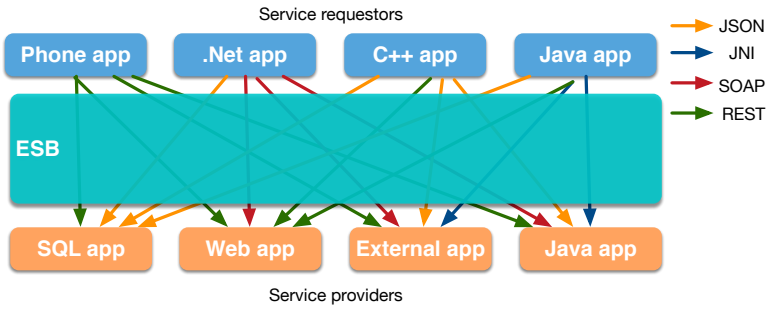


Figure 2.7 – Serial Bus communication architecture

Simple Object Access Protocol (SOAP). It translates the message to the correct type prior to sending it to the right service provider/requester.

The next subsection presents how this classes of middleware are applied to robotics systems and next we present which are the challenges for a robotic middleware.

2.2.2 Middlewares in robotics systems

These families of middleware presented above are applied to general distributed systems. One group of these systems is represented by (multi)-robot systems. Generally, a robot is a complex and heterogeneous distributed system that requires communication and interaction between robot components (various sensors, actuators and software components). An autonomous robot fleet refers to multiple robots (two at least) capable of sharing data and performing one or several tasks together. It can also include mobile or fix connected objects and sensors cooperating together to achieve a common goal.

As mentioned in [Ferber, 1999], in the field of distributed artificial intelligence, the division of tasks of a greater problem reduces the complexity and the difficulty of a problem, even if this requires coordination mechanisms. In the challenge of having large scale multi-robot systems there is a need of information and services sharing between robots and external objects.

Despite many years of work in robotics, there is still a lack of a software architecture and well-accepted family of middlewares [Smart, 2007]. This makes sharing modules and algorithms almost impossible in practice. Furthermore, the robots different hardware, thus different software architectures. This interoperability[6] is a vivid example of the sharing problem.

A family of robotic middlewares should manage heterogeneity of the hardware, facilitate the communication inside and outside a robot, improve software quality, reduce time and costs in order to build new applications, allow robots to be self-configuring, self-adaptive

[6] "Interoperability is a characteristic of a product or system, whose interfaces are completely understood, to work with other products or systems, present or future, in either implementation or access, without any restrictions." [McCreesh and Daniel, 2014]

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

and self-optimizing to environment changes. Combining component and service-oriented programming greatly simplifies the implementation of highly-adaptive, constantly-evolving applications [Frénot et al., 2010]. In our vision, robots could be capable of auto-provisioning by automatically discover their peers and being able to self-install software modules and libraries used by these peers in the fleet. Furthermore, robots could benefit from self-profiling which allows them to set up and launch software components and services without the direct intervention of external components.

Using the appropriate family of middlewares, multi-robot systems can increase their computation power using external architectures like data-grids [Torkestani, 2013] or *clouds* for robots. The main advantage of a *cloud* of robots is the decreased time of computation as it is parallelized, since the computation is executed into a datacenter with many *Central Processing Unit (CPU)* working on the same task. This approach has also its downsides, since each robotic system has to communicate and share information with a centralized system hosted in a datacenter using Internet network.

However, there is a convergence trend between the robotic and the middleware world, in order to build efficient middleware solutions for robotics. This trend establishes a more typical loosely-coupled, layered software architecture as found in traditional general-purpose software engineering.

There already exists middlewares that try to achieve parts of the desired needs. Most of them are designed for single robot contexts and they can also be used in a fleet context, but there also exist new *cloud* based approaches designed for multi-robot goals. The next section discusses the needs of having a family of middlewares in large scale multi-robot systems and how it facilitates software development. We compare the different existing solutions presenting the advantages and down-sides of the existing middleware based on several criteria that cover the architecture, infrastructure and use of each framework.

2.3 Challenges for middleware in robotics

As seen in the classical distributed computing, middleware is an important asset on which relies the development of new applications since it can hide the complexity of the heterogeneous components by providing a layer of abstraction, it can offer value-added components and functionality and it can facilitate the deployment of new services.

Nowadays, robots are more and more used in a fleet context, being capable of having a global environment perception and also a communication inside the fleet and with external communicating objects like sensors, network and service gateways, mobile devices with wireless capabilities. The robots are often heterogeneous. More, all the devices and the robots themselves are made of a diversity of hardware controlled by a variety of software developed in different programming languages using multiple standards and protocols to communicate. Robotic middleware could be used to manage this heterogeneity and interoperability

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

problems.

2.3.1 Why a middleware for robotics ?

All these aspects of communication, application deployment and configuration can be facilitated using a proper middleware. The biggest difference between a classic middleware that runs in a *cloud* infrastructure and a robotic one, is the mobility of the fleet and the decentralization of its components. Furthermore, a datacenter has a reliable and stable network, while in a robotic fleet context the network is considered unreliable and changing due to the mobility of the robots, thus the robotic applications must compensate for this problem.

One of the challenges is software modularity as presented in [Elkady and Sobh, 2012]. In the fleet context, task dedicated software modules can be composed in order to form a complex behaviour for all the peers in the fleet.. The robotic applications development need to embrace a more software oriented modular vision. The software design should emphasize separating functionalities and algorithms into independent, interchangeable, cross-platform modules. Applications for multi-robot environments are not easy to develop. The development process should be simplified by integrating higher-layers of abstraction with *APIs* [Mohamed et al., 2008]. Old modules and code should be easy to integrate in new projects, even if the robot architecture is different. Also, the middleware should support plug-and-play mechanism for new developed modules, being capable of hot swapping new packages.

Furthermore, a robotic middleware should integrate the functionalities of a classic middleware [Issarny et al., 2007]. It should have the properties of scalability of a *MOM*. The middleware should be service oriented in order to allow robotic services to be published by the providers and discovered by the consumers.

2.3.2 Infrastructure and communication

The software components of a robotic application should run on any infrastructure, which implies that the middleware should propose a hardware abstraction layer in order to facilitate the reuse of the modules. This need is generated by the heterogeneous hardware and software involved in operating a robot. The middleware should hide the complexity and diversity of the components and provide a mechanism of self-service discovery of its hardware elements. The robots can be based on different architectures, using different sensors and actuators which offer a variety of services. The middleware should make the robot aware of its capabilities by automatically discovering the sensors and actuators running on the robot.

Those capabilities should be organized in robotic services that should be broadcasted to allow each robot to know what its team members are capable of. Such automatic resource and service discovery and configuration mechanisms could increase the potential of a robot. Since the robots are part of environments that can evolve, move and be dynamic, they need to

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

organise[7] inside the fleet in a decentralized network.

Also, due to the mobility of robots, the fleet can divide or regroup itself at a communication layer (physical layer) but keeping the same fleet configuration at an application layer, allowing the members should self-adapt to the new fleet-configuration [Valle et al., 2013]. New robots can be integrated into the fleet meaning that a robot should be capable of self-profiling and self-provisioning. It is very useful to have a mechanism that allows to deploy new non-configured robots into a fleet and to have them automatically perform packages update and service configuration based on the fleet previous profile.

Furthermore, having a layer of hardware abstraction, the software will be platform independent. This means that the robots can be built with different hardware, sensors and actuators. All these internal components need to share information in order to make the robot function correctly. The middleware should provide a system of information sharing and collaboration among all involved components offering communication support and interoperability. It should make this system transparent to the developer by masking the low-level communication with a more human-comprehensible language. Also, this system should be extensible at a network layer, allowing direct information sharing across the fleet. The network communication layer should support both centralized networks (access points, media gateways), as well as ad-hoc networks that could be created on demand to allow communication across the fleet.

2.3.3 Application programming interfaces and services

The middleware should also provide collaboration support among the robots making sure that all robots share the same values of shared information. Also, it should provide *API* that will make the development of multi-robot collaboration application easier. This will make the application layer portable across fleets of different architecture robots. The robotic application developer will not have to program consensus mechanism for networked shared memories or rewrite network modules to share services. The development time of a new multi-robot application will decrease, facilitating the research effort to discover new algorithms and applications for robot fleets.

Another challenge for a robotic fleet middleware is to provide specific uses for a robot. Well-known functionality like *Simultaneous localization and mapping (SLAM)*, obstacle avoidance, autonomous navigation in a known map or object follower should be provided by the middleware. Without a middleware, the same known algorithms must be rewritten depending on the robot evolutionary hardware. Those features should use hardware abstraction message interfaces that should be independent of the platform that the middleware is running onto, supporting a large number of inputs (different sensors, actuators). This can be done using the previous explained hardware abstraction layer and software modularity.

[7] e.g. A leader election using a peer to peer biding system

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

2.3.4 Robotic applications as services

In our vision, robots need to advertise their functionality as services in order to allow other members of the fleet to interact with them. In network based application, service-oriented programming is now a largely accepted principle [Issarny et al., 2011].

Service-oriented architecture greatly simplifies the implementation of highly-adaptive, constantly-evolving applications [Frénot et al., 2010]. It also reduces the process of developing and deploying new robotic applications. This architecture is very suitable to quickly cope with new developing models and requirements.

Services can become the basic blocks of complex robotic behaviors and applications. This provide sand-boxing for each software component which renders the robotic application more robust and tolerant to failure and still disposing of the flexibility in developing new components.

Services are platform independent and they can be described, discovered and composed dynamically. Having a service oriented architecture increases the ability to develop distributed software components in various programming languages and for heterogeneous target devices. In addition, higher levels of functionality provided by service-oriented programming reduce the implementation of redundant software.

In centralized communication systems, robots mobility is reduced because they cannot move outside the coverage area of the infrastructure. Moreover, if the central node fails, the whole communication of the fleet stops. In order to increase the mobility of the robots and to distribute the communication without having a central node, there is a need for the communication to be decentralized using ad-hoc networks. In this case, the robots do not have a complete communication scheme of their nearby neighbors. Furthermore, the communication across peers is susceptible to route change and different peers can be used to relay a data package as shown in Figure 2.8. The ad-hoc network becomes the sum of peer to peer network across at least 2 robots.

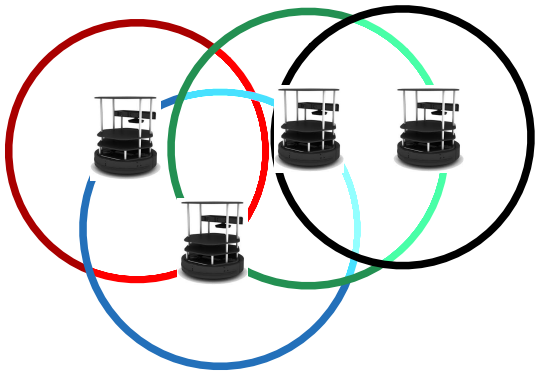


Figure 2.8 – Ad-hoc fleet infrastructure with per-robot communication ranges.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

2.3.5 Limitations

Even the most elaborated middleware might have problems. As mentioned in [Smart, 2007], the fact of having a hardware abstraction layer that hides the heterogeneity of the sensor and actuators has its down-sides. The specificity of sensors, their position, their limits and failures, the shape of the robot increases the complexity of a controlling software. Extrapolating and/or integrating these assumptions makes the middleware more complex and more failure prone. Also, the heterogeneity of a robot fleet means that different robots have different resources like computation power, hardware capabilities, battery life, which makes the design of a middleware further difficult.

While in classic *cloud* environment the network could be considered as *almost* totally-reliable, in a robotic fleet context is susceptible to frequent failures. The middleware should not try to catch a network failure exception, but instead accept that the network is temporary unreachable and operate in a degraded mode until the network communication is reestablished. The same logic should be applied also in case of hardware failure since robots usually run in hazardous environment.

Taking everything into account, the challenges for a multi-robot middleware are high. There are lots of techniques and research done in *cloud* middleware that can be applied into a fleet context. However, there is a lot of differences between a *cloud* and a fleet due to mobility and communication limits inside a fleet. Up to now, many attempts into creating a promising middleware for robots have been done.

The next sections will present and compare the most relevant middlewares for robotic fleets.

2.4 Existing middlewares

In this section, we present the most used middleware with applicability in a fleet. A complete survey of all the middleware for single robot contexts is clearly impossible because of the large number of existing middleware and frequent releases of new ones. To reduce the amount presented, we first considered their compatibility in a multi-robot environment and the number of citations. Based on that, we have selected eight most used robot middleware:

- *Player/Stage*
- *ROS*
- *Miro*
- *Microsoft Robotics Developer Studio (MRDS)*
- *Mobile and Autonomous Robotics Integration Environment (MARIE)*
- *Orca*
- *Carnegie Mellon Robot Navigation Toolkit (Carmen)*
- *Python Robotics (Pyro).*

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

The reader should keep in mind that there are also other available middlewares. Some of the middlewares worth mentioning are:

- *Claraty* [Nesnas et al., 2006]
- *OpenRTMaist* [Chishiro et al., 2009]
- *OPRos* [Jang et al., 2010]
- *Orocos* [Soetens, 2010]
- *ERSP* [ERSP, 2010]
- *RoboFrame* [Petters et al., 2007]
- *WURDE* [Heckel et al., 2006]
- *Aseba* [Magnenat et al., 2010]
- *Skilligent* [Skilligent, 2010]
- *SmartSoft* [Schlegel et al., 2009a]
- *iRobotAware* [iRobotware, 2010]
- *Yarp* [Fitzpatrick et al., 2008]
- *Spica* [Baer et al., 2008]
- *Babel* [Fernandez-Madrigal et al., 2006]
- *DROS* [Dave, 2009]
- *IRSP* [Kwak et al., 2006]
- *K-MIDDLEWARE* [Choi et al., 2006]
- *OpenRDK* [Calisi and Censi, 2009]
- *OpenJAUS* [openJaus, 2010]
- *ORCCAD* [Arias et al., 2010]
- *RIK* [Bruemmer et al., 2006]
- *MRPT* [MRPT, 2010]
- *MissionLab* [Endo et al., 2004]
- *Webots* [Michel, 2004]

The following sections gives a brief overview of each selected software. A larger description of them, the compatible robotic platforms and the most relevant features can be found in Appendix A. Then we propose in sections 2.5 and 2.6 a comparison between them.

Player/Stage

The Player/Stage ([Kranz et al., 2006], [Collett et al., 2005]) project is designed to provide an infrastructure, drivers and a collection of dynamically loaded device-shared libraries for robotic applications. It is one of the first middleware that emerged for robotic systems and there are other middlewares that wrap Player. It doesn't consider a robot as a unity, but it instead treats each device separately, being a repository server for actuators and sensors. Players main futures are the device repository server, the variety of the programming languages, the socket based transport protocol, modularity and the implementation being open-source.

Robot operating system (ROS)

ROS is a recent flexible middleware for robot applications [Cousins et al., 2010, ROS, 2014]. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. It provides

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

hardware abstraction, device drivers, visualizers, message-passing, package management. *ROS* is composed of two key components: the *ROS* master and *ROS* nodes. The *ROS* Core is composed of the master node (a name server that allows node to subscribe and keeps tracks of each created node and topic) and *ROS* parameter server (a shared, multi-variate dictionary that is accessible via network *APIs*). The *ROS* nodes are executables that use *ROS* to communicate with other nodes and represent the application layer of the architecture.

Miro

Miro is a distributed, *object*-oriented middleware developed to improve the software development process by increasing the integrality of heterogeneous software, the modularity and the portability of robot applications [Kraetzschmar et al., 2002, Krüger et al., 2006]. It was developed in *C++* for Linux based on the *Common Object Request Broker Architecture (CORBA)*. This allows cross-platform interoperability making the middleware applicable to a distribute multi-robot context. Due to the restrictive nature of *CORBA*, software application can be only written in languages that provide *CORBA* implementations.

Microsoft Robotics Developer Studio (MRDS)

MRDS is a Windows-based middleware for robot control and simulation from Microsoft [Johns and Taylor, 2008, MRDS, 2012]. It is composed of four major components: *Concurrency and Coordination Runtime (CCR)*, *Decentralized Software Services (DSSs)*, *Visual Programming Language (VPL)* and *Visual Simulation Environment (VSE)*. *MRDS* is aimed at academic, hobbyist, and commercial developers. It handles a wide variety of robot hardware like Eddie Robot, ABB Group Robotic, CoroWare CoroBot, Lego Mindstorms NXT, iRobot Create, Parallax Boe-Bot and more.

Mobile and Autonomous Robotics Integration Environment (MARIE)

MARIE is a middleware designed to allow the integration and distribution of software for robotic systems [Côté et al., 2007, Côté et al., 2006]. Its main objectives are to allow developers share, reuse and integrate software in order to accelerate the development of robotic applications. It was created in *C++* and uses the *Adaptive Communication Environment (ACE)* communication framework. The centralized component provided by the middleware called *Mediator Design Pattern (MDP)* allows software components to connect to *MARIE*.

Orca

Orca is an open-source middleware for developing component-based systems [Makarenko et al., 2006, Makarenko et al., 2007]. It provides the mechanics to create building-blocks which can be pieced together to form arbitrarily complex robotic systems. Orca can

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

be used in various applications, from single vehicles to distributed sensor networks. It was designed and developed to maximise the software reuse and modularity in robotic applications. Orca is highly dynamic, with a distributed component base system that allows the user to define custom interfaces and communication protocols.

Carnegie Mellon Robot Navigation Toolkit (Carmen)

Carmen is an open-source collection of middlewares that focuses on the robot control by providing various control interfaces [Montemerlo et al., 2003, CARMEN, 2008]. It is written in the *C* programming language and it is organized in three layers: hardware interface, common services and application layer. The hardware interface provides low-level communication, control by creating a hardware abstraction for sensors and other components. The second layer offers off-needed robotic services like navigation, localization, object tracking, and motion planning. The last layer is represented by the user-defined applications that share information and relies on data revived from the lower layers.

Python Robotics (Pyro)

The goal of *Pyro* is *"to provide a programming environment for easily exploring advanced topics in artificial intelligence and robotics without having to worry about the low-level details of the underlying hardware a robot programming environment"* [Blank et al., 2006, Blank et al., 2005, Pyro, 2012]. It has an educational purpose, and it wraps the Player/Stage middleware so that any component written for this system is also available to *Pyro*.

2.5 Comparative criteria

The comparison of the eight robotic frameworks presented is done from a software engineering vision. It groups the comparative criteria into tree major groups: *Architecture*, *Infrastructure*, *Usage*. Each major group is composed of different criteria relevant to the group.

The *Architecture* evaluates the impact that the framework has over the host operating system and it is composed of:

Overhead (OV) - the consumption of hardware resource that is added besides the operating system. It is important that a middleware to have a lower overhead in order to limit the resource consumption (energy, *CPU*, memory) especially for real-time systems.

Vendor locking (VL) - the middleware operating system dependence. This criteria expresses the portability of a system across multiple platforms and systems.

Robustness to failures (RF) - the detection of a software failure, any degraded model to run and the afterwards recovery process. The fact that a middleware is aware of failures is

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

essential for the robotic applications. Furthermore, it is important that robots continue performing their tasks in a degraded mode until the system has recovered from the failure.

The *Infrastructure* evaluates tools and *APIs* provided by the middleware and it is composed of:

Management and monitoring (MM) - tools provided to manage, debug, configure and monitor the middleware components. Since robots are complex devices, it is important to facilitate the supervisor task by offering a complete vision of the sensors, actuators and other components status of each robot.

Multi-robot coordination services (MCS) - tools to make consensus over network shared values, to elect a leader or to assign specific robotic tasks. Inside a robotic fleet, it is important to have management tools to distribute algorithms in order to reduce the complexity of the robotic applications development.

Scheduled operations and tasks services (SOTS) - tools to perform repetitive tasks like Chron job schedulers.[8] Having a scheduler will facilitate certain tasks.

Durable data storage services (DDSS) - tools that allow to persist data from sensors and other robots from the fleet. The data persistence layer is important for saving mission results, for experimental data validations, for off-line data processing as well as for sensors data replay in a simulator.

Communication (COM) - *APIs* for requesting data or setting parameters from components, services, messaging. The communication is very important between different components of a robot in order to allow it to successfully perform its task, as well as inside a fleet in order to allow robots to interact with others.

The *Usage* evaluates the impact of integrating the middleware into new robotic applications and it is composed of:

Deployment and life-cycle (DLC) - the facility to deploy across the robotic fleet, integration with compilations chains, night builds testing environments and life-cycle management of new robotic applications. It is very useful and time reducing to have a building and automated testing environment that allows task simplifications especially in complex distributed systems.

Programming model (PM) - type of programming model that can be used: synchronous[9], asynchronous[10], event triggered, service triggered, etc. Having different programming

[8] The software utility chron is a time-based job scheduler in Unix-like computer operating systems.

[9] Synchronous programming model are used for sequential blocking execution tasks

[10] Asynchronous programming model are used for programming interactive systems that interact continuously with their environment, at their own speed

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

models approach is very useful since a problem can be solved using one type and other problem using other type.

Code and data integration services (CDIS) - the easiness to integrate new services and modules into existing robotic software via *APIs*. Having a layer of interfaces that allows the developer to enrich the robotic applications will simplify the development tasks.

Extension points and interfaces (EPI) - available libraries with *APIs* to use often-needed services. Offering often-used services reduces the time and the difficulty of developing new robotic software.

2.6 Middleware Comparison

This section analyses each middleware based on the criteria presented in the previous section. Each major group is represented as a separate subsection that includes a table that summaries the subsection. The evaluation is relative to all middlewares. A ✚ represents that all the criteria requirements are satisfied, a ➕ represents that most of the requirements are present, a ~ shows the fact that the criteria is partially satisfied, a ⛔ represents that the criteria is not fulfilled and a ▬ represents that not only the criteria is not satisfied but there is a big gap compared to the other middlewares.

2.6.1 Architecture

Table 2.1 summaries the *Architecture* group. It is composed of *Overhead (OV)*, *Vendor locking (VL)* and *Robustness to failures (RF)*.

Overhead (OV) and *Vendor locking (VL)*

The less overhead is owned by *ROS* since communication and nodes name service framework has small overhead. It can run on a machine with less *CPU* power like the Raspberry Pi. The overhead is generated mostly by imaging processing packages that add extra *CPU* usage. Another middleware with small overhead is *Player/Stage*, but its performances are limited by the speed of the operating system. *Pyro* has a larger overhead because it wraps *Player/Stage*. The major *CPU* load for *Carmen* comes from two sources: localization and navigation. *Carmen* can run on machines compatible with RedHat Linux or SuSE. *MRDS* supports only Windows. The overhead in *Orca* is introduced by the use of ICE [Michi Henning, 2010]. The middleware is cross-platform and their performance differs on the operating system. *Miro* has a large overhead introduced by the use of *CORBA*. Also *MARIE* has considerable overhead caused by the additional software for the functional components.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Middleware	*OV*	*VL*	*RF*
`Player/Stage`	✚	Linux Windows	✚
`ROS`	✚	Ubuntu Debian Windows MacOS	✚
`Miro`	▬ *CORBA*	Linux	✚
`MRDS`	▬ *DSS* and *CCR*	Windows	✚
`MARIE`	▬ additional functional components	Linux	✚
`Orca`	▬ ICE	Linux	✚
`Carmen`	⛔	Red Hat SuSE	✚
`Pyro`	⛔	Linux	⛔ Neither degraded mode, nor component isolation

Table 2.1 – Architecture

Robustness to failures (RF)

None of the middlewares has a special dedicated degraded mode and nodes cannot be restarted automatically after failure. Besides *Pyro*, all the middlewares provide component isolation. *Player/Stage* and *Carmen* supports components isolation sandbox due to *Inter-Process Communication System (IPC)* communication. In *Miro* and *MARIE*, ACE objects are providing component sand-boxing. *MRDS* includes *DSS* that provides components isolation. *Orca* uses ICE which provide objects isolation. *ROS* needs an *Internet Protocol (IP)* address at the initialization to run *roscore*.

2.6.2 Infrastructure

Table 2.2 summarizes the *Infrastructure* criteria. It has the following columns: *Management and monitoring (MM), Multi-robot coordination services (MCS), Scheduled operations and tasks services (SOTS), Durable data storage services (DDSS)* and *Communication (COM)*.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Middleware	*MM*	*MCS*	*SOTS*	*DDSS*	*COM*
Player/Stage	~	+ Third-party coordination	~	~	~
ROS	+ Dashboard and management	~	~	+ Rosbags	+ Multiple type communication
Miro	−	~	~	~	~
MRDS	+ Visual Studio plugins	~	~	~	+ Multiple type communication
MARIE	~	~	+ RobotFlow	~	~
Orca	−	~	~	~	~
Carmen	−	− Single robot	~	~	~
Pyro	~	~	~	~	~

Table 2.2 – Infrastructure

Management and monitoring (MM)

Besides *Miro*, *Orca* and *Carmen* which provide neither a monitoring nor a management interface, the rest of the middlewares include monitoring software. *Player/Stage*, *MARIE* and *Pyro* include a graphical interface to display components status and control them. *MRDS* uses the Microsoft Visual Studio IDE that provides monitoring and management interfaces. *ROS* has multiple management tools that include roslaunch, rosrun and parameter server. It has a dashboard monitoring interface that can be access remotely.

Multi-robot coordination services (MCS)

None of the middlewares provide native multi robot coordination services. *Player/Stage* includes third-party coordination algorithms developed for it. *ROS*, *Miro*, *MRDS*, *Orca*, *MARIE* and *Pyro* delegate the coordination services to the application layer. *Carmen* has no multi-robot services since the middleware has a single robot vision.

Scheduled operations and tasks services (SOTS)

The only framework that provides scheduled operations and tasks services is *MARIE*. It is built on RobotFlow that include operation scheduler. RobotFlow is a mobile robotics toolkit based

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

on the FlowDesigner project. FlowDesigner is a data-flow oriented architecture, similar to Simulink. Neither *Player/Stage*, nor *ROS*, nor *Miro*, nor *Orca*, nor *Carmen*, nor *Pyro* provide a dedicated system. Their tasks can be managed by a Linux scheduler. *MRDS* can be included in Windows Task Scheduler.

Durable data storage services (DDSS)

ROS is the only middleware that provides durable data storage services. Topics and service messages can be persisted in rosbags. The other frameworks don't provide any native *API* to save sensor information. *Player/Stage*, *MRDS*, *Orca* and *Pyro* configuration files are stored into text files, while *Miro*, *MARIE* and *Carmen* use *Extensible Markup Language (XML)* files to store configuration variables.

Communication (COM)

Communication between the infrastructure layers in *Player/Stage* and *Pyro* are done using direct socket connections as their primary method of communication. *Miro* data sharing is assigned to *CORBA*'s IIOP, while *Carmen* uses ICE [Michi Henning, 2010]. *MARIE* uses shared memory and sockets and *Carmen* is based only on *IPC*. *MRDS* and *ROS* support both synchronous and asynchronous communication. *MRDS* uses *Decentralized Software Services Protocol (DSSP)* and *Hypertext Transfer Protocol (HTTP)* as the foundation for interacting with services. *DSSP* is a *SOAP*-based protocol that provides a symmetric state transfer application model with support for state manipulation and an event model driven by state changes. *ROS* supports synchronous communication via services, asynchronous communication via topics, structured messages using a specific *Interface description language (IDL)*. Both *ROS* and *MRDS* support well documented, broad specter communication mechanisms using well known protocols like *Transmission Control Protocol (TCP)* and *HTTP*.

2.6.3 Usage

Table 2.3 summaries the *Usage* group. It is composed of *Deployment and life-cycle (DLC)*, *Programming model (PM)*, *Code and data integration services (CDIS)* and *Extension points and interfaces (EPI)*.

Deployment and life-cycle (DLC)

Neither of the middlewares provide a multi-robot deployment system. *ROS* doesn't have any repository based deployment system but includes a CMake based compilation chain called catkin. It uses the Gazeboo simulator for testing environment. *Miro* has an *IDL* compiler, which helps to generate all the code for the communication and underlying middleware service. It uses Stage and Gazeboo for simulation and testing purposes. *MRDS* uses Visual

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Middleware	*DLC*	*PM*	*CDIS*	*EPI*
Player/Stage	~	~	✚ component relocation at run-time	✚ *API* access
ROS	✚ Catkin, Gazeboo	✚ Asynchronous and synchronous programming model	✚ roslaunch, rosrun	✚ large number of often-demanded services like gmapping, etc
Miro	✚ *IDL* compiler, Gazeboo)	✖ *CORBA*	~	✚
MRDS	✚ Visual Studio	✚ *C#*	✚ *VPL*	✚
MARIE	✖	~	✚ wraps Player and *Carmen*	✚
Orca	✚ CMake	✚ Multiple programming languages	~	✚
Carmen	~	~	~	✚
Pyro	✚ Gazeboo	~	✚ wraps Player	✚

Table 2.3 – Usage

Studio as IDE which provides a compilation chain, deployment tool as well as a simulator for testing. *Orca* uses CMake compilation and includes a graphical simulator for testing. *Pyro* includes multiple simulators for code testing: Stage, Gazeboo and Khepera but it doesn't have a deployment tool. There is no need of compilation chain since the code is interpreted. It needs a properly installed and set-up runtime stack. *Carmen* provides configuration tools, a simulator, and graphical displays and editors, but no deployment and compilation chain tools. *Player/Stage* doesn't have a native compilation chain, but there exist third-party compilation chains included in IDEs to compile the application source code. It provides testing environment in Stage simulator. *MARIE* has no specific compilation tools or deployment system. In general, none of the selected middlewares provide a complete tool-set for managing both the deployment and the life-cycle of a robotic application.

Programming model (PM)

ROS supports both synchronous and asynchronous programming models. The applications can be written in *Python* and *C++* natively but there is integration for *Java*, *Lisp* and other lan-

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

guages. It has the highest grade due to the variety of the programming languages and models. *Player/Stage* application can be written in any programming language. *MARIE* supports both a set of foundation behavior classes and finite state automate. *Carmen* application can be written only in *C*. *Pyro* is *Python* based. *Miro* application can be written in any languages that provide *CORBA* implementations. The data exchanges are event-triggered. *MRDS* uses the *VPL*, a graphical development environment that uses a service and an activity catalog. The main programming language in *MRDS* is *C#*. *Orca* supports essential *C*/*C++* on Linux, but it can be used using *Java*, *Python*, and *C#*.

Code and data integration services (CDIS)

All middlewares support modular architecture and allow easy integration or reuse of code. *Miro* provides service abstractions for sensors and actuators by means of the *CORBA IDL*. *Orca* maximizes the software reuse and modularity in robotic applications while *Carmen* has a large number of libraries. The portability of devices in *Player/Stage* allows manual component relocation at run-time and it is easy to integrate new features in the existing code as well as new modules. *MRDS*, with the use of *VPL*, allows to generate the code of new "macro" services from diagrams created by users. They can interact graphically, a service or an activity is represented by a block that has inputs and outputs that just need to be dragged from the catalog to the diagram. Linking can be done with the mouse, it allows the users to define if signals are simultaneous or not, permits to perform operations on transmitted values. *MARIE* provides translation facilities such that components written for *Carmen* or *Player/Stage* can be used. *Pyro* supports modules created for *Player/Stage*. *ROS* has well designed package and launch system capable of launching the dependencies.

Extension points and interfaces (EPI)

Most of the frameworks provide often-used robotic services and *APIs*. *Player/Stage* has a large extendability due to interfaces for different robot devices and services. It has a direct *API* access to often-needed robotic services. *ROS* has also a large number of often-demanded services. Both *Player/Stage* and *ROS* has a large number of *APIs* reported at the other middlewares. *Miro*, *Carmen* and *MARIE* provide *APIs* for modules specific to robotics. *Pyro* offers integrated *APIs* and interfaces while *Orca* allows the user to define custom interfaces and communication protocols.

2.7 Conclusion

The advantages of a robotic fleet are the information sharing, the robustness to failure and the parallelization of tasks that reduce the time needed to accomplish them. If a robot fails during a task, the task can be reassigned to another fleet member. New distributed software infrastructures are proposed to assist the progress of robotic fleets. The concept of *cloud* for

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

robots is emerging, allowing them to communicate with external *cloud* infrastructure [Tenorth et al., 2012] and deport heavy computing operations as well as allowing them to interact with the Internet of things [KnowRob, 2014]. A robotic cloud is mostly formed of robots, communicating objects and other hardware infrastructure elements that share information and resources in a transparent way for the developer. The down-side of the existing infrastructure for this new concept is the communication infrastructure that supposes: a centralized WiFi access. This reduces the use cases of robotic fleets that may also be used in uncontrolled environments without a broad communication infrastructure.

Based on the tables above, *ROS* [ROS, 2014] is one of the most suitable robot middleware that can be applied to a fleet of robots, followed by *MRDS* [MRDS, 2012]. Both of them are fulfilling totally or partially almost all the criteria. In our opinion, *ROS* [ROS, 2014] is the emerging middleware with the most potential to become the most used framework for robotic fleets [Chitic et al., 2015]. It still needs work since it has no multi-robot coordination system and no automated testing environment, but it has already the advantage of having a large community that develops new packages for it. Another key element of *ROS* is its communication mechanism. It supports both synchronous and asynchronous communications and can easily be customized with new message types. It has a large database of drivers making a very good abstraction of the hardware layer. New modules and packages can be developed and integrated quickly. It is very permissive for the developers allowing them to code in different programming languages. *ROS* is used later in this thesis as a robot middleware that allows the communication between the new applications developed with our contribution and the real hardware.

All these middlewares propose a modular vision of software components. One of the many practices in developing these components is to start with a model design. Multiple models can be composed in order to create software services in a *SOA*. The next chapter presents different types of formalism that can be used to design models for system behavior.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

3 Formalisms to design systems behavior

This chapter presents the Model Driven Development (MDD) approach and a summary of existing formalism used to design (robotic) software components. It focuses on timed automata formalism that will be later used in this thesis for the tool-chain that we propose to be used in order to apply MDD *over* SOA *in a fleet context.*

3.1 Introduction 33

3.2 Model driven development 34

3.3 Classical formalism 38

3.4 Timed automata 40

3.5 Conclusion 49

In this chapter, we give some background knowledge on formalism to design system behavior. We first begin with a brief description of an approach to design (robotic) software based on models, called *Model driven development (MDD)*. Then we present some of the existing formalism that have already been used in developing robotic applications. Later, we focus on timed automata, a formalism that will later be used in the tool-chain that we propose to be used in order to apply *MDD* over *SOA* in a fleet context.

3.1 Introduction

As the robotic applications move towards a fleet context, there is an increase in the need of having software architectures and systems that can perform better in terms of scalability, fault-tolerance, manageability and maintainability as well as understandability. In this context,

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

multi-robot applications can be developed from behavioral models in a *SOA*. One of the paradigm that could be considered in the process of designing model based behaviors for robots is Model Driven Development.

When using such paradigms, choosing the right formalism to model the applications is difficult. Since robots evolve in the physical world, the time should be an important factor in the process of modelling new software architectures. In this case, the model will be bound to the development of the application, thus to the real-world.

3.2 Model driven development

As seen in the previous chapter, *SOA* can increase the modularity of a robotic software while reducing the complexity of development. But the complexity of developing by dividing a bigger application into services still remains. As mentioned in [Bruyninckx et al., 2013], even in the robotics domain, the attitude of software developers is to produce code faster and better in their favorite programming language than via the "detour" of formal models even if in academia models are the starting point of robotic software components [Brugali and Scandurra, 2009]. In this section the reader will be presented with a process of development based on models called *Model driven development (MDD)*.

In other engineering fields, the use of models is motivated by the design of complex systems. The abstraction given by a model helps to understand the solution of a complex problem. The same concept can be applied to software development which deals with the same high level of complexity, especially in robotic applications for fleet context. This paradigm has already been applied with success in context heterogeneous environments like self-aware pervasive systems [Gerbert-Gaillard and Lalanda, 2016]. However, this technique is not a common practice in software engineering. Furthermore, the use of models usually ends in the conception phase of a new application.

MDD is a process to conceive new software using models not just as a starting point but also to develop the corresponding software. The main feature of this technique relies on the automation of the development process.

The main goal behind *MDD* is to increase productivity and to reduce the costs of debugging and testing complex software. As mentioned in [Atkinson and Kühne, 2003], two kinds of impact are visible:

- *Short-term results* because the process minimizes the time of developing new components and maximize the correctness between the model and the final software product.
- *Long term results* because the process maximizes maintainability of the code that is strongly bound to the model, thus the obsoletely of components. Furthermore, it is minimizing the software sensitivity to change which can be:

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

- *Personal* - Software development is tide to the persons developing them. Without a proper documentation, there is a risk that only the original creator can maintain it. Using *MDD* reduces this risk since models are described using a concise notation.
- *Requirements* - Changing requirements during the development phase of a software usually does not imply changing the conceptual model. In *MDD*, this forces the change of the model that will be reflected in the software.
- *Development platform* - Software components are usually tied to the development tools used. But the development platforms are evolving, thus making the components obsolete. *MDD* can help the decouple of software from the development platform allowing interoperability [Kleppe et al., 2003].
- *Deployment platform* - The deployment platform needs to be as transparent as possible for new software artefacts. The constant evolving of deployment environments can render a software obsolete after a short life-time. Using the abstraction in *MDD* allows this cross-platform capacity by using custom mappings [Varró et al., 2002].

3.2.1 Model driven development characteristics

In *MDD*, a model is characterized as a formal meta-model capable of joining the specifications of a given application domain and the syntactic links between these. An infrastructure offering *MDD* support must specify [Bézivin et al., 2003]:

1. How models can be created and how they are allowed to interact and be used.
2. What are the notations in used.
3. What is the connection between the model and the real world/environment.
4. How models can be extended.
5. How models can be shared.
6. How the models can be mapped form other software components.

Up to now, various techniques have been used with success to respond to these specifications. Visual programming has proven successful as a method to create models and the links between them (Spec. 1-3). Even-more, *OOP* [1] has proven its benefits in supporting extensible languages by allowing the extensions of types, *objects* (Spec. 4). Furthermore, meta description techniques have proven efficiency in dynamically extensible run time environments (Spec. 5-6).

[1] *OOP* is a programming paradigm based on the concept of *objects*, which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Summing up a Visual programming applied over an object-oriented language with meta description support can create the right infrastructure for *MDD*. But this is not enough. In order to be efficient, a *MDD* process should satisfy the following properties [Selic, 2003]:

- *abstraction* - the models should remove unnecessary details and create a layer of abstraction in order to form a better way of understanding the system and its interconnections.
- *understandability* - after abstraction, the model should be represented in a comprehensive way. The understandably is bound tightly with the expressiveness of the model. Ideally, a model should minimize the effort required to understand it.
- *predictiveness* - the model should be used to predict and validate the system behavior via experimentation (e.g. executing a software on a robot and analyzing the behavior) or formal analysis (e.g. analyzing using model checking the properties of a model).
- *inexpensiveness* - the costs to model and use *MDD* should be less than normal development (without *MDD*).
- *accuracy* - the capacity of a modelled system to mimic as close as possible the real life system it is modelling.

3.2.2 Model driven development paradigm

MDD is based on the meta-modelling paradigm which has been originally used in *Model driven engineering (MDE)* domain. Its main purpose was to improve the automatic code generation from abstract domain specific models. This paradigm aims to start with a higher level of abstraction in order to end with a detailed specification of the application domain, to go “from platform-independent to platform-specific” [Ringert et al., 2015].

The standardization of *MDD* and *MDE* is mainly conducted by *Object Management Group (OMG)* [OMG, 2016] which is using the term *Model Driven Architecture (MDA)* for those processes. *OMG* defines four levels of model abstraction (shown in figure 3.1) traditionally used in *MDD* meta-model paradigm.

This form of defining the model abstraction consists of hierarchical levels. Each level represents an instance of the upper layer. The lower level, M0 represents the real-world system. In our case, this layer stores the actual software components the application needs to use at run-time. The upper level, M1, stores a model representation of the software components. It is refereed as the Model level because it holds the user models. The next level, M2, holds the model of information used by the M1 layer and it is called the Meta model level because it holds the information model of a model. The top layer, M3, is the higher level of abstraction and it holds the model of information used in M2, thus it is named Meta Meta Model or, historically, Meta Object Facility.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

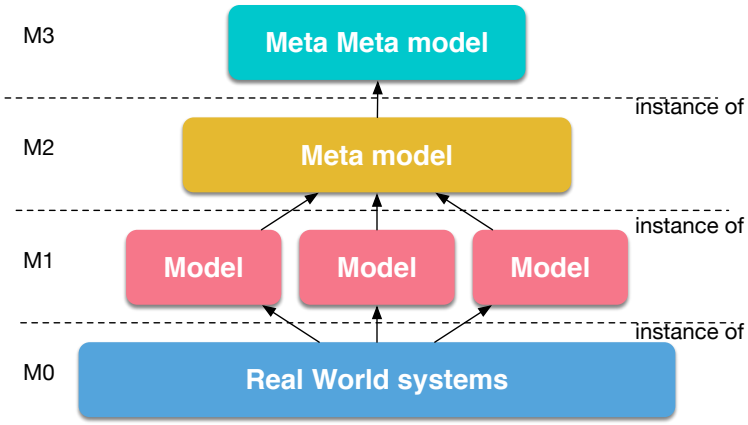


Figure 3.1 – Object Management Group modelling

The model abstraction is used in building tool kits capable of automatically generation of the software components used in M0 based on the information in M3, M2 and M1. Most of the well-defined technologies used in compilers can be applied in model-based automatic code generators [Jouault et al., 2008].

3.2.3 Model driven development in robotics

As mentioned in [Ramaswamy et al., 2014] and to the best of our knowledge, *MDD* has been applied in robotics in frameworks like: RobotML [Dhouib et al., 2012], *V3CMM* [Alonso et al., 2010], SmartSoft [Schlegel et al., 2009b] and BRICS model [Bruyninckx et al., 2013]. A brief description of each of these frameworks can be found in Appendix B.

It is worth mentioning that BRICS applies the separation in layers of the model composition, RobotML uses a *Domain Specific Language (DSL)* in order to define the models while V3CMM and SmartSoft allows for code skeleton generation from the model. None of the frameworks offer a integrated model checker, thus the models cannot be validated against any properties.

3.2.4 Conclusion

The general experience with *MDD* has shown that it can improve the process of development by minimizing the costs and maximizing productivity. The focus on *MDD* is oriented on the automatically code generated based on models. But there is a great gap between *MDD* and a software that is formally proven to be correct [Bert et al., 2005].

In the next sections, the reader may find a brief overview of some of the formalism used to model and analyze systems and how they are applied in robotics.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

3.3 Classical formalism

In this section, we present some of the existing formalism that can be used to define the behavior of a robotic application when using a *MDD* approach. We present briefly each formalism and its applicability in computer science, in robotics and in industry.

All the formalism presented in this section use or are based on *Finite-state machine s (FSMs)*. Next sections present a detailed background on a particular extension of *FSM*, called timed automata.

3.3.1 Finite state machine

A *Finite-state machine (FSM)* is a mathematical formalism used to design both software and sequential logic circuits [Gill et al., 1962]. It is defined by the its (finite) number of states, its triggering conditions (or inputs) and its transitions. The particularity for *FSM* is that the machine can be in just one state at any given time, called current state.

FSMs are present in many automated devices that surround us [Gajski et al., 1994]. They can be found in elevators, vending machines, traffic lights, washing machines, etc. But their applications are beyond hardware logical circuits. Mostly, *FSMs* are used in computer software. They have been used to design programming languages and models [Berry and Gonthier, 1992], compilers [Corbett et al., 2000], networked system [Hershey et al., 1995], software testing environments and methods [Chow, 1978], etc. Furthermore, *FSM* are at the base of Turing machines [Shannon, 1957].

FSM is a classical formalism used in robotics. The applications include modelling autonomous navigation [Sales et al., 2010], path planning [Choset, 2001], mission planning and control [Pirjanian et al., 2000], defining the entire robotic behavior based on *FSM* [Martinoli et al., 2004, Bautin et al., 2012].

Section 3.4 will present a detailed background on a particular extension of FSM, called timed automata.

3.3.2 Petri nets

Introduced by Carl Adam Petri in 1962, *Petri nets* are a powerful graphical and mathematical tool that can be used to represent complex sequential mechanisms and phenomena [Petri, 1962]. It allows to model complex processes by supporting synchronization and path choice. It has been used to model and analyze discrete event systems [Murata, 1989], [David and Alla, 2010].

Petri nets have been used in several domains. In the manufacturing system, they have been used to model and analyze production lines including automated assembly lines, resource

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

sharing systems and Kanban productions[2] lines. Another domain where *Petri nets* were successfully used is to model sequence controller on Programmable Logic Controllers allowing a decrease in the development time compared to the traditional approach. Related works in this domain can be found in: [Murata et al., 1986], [Crockett et al., 1987] or [Jafari, 1992].

Petri nets have also been used in software developments. In order to model and analyze software system using *Petri nets* [Reisig, 1986], an extension of them, called *Colored Petri nets* has been introduced [Jensen, 1989]. Integrated software development system [Pinci and Shapiro, 1990] allows for the automatic conversion of graphical *Petri nets* into executable code. Other interesting works can be found in: [McLendon Jr and Vidale, 1992] or [Murata et al., 1989].

In robotics, *Petri nets* have been successfully used to model flexible manufacturing systems [Beck and Krogh, 1986], [Kodate et al., 1987]. It was also used to model Sensory-Based robots [Lyons and Arbib, 1989] as well as unmanned vehicles [Jaulin et al., 2012]. As mentioned in [Freedman, 1991], *Petri nets* supports a convenient mechanism to express a complex robotic behavior. Lately, *Petri Nets* were used in a number of frameworks and architectures for modelling both single and multi-robot plans [King et al., 2003] [Costelha and Lima, 2007], [Kotb et al., 2007], [Ziparo et al., 2011]. The various extensions of *Petri Nets* used in robotics include *Colored Petri nets* [Marciano, 2013], *Timed Petri nets* [Zuberek, 2001] or *Self-Modifying nets* [Rust and Rammig, 2004].

3.3.3 Markov decision process

Markov decision process (MDP) is a formalism framework that supplies the mathematical tools to model decision making process where the result can be partially based on the decision as well as partially random. As presented by [Bellman, 1957], a *MDP* represents a discrete time stochastic control mechanism that satisfies the *Markov property*[3]. Over time, multiple extensions for *MDP* emerged. These include *Partially observable Markov decision process (POMDP)* [Spaan, 2012], *Constrained Markov decision processes (CMDP)* [Altman, 1999], Continuous-time Markov Decision Process [Guo and Hernández-Lerma, 2009].

MDP has a large applicability. In industry, the applications include the modelling water reservoirs [Lamond and Boukhtouta, 2002], design and maintenance support for traffic systems [Robelin and Madanat, 2007, Zhang and Gao, 2012], etc. *MDP* was also used in finance to model stock markets in order to maximize investors profit [Schäl, 2002]. In (tele)-communications, *MDP* has been used to model the management of traffic in core networks [Altman, 2000] as well as in wireless communications [Djonin and Krishnamurthy, 2007].

One of the largest application of *MDP* is in computer science. It has been used to design

[2] Kanban is a scheduling mechanism for lean manufacturing (a management philosophy derived mostly from the Toyota) and just-in-time manufacturing (methodology aimed primarily at reducing flow times within production, also derived mostly from the Toyota

[3] The decision in the current state is conditionally independent of all the previous states and actions.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

algorithms for dynamic programming [Lovejoy, 1991, Puterman, 2014]. In artificial intelligence and machine learning, *MDP* contributed to reinforcement learning [Kaelbling et al., 1998], learning automata [Barto and Anandan, 1985], etc. Game theory is another area of applicability of *MDP* [Liggett and Lippman, 1969].

In robotics, *MDP* has been used for planning and control of robotic navigation [Christensen and Pirjanian, 1997], in the process of planning the robotic missions [Theocharous et al., 2001] as well as in unmanned vehicles [Bagnell and Schneider, 2001]. *MDP* has also been used in the decision making process of robots [Mihaylova et al., 2002]. Furthermore, it has been used to design entire new robotic architectures [Koenig and Simmons, 1998], etc.

3.3.4 Process algebras

Process algebras represent a family of approaches for formally modelling concurrent systems. As defined in [Baeten, 2005], the term *process algebras* refer to the behavior of a system defined using an algebraic approach [Birkhoff and MacLane, 1948]. In this context, a behavior represents the composition of all the events and actions that a system can perform. *Process algebras* represents a high-level formalism used to model the interactions, communications, and synchronizations between a set of independent processes or agents [Hermanns et al., 2002]. They also define methods that allow the manipulation of process description and offer a mechanism for analyzing the equivalence between processes via bisimulation[4] [Bergstra and Klop, 1986].

In software development, *Process algebras* has been used at the core of frameworks used in the design of communication protocols and distributed systems like *Construction and Analysis of Distributed Processes(CADP)* [Garavel et al., 2013]. It has also been used in tools for analyzing system behavior like mCRL2 [Cranen et al., 2013] and in various software applications like web services [Ferrara, 2004], etc.

Process algebras usage in robotics includes specifications and planning of robotic missions [Karaman et al., 2009], distributed control architecture for robotics [Petersson et al., 2001] and definition of robotic behavior [Košecká et al., 1997]

3.4 Timed automata

In this section, we present some detailed background information on timed automata, a formalism that has been used in the process of modelling and validating real-time applications as well as in robotic applications. This formalism is later used in this thesis for a new programming methodology designed for multi-robot applications. The reader will first be presented with an overview of the model. Then we present some of the different classes of timed automata focusing on a particular class related to this work before reviewing different

[4] A bisimulation represents a binary relation between state transition systems

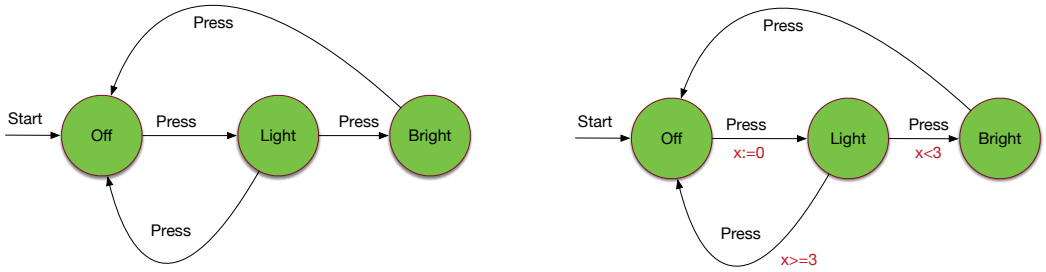
Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

model checking software.

In the later part of this thesis, timed automata will be very useful in the conception phase of robotic applications using our new development methodology. It is used as the base of the tool-set to design new robotic behaviors. Our work depends on the closure under intersection used in the composition of behaviors models of components in order to analyse the reachability properties of applications running on the entire fleet.

3.4.1 Overview

Defined as an extensions of classical finite state automata [Hopcroft, 1979], a timed (finite automata) was introduce by [Alur and Dill, 1994] to model of real-time systems. It provides a simple and powerful annotations of states-transitions timed constrained graphs by using real-valued clocks [Alur, 1999]. In order to better understand the need of adding time to a finite state automata, [Alur, 2004] presents the following problem: "A simple light controller with one button needs to be modelled. When the button is pressed two times with a delay less than 3 seconds, the light becomes brighter. If the delay is greater than 3 seconds, the light turns off."



(a) Finite state automata (b) Timed automata

Figure 3.2 – Simple light controller model

Figure 3.2a presents the light controller modelled as a finite state automata. The reader should notice that the automata is non-deterministic because, when in state $light$, after the press of the button, the transition $press$ can bright the system into both $bright$ or off states. This problem is solved in fig. 3.2b by the introduction of clock x, which transforms the non-deterministic automaton (fig. 3.2a) into a deterministic timed automaton.

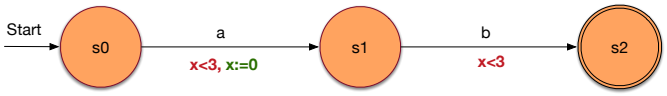


Figure 3.3 – Example of a simple timed automata A

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Definition and properties

The real-life example of the light controller is abstracted into the example of a simple timed automata A in fig. 3.3. The automata A can be considered as a finite state automata because it presents three locations (s_0, s_1 ,s_2) of which s_0 is the initial state and s_2 is an accepting state. The automata A has also two possible transitions over the alphabet $\sum = \{a, b\}$, thus A can recognized only $a \cdot b$ as an accepting word.

What differentiate the sample automata A from a finite state automata, is the presence of timing conditions over a clock x which is a continuous variable over the set of real-valued numbers $\mathbb{R}_{\geq 0}$. A is recognizing the timed word $a \cdot b$ only if the transition from s_0 towards s_1 is done in less than three time units and the transition from s_1 towards s_2 is also done is less than three time units. Initially, the clock x is set to 0, evolving synchronously as time advances, in the state s_0 and it is reset to 0 when the automaton switches from state s_0 to s_1. A timed automaton can have an elastic number of clocks and any transition can reset an arbitrary number of clocks. The time constrains that validate or invalidate labeled transitions are called guards. A guard allows or not a transition to be executed depending on the result of the boolean function represented by the guard. In the automata A, the clock x is used in the guard of both transitions so that a or b cannot be recognized after more than 3 times units elapsed in state s_0 or s_1.

Time always progresses. In the case of the timed automata, time evolves in the states, while the transitions are instantaneous. In order to ensure time progress, it is possible to bound the time elapsed in a state by defining time constrains inside a state called *invariants*. An *invariant* can be used to force a transition to be triggered before its constraint becomes violated. This ensures that the time can always progress. Furthermore, the progress of time is always non-negative and this is also visible in the *timed words* recognized by any timed automata. A *timed word* is a sequence of tuples formed by a non-negative real value attached to a symbol. For example, the A automaton can recognize the timed word $w = (a, 1) \cdot (b, 2)$ where b was recognized after 1 unit of time after a. In general, a timed word over an alphabet $\sum$ is a sequence $(a_0, t_0) \cdot (a_1, t_1) \cdots (a_n, t_n)$ such that $a_i \in \sum$, $t_i \in \mathbb{R}_{\geq 0}$ and $t_0 < t_1 < \cdots < t_n$.

In the following definition and explications, the notation used are the same as in [Alur and Dill, 1994, Alur, 1999].

Definition 1 (Timed Automata) *[Alur and Dill, 1994] A timed automaton A is a tuple* $A = (\sum L, L^0, L^f, X, I, E)$ *where:*

- $\sum$ *is the* alphabet,
- L *is a finite set of* states *(or locations),*
- $L^0 \subseteq L$ *is the set of* initial states,
- $L^f \subseteq L$ *is the set of* final (accepting) states,

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

- X *is a finite set of clocks,*
- $I : L \rightarrow C_{\prec}(X)$ *the function that associates an* invariant *to each state*
- $E \subseteq L \times C(X) \times \sum \times 2^X \times L$ *is a finite set of transitions where* $e = (l, g, a, r, l') \in E$ *is a transition from state l to l', where g is the guard, r is the set of clock to be reset and a is the label.*

The timed words recognized by a timed automata A is represent by $(a_0, t_0) \cdot (a_1, t_1) \cdots (a_n, t_n)$ where $\forall i \in 1, 2, \cdots, n, a_i \in \sum$ is a symbol of the alphabet and $t_i \in \mathbb{R}_{\geq 0}$ is the time when a_i was recognized. $L(A)$ represents the timed language of the timed automata A and is the set of all timed words recognized by A.

Let X be the set of clocks for a timed automata A having its values in $\mathbb{R}_{\geq 0}$. A clocks valuation v for X is a function $X \rightarrow \mathbb{R}_{\geq 0}$ which maps each clock $x \in X$ with the value $v(x)$. $\mathbb{R}^X_{\geq 0}$ denotes the notation for the set of clocks valuations for X. $C(X)$ represents the set of clocks constrains over X and it is formed using an arbitrary number of combinations of atomic expressions $x \# c$ where $x \in X$, $\# \in \{<, \leq, =, \neq, \geq, >\}$ and $c \in \mathbb{Q}$. The set of the clocks constraints $\in C(X)$ of the form $x < c$ or $x \leq c$ is noted $C_{\prec}(X)$.

A clocks valuation v fulfils an atomic expression $x \# c$ if and only if $v(x) \# c$ evaluates positive. Using this way, a complete constraint g, formed by an arbitrary combinations of atomic expressions, can be check if it is satisfied by a clocks valuation v. $v \models g$ represents the clock valuation v that satisfies g. A clocks valuation $v' = v + d$ implies that $v'(x) = v(x) + d \; \forall d \in \mathbb{R}_{\geq 0}$ and $\forall x \in X$. Furthermore, given a subset of clocks $r \subseteq X$, $v' = [r \leftarrow 0]v$ represents the clocks valuations that $v'(x) = 0 \; \forall x \in r$ and $v'(x) = v(x) \; \forall x \in X \setminus r$.

Being an extension of classical automata [Hopcroft, 1979], the union and the intersection of timed automata are also an extension of classical operators on generic automata. The closure under both the operands stands from the property of timed automata of being in-deterministic, supporting more than one location. [Alur and Dill, 1994] has proven that reachability analysis is decidable, yet PSPACE-Complete [Ponge, 2008].

Semantics

The semantics of a timed automata A is defined by associating an infinite timed *Labeled transition system (LTS)* with it. A state of the *LTS* is a pair $(l, v) \in L \times \mathbb{R}^X_{\geq 0}$ such that l is the current state in A and v is a clock valuation. The semantics of $A = (\sum L, L^0, L^f, X, I, E)$ is given by the *LTS* $S_A = (S, s_0, \rightarrow, \sum)$ where:

- $S = L \times \mathbb{R}^X_{\geq 0}$,
- $s_0 = (l_0, v_0)$ where $l_0 \in L^0$ and $v_0 = 0 \; \forall \; x \in X$,
- $\rightarrow$ is the transition,

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

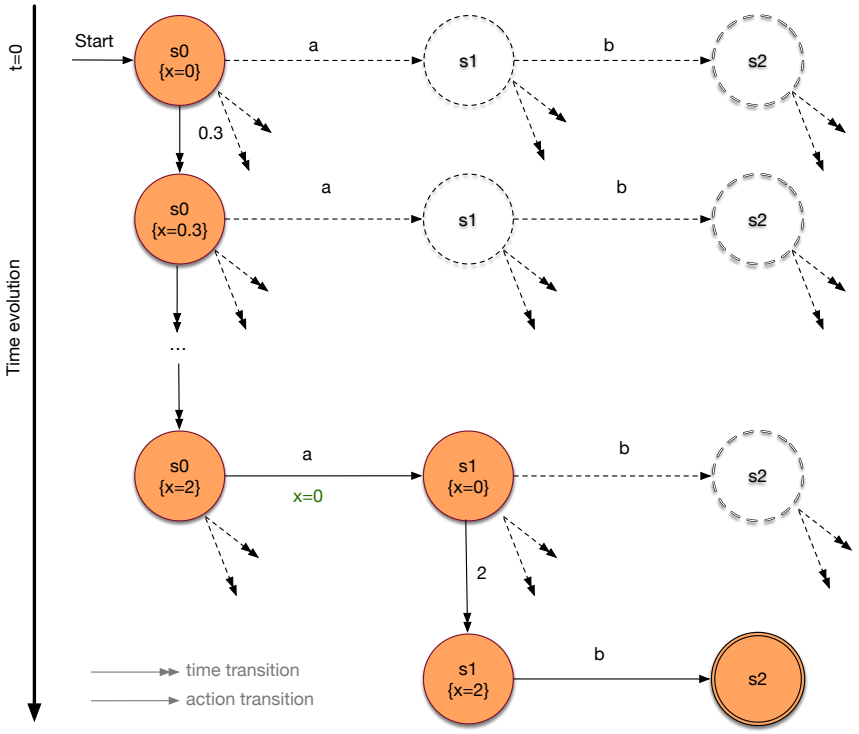


Figure 3.4 – The LTS associated with the timed automaton in fig. 3.3[5]

- $\sum$ is the alphabet of A.

There are two types of transition ins S_A:

- *action transition* - a state can change due to a location change
  $(l, v) \xrightarrow{a} (l', v') \Leftrightarrow \exists\, e = (l, g, a, r, l') \in E$ such that $v \models g$, $v' = [r \leftarrow 0]v$ and $v' \models I(l')$

- *time transition* - a state can change due to elapse of time
  $\forall\, d \in \mathbb{R}_{\geq 0}, (l, v) \xrightarrow{d} (l, v + d) \Leftrightarrow v + d \models I(l)$

Figure 3.4 presents the semantic *LTS* S_A of the timed automaton A presented in fig. 3.3. $(s_0, 0) \xrightarrow{0.3} (s_0, 0.3) \xrightarrow{0.1} (s_0, 0.4) \xrightarrow{1.6} (s_0, 2) \xrightarrow{a} (s_1, 0) \xrightarrow{2} (s_1, 2) \xrightarrow{b} (s_2, 2)$ is a valid execution of the S_A. The timed word recognized is $(a, 2) \cdot (b, 4)$ of the timed language $L(A)$. S_A starts from an initial state with each clock set to 0. With the time progress, either an action transitions changing the state of the automaton with the possibility of resetting a subset of clocks or time transitions allow the synchronous evolution of clocks values. A can recognize an infinite number of timed words resulted in the execution of the S_A from the initial states to final states.

[5] Based on figure 3.2 from [Ponge, 2008]

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

3.4.2 Classes of timed automata

Multiple extensions and classes of timed automata have been studied. In this subsection, we focus on the classes that were mostly related to modelling robotic behavior and is used in the later contributions. Interesting contribution are presented in [Tripakis, 2003], [Ouaknine and Worrell, 2004], [Alur and Madhusudan, 2004] or [Bouyer and Laroussinie, 2010].

Deterministic timed automata

Defined by [Alur and Dill, 1994], the class of *deterministic finite automata* narrows the definition of a timed automata because:

- It allows only for a single initial state.
- It imposed that if two transitions from the same state have the same input symbol, then the guards associated with this transitions need to be disjoint. In this case the determinism of the transition is maintained.

Both the automate in fig. 3.2b and fig. 3.3 are deterministic. In the case of fig. 3.2b, the reader should notice that, even the symbol is identical for the transitions from state “light”, the guards are disjoint. This class can be used in the design of behavior models for robotics because the result of a recognized word (a behavior in the robotic context) is the same given the same conditions, only the value of clocks changes.

Event-recording timed automata

Proposed by [Alur et al., 1999], event-recording timed automata is a particular class of timed automata, where each input symbol of the alphabet is associated with a clock. The specificity of this class stands from the fact that when a symbol is recognized, the corresponding clock is reset to 0. Even if the guards can be composed of several clocks, only the clock corresponding to the action transition can be reset.

This restriction of this class makes the values of clock tight to the input word recognized. Furthermore, an in-deterministic event-clock automaton can be translated into a deterministic automaton. This is not the case for any generic timed automata [Alur et al., 1999]. This class can be used to design behavior models for robotic fleet that react on signal (symbols) from various sensors, actuators and environmental surroundings.

Figure 3.5 presents a transformation of the example in fig. 3.2b form a general timed automaton into an event-recording timed automaton. The reader should notice that, since the alphabet only has one symbol (*Press*), only one clock is present (*xPress*). This clock is reset at every transition.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

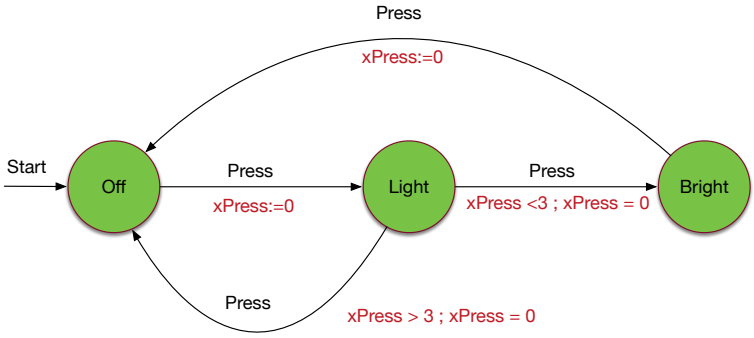


Figure 3.5 – An example of event timed automata corresponding with the timed automaton in fig. 3.2b

.

Other classes of timed automata

Some classes or extensions of time automata worth mentioning are:

- *Robust automata* - this class of timed automata allows time words to be recognised with a certain error measuring interval for the value of the clocks, which correspond better to real physical system. Their timed languages expressiveness cannot be compared with one timed language. [Alur and Madhusudan, 2004]

  Figure 3.6 presents a transformation of the example in fig. 3.2b form a general timed automaton into a robust timed automaton. The *dx* present in each state signifies the interval of error (e.g. for state *Off* is +/- 3 times units) for the clocks value when the transition(s) from this state is performed.

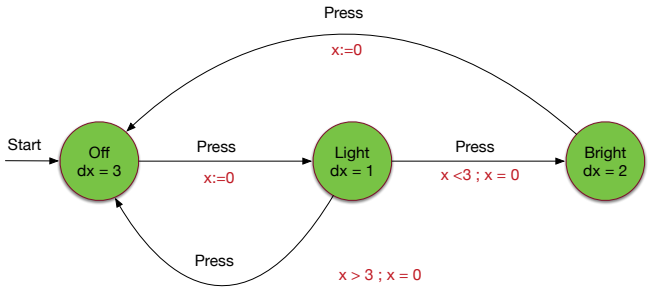


Figure 3.6 – An example of robust timed automata corresponding with the timed automaton in fig. 3.2b

.

- *Silent transitions* - Silent transitions correspond to the internal communications or internal states changes of a timed automata state. In timed automaton, silent transitions can be used to model discrete-time behaviors embedded in continuous time.

  Figure 3.7 presents a transformation of the example in fig. 3.2b by adding the silent transitions ϵ. The ϵ transition corresponds with an internal silent transition in the

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

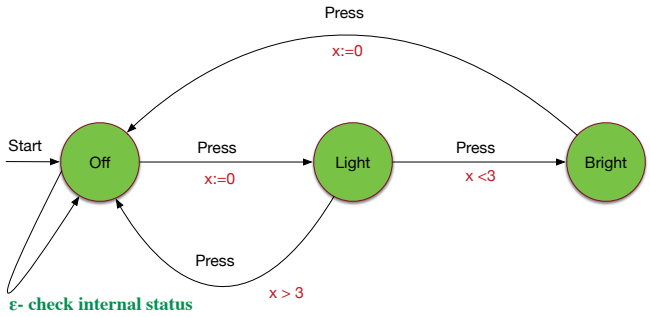


Figure 3.7 – An example of silent transition corresponding with the timed automaton in fig. 3.2b

.

state *Off*, where the module (in our case, the lamp) checks its internal components for dysfunctions.

3.4.3 Software tools

In order to verify that a (timed) automata model corresponds to the specifications of the system modelled, the notion of *model checking* has been introduced. It allows to test *properties* of the system against the *model* version of the system. A *Model checker*, represented in fig 3.8 as a black box, has as input the model of the system and the property to verify and outputs a boolean that shows if the property is satisfied and, optional, a trace of errors if the property is not satisfied.

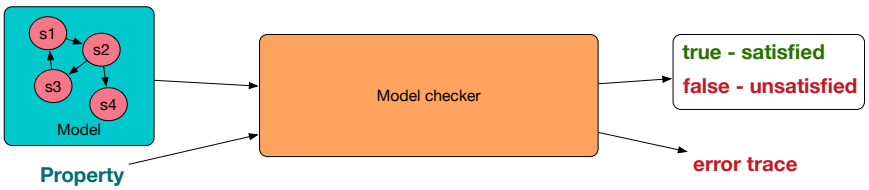


Figure 3.8 – Model checking principle

Even if a model checker dose not classify the properties to be verified, them can be regrouped into:

- *Reachability* properties specify if a property can possibly be satisfied by the model (e.g the light can be brighter).
- *Safety* properties specify that "bad" things will never happen in the model (e.g the light cannot stay off for more than 24h hours).
- *Liveness* properties specify that "good" things will eventually happen in the model (e.g pressing the button will trigger the light to turn on).

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Temporal logics

The properties used in the model checkers usually are written as form of *temporal logics*. Temporal logic focuses on the qualitative time properties rather than quantitative ones. Take the example in fig. 3.2b, a temporal logic can verify the succession of events (e.g. When the button is *pressed* and the light is *off*, the *light* will turn on). On the other hand, it cannot verify the quantity of events (e.g. When the button is *pressed* twice in less than 3 time units, the *light* will turn brighter). The main purposed of temporal logics is to verify if there exists a path between the state that will satisfy it. A detailed survey of timed temporal logics can be found in [Bouyer, 2009].

It exists two branches of temporal logics:

Linear-time temporal logics - that allows the verification of the formula over a single time line. The most common temporal logic in this category is *Linear temporal logic (LTL)* [Pnueli, 1977]. In the base form, it supports only qualitative time properties. In order to extend these properties for quantitative time, [Koymans, 1990] proposes *Metric temporal logic (MTL)* (with its extension *Metric interval temporal logic (MITL)* [Alur et al., 1996], *Safety metric temporal logic (Safety-MTL)* [Ouaknine, 2007] and *Flat metric temporal logic (Flat-MTL)* [Ouaknine and Worrell, 2005]) and [Alur and Henzinger, 1994] proposes *Timed propositional temporal logic (TPTL)*.

Branching-time temporal logics - that allows the verification of the formula over several branching time line. The most common temporal logic in this category is *Computational tree logic (CTL)* [Clarke et al., 1986]. One extension of *CTL* work mentioning is *Time computational tree logic (TCTL)* [Henzinger et al., 1994].

Model checkers

It exists a large number of model checkers. They differ from the classes of timed automata used as models as well as from the branch and type of temporal logics used for expressing the properties and query the model. We briefly introduce the main tools and focus on the *UPAAL*, which is later used in the contributions.

Most of the tools are using branching timed temporal logics due to the decidability of the model checking. *Kronos* [Bozga et al., 1998] is s a model checker that support analysis of a multiple communicating timed automata. It is one of the few model checkers that uses generic timed automata. *Tempo* [Sorea, 2001] is a model checker for event-recording timed automata. *Timed COSPAN* [Hardin et al., 1996] was developed at Bell Labs and uses an approximation of continuous semantics as one of its heuristics. *HyTech* [Henzinger et al., 1997] is a model checker that uses a hybrid automata [6] [Alur et al., 1997] as model and an extension of *TCTL* as temporal logics called *ICTL*.

[6] An hybrid automata includes both continuous (e.g., variables in $\mathbb{R}$) and discrete behavior (e.g., variables in $\mathbb{N}$)

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

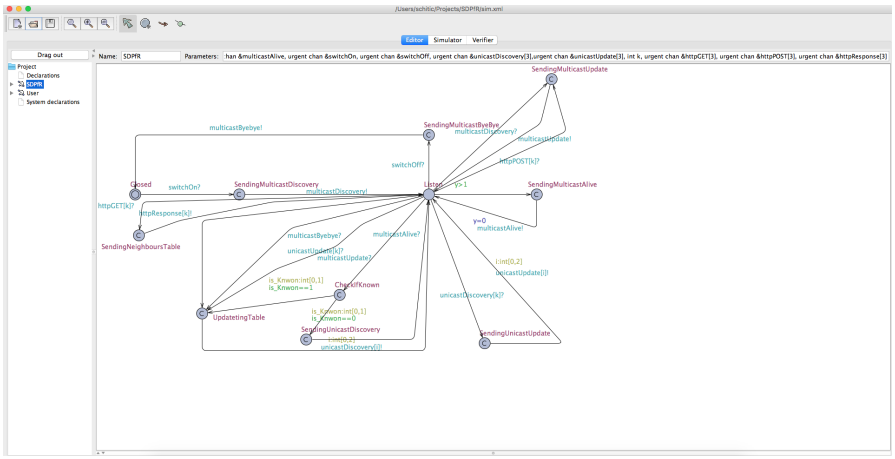


Figure 3.9 – UPPALL screen-shot

UPPAAL is a model designer and checker tool emerged from an academic research prototype to a commercial product. *UPPAAL*, as for *HyTech*, uses an hybrid extension of timed automata as a model and *TCTL* as a query language to express properties. It can be used to describe systems that can be modelled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables. It used typically for application that timing aspects are critical like real-time controllers, communication protocols (fig. 3.9 shows the timed automata of a discovery protocol, part of our contributions, designed in *UPPALL*). It has been used in several industrial studies[7] like : [Iversen et al., 2000], [Lindahl et al., 2001], [David et al., 2003], [Hessel and Pettersson, 2004] or [Ravn et al., 2011].

As for our knowledge, *UPPAAL* is the only mature model checker project that has extensible *API* and can be integrated as external library with other projects. This key aspect was considered when choosing *UPPALL* as the model checker for our contributions.

3.5 Conclusion

In this section, we have presented a development approach to design and develop new software components called *MDD*. It was presented a series of classical formalism to design software behavior and their applications in robotics. We have focused on a particular formalism called timed automata.

As show above, models have been used as a starting point into developing robotic software and architectures. We believe *MDD* can also be applied in robotics, to allow an automated translation from models to software components.

This chapter ends the first part of this thesis related to the state of the art in (robotic) middle-

[7] (A detailed list can be found at http://www.it.uu.se/research/group/darts/uppaal/examples.shtml).

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

ware and formalism and methodology to design (robotic) system behavior. We have focused on *ROS*, which became a largely accepted middleware for robotics, on *MDD* approach to develop software components and on timed automata formalism. Robotic applications are tight to real-time systems. It is clear that time is an important component in modelling robotic software. Furthermore, we have chosen timed automata as formalism applied to *MDD* because it allows time to be considered in the modelling phase of a robotic application and it provides a powerful mathematical tool-set for model checking.

In the next part of the thesis, we present our contributions. We begin with a protocol for service discovery in robotics modelled using timed automata. This protocol will be integrated with a tool-set designed to develop *ROS* based applications using a model based programming methodology, presented in the following chapters. This programming model applied the concepts of *SOA* and *MDD*.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Part II

Model driven multi-robot applications development

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

4 Service discovery for robots

This chapter presents a protocol for service and neighbors discovery, called Service Discovery for Robots, in the context of highly mobile fleet robots and evaluates several variants of its implementation.

4.1 Objectives and motivation for fleet service discovery 53

4.2 Limitation of existing service discovery protocols 54

4.3 Definition of SDfR protocol 56

4.4 Evaluation of SDfR overhead with robots 68

4.5 Summary 81

4.1 Objectives and motivation for fleet service discovery

In order to cooperate inside the fleet and be able to share data, the robots need to know with which peers they can exchange data, how to manage the communications and what are the services offered by their peers.

New applications that are operating in a multi-robot context are generating multiple layers of complexity into the robotic development. This chapter focuses on a central need of fleets of robots: how to allow them to be aware of connected neighbors and their services using the network interfaces. Combining component and service-oriented programming greatly simplifies the implementation of highly-adaptive, constantly-evolving applications [Frénot et al., 2010]. We think robots should advertise their functionality as services in order to allow other members of the fleet to interact with them. Furthermore, robotic fleets need an automated mechanism that allows for an ad-hoc network to be automatically provisioned.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

In order to solve the problem of neighbors and service discovery in an ad-hoc network, a robot needs a protocol that is able to constantly discover new robots in its coverage area, while maintaining a neighbor connectivity quality indicator. Since there is not any central node that can manage *IP* address allocation, the protocol should be able to negotiate an *IP* address inside the network and to have a conflict management tool in case of an *IP* collision.

In the robotic context, this chapter contribution proposes to adapt *Simple Service Discovery Protocol (SSDP)*, a well-known network discovery protocol in order to allow robots to discover their connected neighbors, their services and their capabilities in any IP based Wifi infrastructure. Discovery protocols are highly used nowadays in most of the connected devices. To take into account the mobility of robots, we change a series of fields in the messages headers as well as add a memory mechanism to limit consumed bandwidth. This proposal is validated using experimental benchmarks on multiple scenarios with a various number of Turtlebot 2.

The chapter is structured as follows: Section 4.2 discusses limitation of existing service discovery protocols if applied into robotics, Section 4.3 defines a proposal for service discovery in a fleet context, called *Service Discovery for Robots (SDfR)*. Section 4.4 evaluates the protocol via a series of benchmarks and Section 4.5 concludes the chapter.

4.2 Limitation of existing service discovery protocols

A way to see a fleet of robots is like a service-oriented multi agent system. Such environments like *Peer to peer (P2P)*, *Multi-Agent Systems (MAS)* or *Service-Oriented Environments (SOE)* tend to approach the problem of service discovery in a *centralized*, *distributed* or *decentralized* way:

- Centralized mechanisms like super-peers [Gummadi et al., 2002], middle-agents [Klusch et al., 2006] or central registries [Rompothong and Senivongse, 2003] are limited in number of peers in the system and in terms of number of requests. They also use a centralized node which can have serious impact if the central point becomes unreachable.
- Distributed approaches such as *Distributed Hash Tables (DHT)* [Maymounkov and Mazieres, 2002] offer more scalability and robustness by having multiple specific nodes that can manage the resources.
- Decentralized systems consider all the nodes to be equal. This approach provides more flexibility, but it has its downsides, since each node only has partial view of the entire system. As mentioned in [del Val et al., 2014], an interesting way to discover service inside a decentralized and self-organized multi-peer system is to use homogeneity between agents.

Another way is to apply classical protocols and middlewares for service discovery in distributed

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

environments like data-grids, clouds or even smart environments.

Universal Plug and Play (UPnP) [Jeronimo and Weast, 2003] represents a service discovery mechanism that enables network devices to advertise, discover and control their services. Initially developed by Microsoft, the software stack of the protocol is developed over the *IP* in order to facilitate the communication between peers by using a series of standardized protocols like *HTTP* for discovery, *XML* for description and *SOAP* for control of the services. The main purposes of *UPnP* are [Talal and Rachid, 2013]:

Address management *UPnP* manages the *IP* address allocation for peers by either requesting an address from a *Dynamic Host Configuration Protocol (DHCP)* server on the network or by assigning a random address to each client. The address conflict detection is then delegated to the client.

Service description In *UPnP*, each peer describes via a *XML* document. This documents contains the device related information (e.g. model, serial number, position, etc.) as well as a list of available services on the peer. Each service is described via a URL that is also included in the list.

Service control Based on the retrieved service description, the control manager can invoke remote services via control messages to a specific URL. This messages are sent via *SOAP* protocol.

Events management In *UPnP*, the peers can receive notification including update of services or status of other peers. A service that wants to publish an update will send an event messages formatted with *General Event Notification Architecture (GENA)* via a *XML* message.

Discovery The discovery protocol of *UPnP* is based on *SSDP* which allows *UPnP* ready device to advertise their presence and their services as well as to discover other peers' services. It uses a series of multicast messages [Jeronimo and Weast, 2003]. *SSDP* operates on the top of the existing open standard protocols, using *HTTP* and *User Datagram Protocol (UDP)*. The main disadvantage of SSDP is the absence of a attribute-based querying mechanism for services [Ververidis and Polyzos, 2008].

In a centralized infrastructure, all the robots can have a complete image of their neighbors and can use classical Service Discovery Protocol like *UPnP* [Ahn et al., 2005] that manages into a repository all the services published by other members of the fleet.

In the robotic world, an approach for service discovery in centralized networks could be to use classical *UPnP* protocol. Since the concept of having the robotic tasks and processes as services is not mature yet, the main focus on research on service discovery in robotics is oriented toward the integration with the environment where the robot is considered only as one device, part of the smart environment. In [Borja et al., 2013] provides a case study of

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

integration of service robots and smart-homes via *UPnP*. In these cases, the authors are not referring to a robot as part of a specific fleet, but as part of an environment, in which the robot is considered as an entity that can offer services. This point of view is slightly different in case of a robotic fleet [Ververidis and Polyzos, 2008], where robots are composed of multiple services that need to be discovered by the other members.

Decentralized systems (e.g *UPnP* [Ahn et al., 2005], Jini [Pereira et al., 2011] or *Service location protocol (SLP)* [Romero et al., 2010]) can be *a purely distributed* solution where each node stores its own service repositories or a *hybrid* solution that includes super-nodes that aggregate information from other peers.

The solutions presented above have their downsides if applied to ad-hoc multi-robot systems. Firstly, due to the mobility of the robots, the network connection is highly unstable. *UPnP* discovery protocol, *SSDP*, does not perform the same way in a highly mobile environment as in a static one due to the mobility of the robots. As mentioned in [Issarny et al., 2011], the challenge is to set the tradeoff between physical mobility and scalability. The discovery protocol should be ready to be used at any time and track its usage and failures. Secondly, existing protocols are not very adaptive in terms of same user connection/disconnection from the IP network. If a robot moves out of the communication area, *SSDP* protocol needs to wait unit a time-out is reached in order to remove the robot from the neighbors list. This may cause other robots to requests services that are out of their communication area, resulting in failures. Existing protocols like *SSDP*, do not remember already connected nodes, thus, when the connection is timed-outed, the discovery process is reinitialized. When the robots rejoin the network, the discovery process of its services is re-triggered in multicast, thus flooding the network with the same messages as the previous discovery step.

4.3 Definition of SDfR protocol

This contribution main goal is to propose a mechanism that allows highly mobile robots to keep track of the reachable peers inside a fleet while using an ad-hoc infrastructure. This mechanism is able to provide a list of services available on each peer. Another objective is to propose a network configuration negotiation protocol, because due to the mobility of robots, classical peer to peer network configuration techniques are not suitable.

This section presents the general description of a service discovery protocol for robotic applications, called *Service Discovery for Robots (SDfR)*. Based on this description, the evaluation section presents a comparison of different variations of *SDfR* by combining a series of binary and text-based protocols in the different layers of *SDfR*.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

4.3.1 SDfR as a derivate of SSDP

The contribution proposes a protocol that is not flooding the network and has an already seen memory feature build-in. *SDfR* protocol is a highly dynamic, adaptive and scalable protocol adapted from *SSDP* that is being used in *UPnP*. The main advantage of deriving from *SSDP* is represented by the possibility of interconnections with already deployed devices and architectures. *SDfR* can be also used to provide service discovery with the smart-environment in which the robots are being deployed. Having an identical messages exchanges diagram (shown in fig. 4.1), the interoperability between *SDfR* and *SSDP* ready devices is supported.

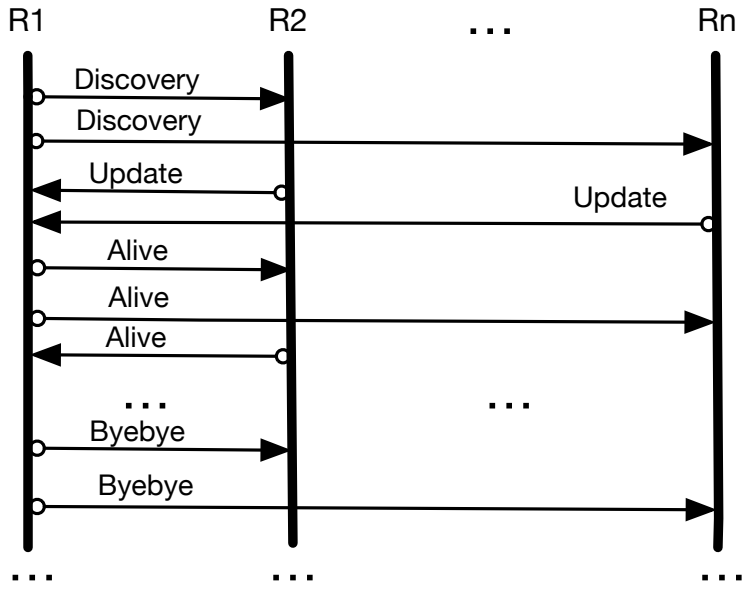


Figure 4.1 – SSDP and SDfR protocol timed diagram.

In order to limit the network use for the service discovery process, *SDfR* is sending most of the internal messages in multicast[1], avoiding the overhead generated by unicast[2] transmission in order to propagate the same message. In addition, in order to avoid failure in case of a disconnection due to the movement of the robots outside the coverage area, all the communications are done using *UDP*. Furthermore, to limit the network flooding when the protocol needs information from just one robot, a second transmission is enabled in unicast [2] mode. *SDfR* does not need to reinitialize the entire discovery protocol when the connection is lost, because it disposes of a history map of all the already seen robots and their services.

In order to avoid services that are out of reach (e.g. service of robots that are present in the history map but are not present in the covered communication area), a connection indicator is computed for each robot represented by the success rate of pinging the connected peers.

[1] In networking, multicast refers to a mechanism to address the same information simultaneously to a group of nodes.

[2] In networking, unicast refers to a mechanism to address the information to a single node.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

As shown in fig. 4.2, *SSDP* and *SDfR* are based on a similar automaton. Figure 4.2a shows the main automaton that uses only multicast transmissions, that are being reused in fig. 4.2b in order to maintain the retro-compatibility with *SSDP*. However, the *SSDP* native automaton has its downsides when applied to a highly mobile ad-hoc robotic environment because it uses a request-response model and it only sends multicast messages. This mechanism can generate a significant overhead on the network, making it unreliable for other robotic usages.

Both protocols propose:

Multicast transmissions In order to avoid the overhead of re-transmitting the same unicast message, most of the internal messages are multicast and uses the same timed diagram as *SSDP* (fig. fig. 4.1).

HTTP-style messages The messages that are being sent use an HTTP-style structure composed of headers and a body.

In Section 4.3.3, different protocol versions in the *SDfR* protocol layer are proposed. The text version protocol is represented by the actual HTTP message equivalent to the SSDP message. The binary version protocol represents an encapsulation of the HTTP message inside a compressed binary stream.

Differences between SSDP and SDfR

The main differences between *SSDP* and *SDfR* are:

Limited multicast transmissions To avoid failure in case of a disconnection due to the movement of the robots outside the coverage area, all the communications in *SDfR* are done using *UDP* and only in request mode.

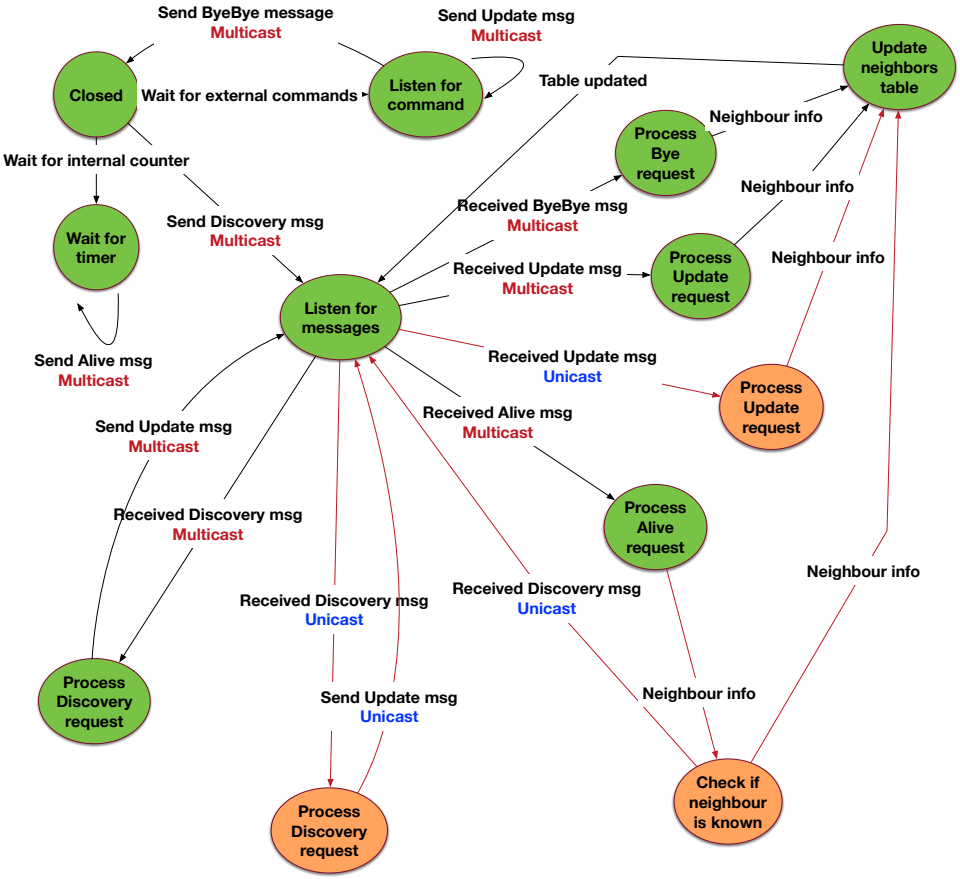
Unicast transmissions *SDfR* add to the *SSDP* protocol a new mechanism (see fig.4.2b) that sends only unicast messages. To limit the network flooding when the protocol needs information from just one robot (e.g. a "Alive messages" arrived from a robot that has just entered the communication area of a peer), a second transmission mechanism is enabled in unicast mode in *SDfR*.

History map *SDfR* does not need to reinitialize the entire discovery protocol when the connection is lost, because it disposes of a history map of all the already seen robots and their services. In order to avoid services that are out of reach (e.g service of robots that are present in the history map but are not present in the covered communication area), a connection indicator is computed for each robot. This feature also plays an important role in deciding in the type of transmissions used when a robot reenters the communication area of its peers. fig. 4.2b shows a state where the protocol checks in the history map if the robot has already been seen and decides if the message is sent in unicast or multicast.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés



(a) SSDP automaton



(b) SDfR automaton.

Figure 4.2 – SSDP and SDfR differences (represented in orange)

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

4.3.2 Protocol model and description

The protocol is designed as a finite state timed automaton. *SDfR* protocol behavior is defined by the request method. Each method has at least one type of message that reside inside the request payload. Two methods representing the desired action of a request are used in *SDfR*: *M-SEARCH* and *NOTIFY*.

The *M-SEARCH* method is used for discovery requests to get the list of nearby members and their services. The only message type associated with this method is *Discovery*.

The other method, *NOTIFY* is used to respond to a *Discovery* request or to inform the others about changes in the current state of the robot. The message types associated with this method are: *Update*, *Alive* and *Byebye*.

The *Update* message is sent as a response to a *Discovery* request or when the current services or capacities of the robot change.

The *Alive* message is sent recurrently, as a beacon, in order to inform the others about the presence of the robot. The rate to send the beacon can be set depending on the services need. The default value is at 10s.

The *Byebye* message is sent when the robot stops gracefully, in order to inform the others about its disappearance. Fi Figure 4.3 presents a *SDfR* specific view of the timed diagram in fig. 4.1. It shows the state changes of the automaton in fig. 4.2 and the message chains that triggers the state changes.

When the protocol initializes, a discovery multicast message ((1) of fig. 4.3) is sent, and then the protocol changes state into *listen* on a multicast as well as on an unicast socket. When the other robots receive a discovery message, they will respond with an update message ((2) of fig. 4.3).

When the protocol receives an update message, it passes into an atomic state, *Updating neighbors table* and updates their neighbors table. The protocol sends periodic alive messages ((3) of fig. 4.3). When the protocol receives an alive message ((3) of fig. 4.3), it passes into *Check if known* state, that determines if the unicast IP of the sender is already known. If so, it changes into *Updating neighbors table*, otherwise it will send a unicast discovery message ((4) of fig. 4.3). The sender of the alive message responds by sending an unicast update message ((5) of fig. 4.3).

If the protocol traps a graceful shutdown, a byebye message ((6) of fig. 4.3) is sent and the other robots will update their neighbor table.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

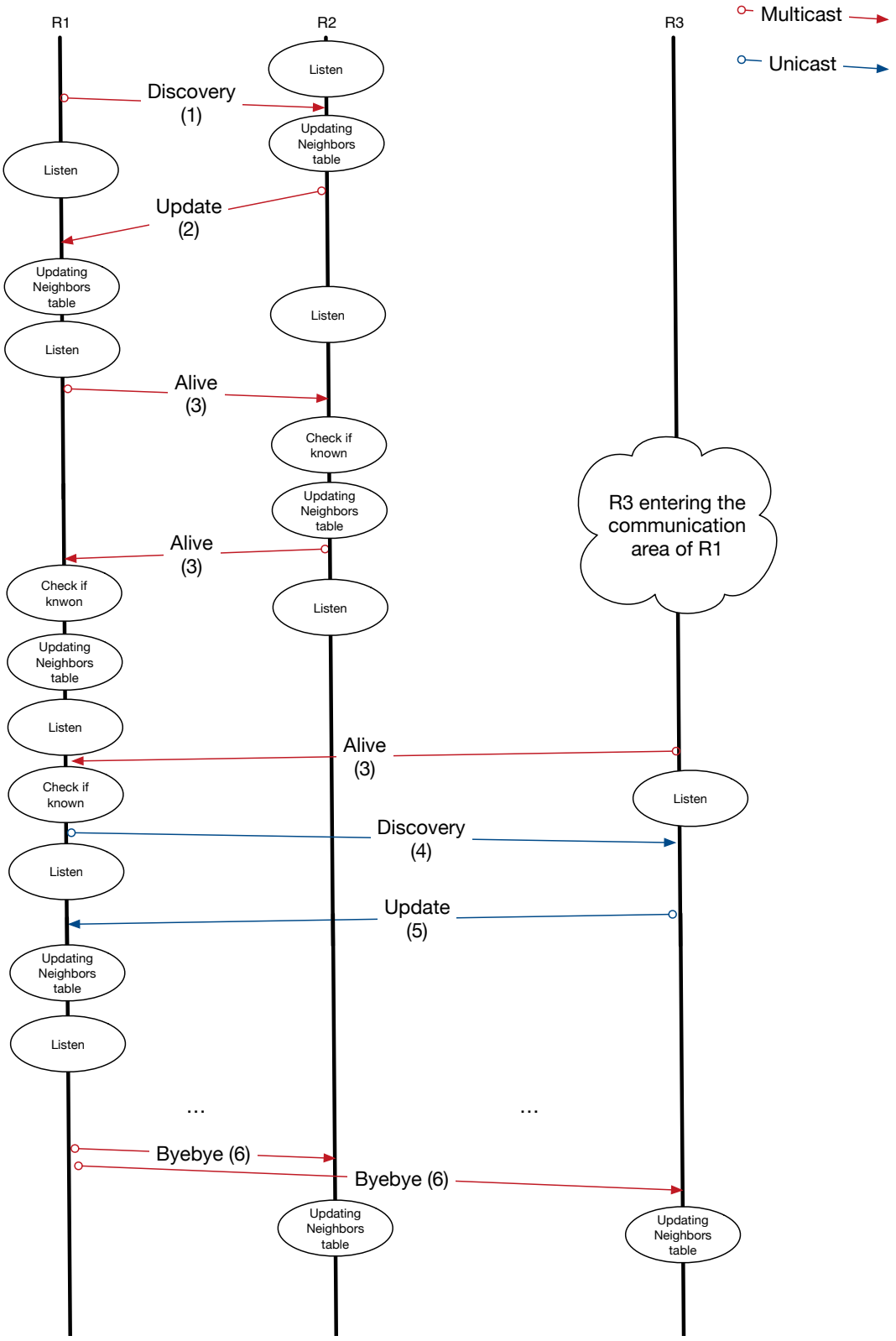


Figure 4.3 – SDfR protocol timed diagram.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Messages and headers

To better understand the dynamic of the protocol, the following subsection focuses on the messages and their headers, and on how they differ from *UPnP* in order to be adapted for multi-robot systems.

The *SDfR* version that uses plain-text communication protocol is compatible with *UPnP* because it uses the structures of SSDP, the service discovery protocol used in *UPnP*. This makes *SDfR* inter-operable with any smart environment. In Fig 4.4, the fields inside a *SDfR* message header is displayed.

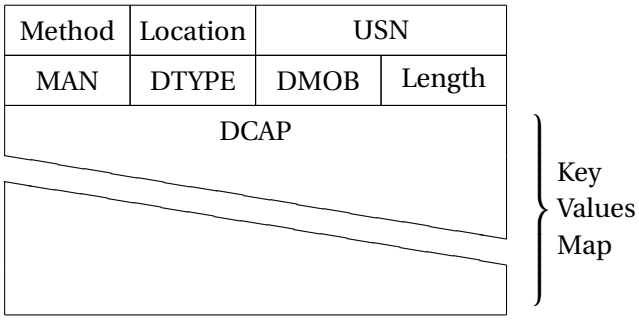


Figure 4.4 – SDfR common header.

A full description of the *SDfR* header can be found below (fields with a $^+$ are new):

Location - Location of the device. This field was present in *UPnP* and was kept for retro-compatibility with this protocol.

USN - Unique Service Name. The field was present already in *UPnP* and reused by *SDfR*.

MAN - Message Type. The field was present already in *UPnP* and reused by *SDfR*.

DTYPE$^+$ - Device Type. This represents the type of hardware platform. (e.g. Turtlebot2, PR2, etc).

DMOB$^+$ - Device Mobility. This new field was added in order to characterise the mobility of a robot. (e.g. Mobile, Temporary Mobile, Static, etc.).

Content Length - The length of the message content without the header. For transmissions without any payload, the field is set to 0.

DCAP$^+$ - Device Capacities. It represents a dictionary of keys and values that characterizes the state of the robot. It can include static information like the CPU frequency or the memory capacity, as well as dynamic information like the percentage of battery, the CPU usage ratio, etc. This data is included in every header of the *SDfR* because the information sent is highly-dynamic.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

All the messages share the same header information, only the payload of the message differs from one message type to another. There are messages without payload like *Byebye* or *Discovery*.

The main difference between *SDfR* and *SSDP* in the message headers is the addition of 3 new fields: *DTYPE*, *DMOB* and *DCAP*. Another difference is the location field that is always marked by a '*'. In *SSDP*, the location was used to physically pin point the device like 'kitchen-fridge', but in a fleet context it is hardly the case to have a fix physical location. Furthermore, the USN from *SSDP*, which represents the unique name of the service, is replaced with the unicast IP address of the robot. All the *SDfR* messages are being sent in multicast, but the robot needs the unicast address in order to use the information given by the protocol.

Bellow are two examples of SDfR message. First a discovery message is presented.

```
M-SEARCH
Location:*
USN:10.1.124.134
MAN:ssdp:discovery
DTYPE:Turtlebot2
DCAP:CPU=2.0Ghz|RAM=4Gb|BAT=59%
DMOB:Mobile,
ContentLength:0
```

The following message is the update message send in response to the previous message by another robot. In this example, the reader can notice that the *DCAP* filed specifies that capabilities of the robot are 2.0Ghz, with GB of ram and the battery level is at 98%. In the same header, it is specified the that the robot is a Turtulebot2, in the *DTYPE* field. Furthermore, in the payload of the message, each service is described by its name as well as a compulsory auto-description *URL*. In this example, the *URL* is *10.1.101.94:8042/auto_description*

```
NOTIFY
Location:*
USN:10.1.101.94
MAN:ssdp:update
DTYPE:Turtlebot2
DCAP:CPU=2.0Ghz|RAM=4Gb|BAT=98%
DMOB:Mobile,
ContentLength:247

{"Services":
  [
    {
      "Name":"P2P Monitoring",
      "URL":"10.1.101.94:8042/auto_description",
```


Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

```
      "Uuid":"3FA2F711-E142-4572-9AF0"
      "Metadata":{
          "status" : "ok",
          "alerts" : "0",
      }
    }
  ]
}
```

4.3.3 Implementation

Service Discovery for Robots is developed as a service itself. The service-oriented architecture approach for robotic software development is not very wildly used in the robotic community. The practice in this community is to develop built-in libraries in order to extend software features.

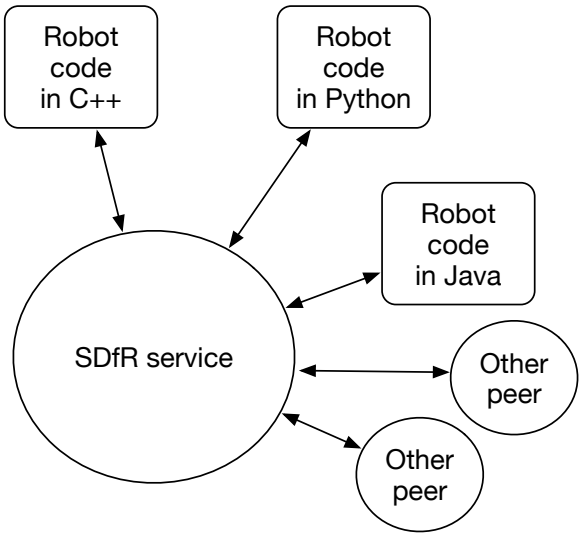


Figure 4.5 – Service oriented architecture in SDfR.

The main advantage of having a service-oriented architecture, as shown in Figure 4.5, is the compatibility with other robotic services developed in different programming languages and running over different operating systems. This is a critical feature for a heterogeneous robotic fleet.

Furthermore, *SDfR* service can run separately of the other processes on the robot and all the messages are consumed by instances of the service on multiple robots. If it fails, it would not affect the other services running on the robot. This sand-boxing also ensures that the information sent by the protocol is not corrupted by any other third-parties.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

SDfR service is composed of two layers as shown in Figure 4.6: an API layer that communicates with other services and a Discovery Protocol layer. Each layer has an independent life-cycle and communicates internally via a shared memory. The *API* layer responds to requests independently from the lower layer, using the information from the shared memory. The lower discovery protocol layer is in charge of communication with the other *SDfR* nodes on an elastic number of robots in order to discover the reachable peers and their services.

The *SDfR* service is implemented in the 'Go' programming language [Pike, 2012] version 1.3.3. Go provides concurrent abstractions and safe memory management, something lacking in C/C++ and to a certain degree from Python. 'Go' can build all-in-one package that does not have any dependencies since the binary offers static linking for them. Considering all the dependencies, the executable has still a small size in memory. Furthermore, 'Go' allows the built of cross-platform executable which is an important aspect in deploying *SDfR* service across a heterogeneous platform of robots.

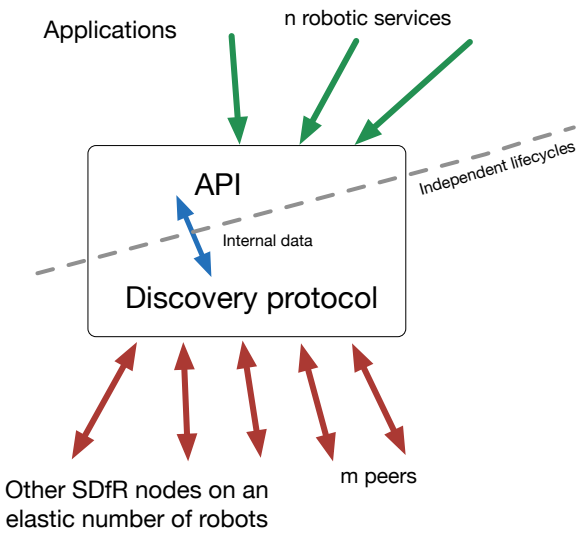


Figure 4.6 – SDfR service architecture.

RESTful communication API

A *Representational State Transfer (REST)* [Fielding, 2000] web-service was chosen for the API that lets other services to communicate with *SDfR* service because it is based on normal *HTTP* requests which is completely stateless. A full description of each web-service provided by *SDfR* can be found in Table 4.1. All the responses are *JSON*[3] messages.

When a producer wants to register to *SDfR* Service, it sends a POST request to the *API*. One of the parameters that needs to be included is an auto-description URL which is used for other peer services to negotiate the use of this producer. When the request gets processed by the *API*, *SDfR* will perform a GET request to check if the auto-description URL is working. Only

[3]JSON is a data format used for asynchronous communication. It is an open-standard human-readable format formed by attribute-value pair objects. A legacy alternative is represented by XML.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

URL	*Method*	*Description*
`/me/services`	GET	Returns the registered service list
`/me/services`	POST	Registers or updates a service in the local service list
`/me/services/<uuid>`	DELETE	Deletes a local service from the list
`/me/capacities`	GET	Returns the capacities of the robot
`/me/capacities`	POST	Adds or updates a capacity
`/me/capacities/<uuid>`	DELETE	Deletes a capacity
`/neighbors`	GET	Returns a full list of neighbors and full description of their services
`/search/capacities/?<value>=[><]<key>`	GET	Filters the list of neighbors for capacities in the URL query. > < can be used for comparison and | for regular expression
`/search/services/<name>/?<value>=[><]<key>`	GET	Filters the list of neighbors for services with the specified name and metadata filters from the URL query. > < can be used for comparison and | for regular expression

Table 4.1 – RESTful API for SDfR protocol.

if this is working, the producer will become register into *SDfR*. When another service wants to retrieve information about the services of the neighbors, it sends a GET request to one of the *REST APIs*. A *JSON* message that represents the list of neighbors is generated. Since the communication between the upper *REST* layer and *SDfR* protocol layer is done via a shared memory, responses are generated immediately, without having to wait for an internal *SDfR* protocol transmission.

Ad-hoc configuration management

Since the fleet is operating in an ad-hoc infrastructure, the peers need to be able to negotiate and auto-configure their network configuration. A robotic fleet ad-hoc network is different from a classic ad-hoc hot-spot because the robots can move, thus rapidly change position, and the network can be partitioned or merged. The mobility of the peers needs to be taken into consideration in the negotiation protocol of the configuration. *SDfR* service, based on a simple configuration file, is able to automatically connect to an ad-hoc network. The secured WiFi network is composed using the fleet id. This mechanism allows to have multiple fleets of robots in the same networked space. Moreover, the robots can auto assign IP addresses. The standard network space is *10.<fleet id>.<x>.<y>*, where x and y are computed by each robot from its internal MAC address in order to avoid IP conflict [Thomson et al., 2007]. Furthermore, if an IP conflict happens, the service has a mechanism to trigger an IP change on the robots. This mechanism is available all the time since an IP conflict can be triggered by a merge of 2

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

sub networks.

ROS integration

In order to make *SDfR* service user friendly, a ROS node that communicates with *SDfR* service and can be used by other nodes via topics and services was created. When the node starts, it launches a instance of *SDfR* if it is not running and then it provides support for other *ROS* nodes to publish or unpublish their services and their capacities. Furthermore, the *ROS* node provides the neighbors list of services and is capable of allowing other ROS applications to search for a specific service with a specific configuration.

A producer node can publish its services or capacities in an asynchronous way using *ROS* topics because the registration is not highly bound to time. The same concept applies for the unpublish commands and for getting the list of neighbors. On the other hand, the search command for a specific neighbor and their services needs to be done in synchronous way using *ROS* services because the behaviors of the consumer node is depending on it.

SDfR alternative implementations

Several variants of *SDfR* service were implemented in order to better observe their overhead by using different protocols in both the *API* layer as well as the internal *SDfR* protocol layer.

SDfR base variant is derived from *SSDP* and is designed to maintain a retro-compatibility with *UPnP*. This is why in *SDfR* base variant the messages between peer instances are created using a HTTP/1.1 like message in plain-text format. *SDfR* base variant service is piloted using a *REST* web-service in order to standardize the control *API*.

To compare the base variant, different alteration of the protocols used in both the *API* layer and internal *SDfR* protocol layer of the service by switching from text-plain encoding into a binary encoding were chosen.

***API* layer** Since the *API* layer was designed as a *REST* web-service using HTTP/1.1 request, it is consider encoding the same *REST* requests /responses in HTTP/2.0. As mentioned in [Grigorik, 2013], “HTTP/2.0 makes applications faster, secured, and more robust by enabling efficient multiplexing and low-latency delivery over a single connection”. This allowed to have a compressed binary channel between the clients and the *API*. Furthermore, the API is secured since HTTP/2.0 provides a native TLS encryption. Besides HTTP/2.0, *CoAP* is also used because it offers a *REST* like communication scheme over UDP and was designed for nodes with low-computation power [Shelby et al., 2014].

***SDfR* protocol layer** A variant for *SDfR* is to encode the transmission into binary by using Protocol buffers because this offers a reduced overhead when sending as binary com-

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

pressed variant a large number of object types [Google,]. Another way is to include a widely used protocol in The Internet of things, MqTT, a highly used Wireless Sensor Network protocol [Thangavel et al., 2014]. It is being used for the lower layer of *SDfR* Service as publish/subscribe protocol.

The different variants of *SDfR*-base protocol for experiments are:

SDfR This is the base variant of the service. The *API* layer is using a HTTP/1.1 *REST* web-service and the lower internal layer is using SSDP like HTTP/1.1 plain-text messages. The retro-compatibility with *UPnP* is maintained.

***SDfR*-binary** - This variant keeps the HTTP/1.1 *REST* web-service but has a compressed internal communication layer using the binary encoding of the plain-text messages with Protocol Buffers. The retro-compatibility with *UPnP* is not maintained.

***SDfR*-Http2** - This variant encodes the *REST* web-service in HTTP/2.0 and keeps the SDfR communication in a pain-text protocol. The retro-compatibility with *UPnP* is maintained.

***SDfR*-Http2-binary** - This variant combines the HTTP/2.0 encoding with the Protocol Buffers binary messages. The retro-compatibility with *UPnP* is not maintained.

***SDfR*-mqtt-coap** - This variant simulates the behavior of *SDfR* service by using *CoAP* as *API* protocol to pilot it and MqTT as publish / subscribe environment in the lower layer. The retro-compatibility with *UPnP* is not maintained. The main drawback of this variant is the use of a central node as MqTT broker.

4.4 Evaluation of SDfR overhead with robots

The evaluation aims to measure what is the *CPU*, memory and network overload generated by the use of *SDfR* in a robotic fleet context. Another objective is to see the impact of using text-plain protocol in the upper and the lower layer of the *SDfR* service. This evaluation includes different combination of text-base and binary protocols based on *SDfR* in order to compare important metrics in a multi-robot context. This section provides the bench-marking scenario and the evaluation results.

4.4.1 Experimental settings

The evaluation of the five variants of *SDfR* is performed in two types of contexts:

- a static scenario where the robots do not move to evaluate the overhead in an ideal WiFi communication scheme

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Nb Robots	*Pub/Sub ratio*	*Nb pub*	*Nb sub*
2	30%	60	140
2	50%	100	100
2	70%	140	60
4	30%	120	280
4	50%	200	200
4	70%	280	120
6	30%	180	420
6	50%	300	300
6	70%	420	180

Table 4.2 – Test-cases for static scenario.

- a real dynamic scenario where robots are moving and transmission can drop.

In both scenarios, all peers should discover their neighbors, but in the second one, the neighbors discovery depends on the distance between peers.

The benchmarks were performed on Turtlebot 2 robots equipped with an Intel Core 2 Duo, 2.1 GHz CPU, 4Gb of Ram PC, WiFi enabled (supporting Ad-Hoc networks) running on Ubuntu 13.04

Each test run was given 5 minutes to collect the data.

In the test runs, simulated services were used that try to register/subscribe into *SDfR*. Three type of actions were simulated:

1. **Publish.** New service providers try to *publish* via a POST to `/me/services/` with a delay time of 10 seconds. In order to simulate publishers, an Apache server was used on each robot that responds to the auto-discovery URL of each publisher.

2. **Unpublish.** Each of the already published service provider could be *unpublished* with a random delay between 5 seconds via a DELETE to `/me/services/<uuid>`.

3. **Subscribe.** Separated threads for each *consumer* that perform GET requests on `/neighbors/` were generated. Each thread constantly request the table of neighbors from *SDfR*, in order to stress at maximum the protocol.

In the static scenario, different numbers of robots (2, 4, 6) were considered. Each robot had a total number of service-providers and service-consumers equal to 100 simulated services. For each number of robots different ratios between providers and consumers were used: 30%, 50% and 70% publishers. Table 4.2 recalls the total number of providers and consumers per number of robots used.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

For each variant of *SDfR*, 6 robots with a number of 100 simulated services per robot were used in the dynamic test-case. 70% of the services on each robot were publishers. The robots moved randomly in a 200 square meters room with poles and other obstacles. The room (see Fig. 4.7) was exposed to WiFi interference from other networks that occupy all of the 2.4Ghz channels.



Figure 4.7 – Turtlebots in the experimentation hall.

4.4.2 Functional validation

To perform the bench-marking of the different variants, the various impacts that *SDfR* variants have on the system composed by the robots were considered. Firstly, it is important to consider the request time of a producer that advertises its service and a consumer that requests information about the services on nearby neighbors. Secondly, the impact on the machine on which the *SDfR* runs, especially the CPU and memory used was analyzed. Finally, keeping in mind that the protocol should not use a large bandwidth, the quantity of sent and received bytes was studied.

Latency

A robotic application that provides a service for the fleet needs to register with *SDfR*. This must be done as fast as possible in order to avoid blocking the service when it starts. Each provider needs to provide an auto-description URL that allows the consumers to negotiate the configuration in order to use the service. Since *SDfR* has to check the existence of this URL, the registration process is completed only after this step.

In the registration time evaluation, it was measured the time since a registration request has been made and the time when the response from *SDfR* has arrived. This includes the time of the auto-description URL check. On each instance of *SDfR* registration requests were

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

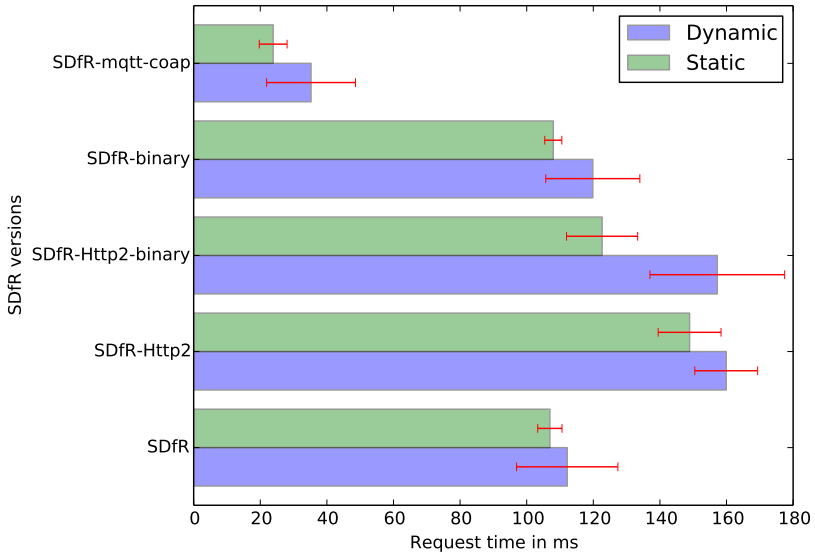


Figure 4.8 – Average request time for publishing a service.

simulated with a delay of 10 seconds per producer.

Figure 4.8 reveals the result of time consumed by a producer to publish its service in the static and dynamic scenarios.

In the static scenario, when using the *SDfR* and *SDfR*-binary variant, it is noticed a time of response for registration request ranging from 103ms to 109ms. This represents the time of performing HTTP/1.1 requests. In *SDfR*-Http2 and *SDfR*-Http2-binary it is observed a fairly higher time of response between 118ms and 159ms. This is explained by the time to encode the request and the response using the built in TLS3.0 encryption method from HTTP/2.0. The less time consuming is *SDfR*-mqtt-coap variant because it used a plain-text encoding over *UDP* in *CoAP* protocol. It is noticed that during all the scenarios this response time remains in the same variation interval regardless the number of robots used because the registration requests are concluded in local host.

In the dynamic scenario (Fig. 4.8) the latency is higher but the difference from the static scenario are less than 30%. It is noticed that *SDfR*-mqtt-coap is still the less time consuming, but with a higher standard deviation than in the static scenarios. The grouping of SDfR variants remain the same: *SDfR* and *SDfR*-binary that are using plain-text encoding have smaller response times than *SDfR*-Http2 and *SDfR*-Http2-binary. In general, all the response times are higher than the static scenarios and this can be explained by the increase in computation load on robots generated by the mobility of the fleet.

In both scenarios, it is notice that the HTTP/2.0 protocol variants have longer response times than the plain-text variants. Even if the requests are compressed in terms of data sent, since the request are performed on local-host, the time to encrypt the date is a downside for response times. The best variant from registration response time is *SDfR*-mqtt-coap by using the *CoAP*

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

protocol.

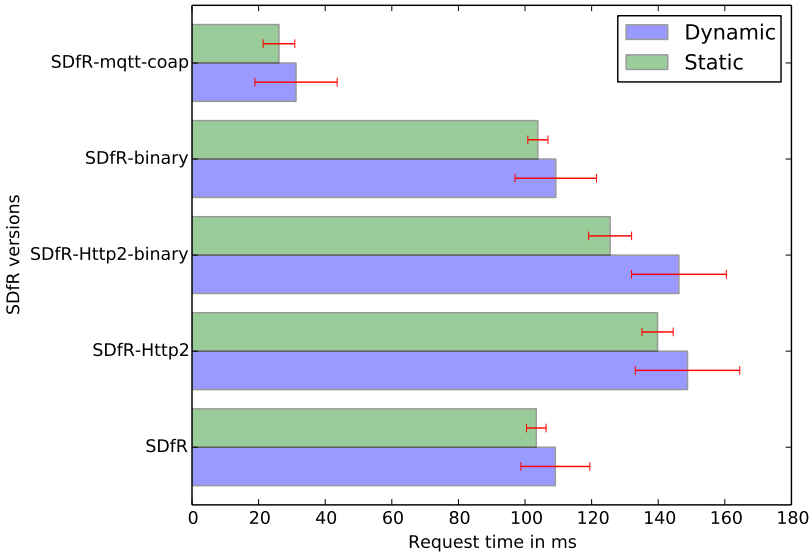


Figure 4.9 – Average request time for unpublishing a service.

Another important overhead measure is the time of unregister request. This happens whenever a producer wants to remove itself from the *SDfR* registry. This may occur when a robotic application gracefully ends or when it recovers after a failure and it needs to register to *SDfR*. It is an important metric since the unregister time may affect the run-cycle time of a stopping or restarting robotic node.

In the evaluation, in each test-run an unregister request for each register provider at a time interval of 5 seconds is performed. It is measured the time since the request is sent until the response is received.

Figure 4.9 shows the response time for unregistered request in static scenarios. As for the registration request, the time was in the same variation interval for all robots in all the scenarios. The reader can notice the same grouping due to the encoding techniques: *SDfR* and *SDfR*-binary with plain-text protocols and *SDfR*-Http2 and *SDfR*-Http2-binary with binary protocols at the API-level. Also in this case, *SDfR*-mqtt-coap has the lower response time due to *CoAP* protocol. In the dynamic case (Fig. 4.9) the reader can see an increase in the time for each variant tested as well as an increase in the standard variation interval.

As for the publish request time, the unpublished time is greater for compressed protocols as for plain-text ones. The same explanation applies here. Furthermore, it is noticed that the grouping of protocols in binary and plain-text variant is more pronounced in the dynamic scenario.

One of the most important metric for robotics application from a latency point of view is the time to request the list of neighbors and their services. In a real scenario, a producer registers once for its life-time cycle, but a consumer may request multiple times the list of reachable

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

neighbors and services. The subscription response time can have a impact on the time to complete a fleet task.

In order to stress the protocol, multiple threads based on the number of consumers per robot are generated. Each thread represents a consumer and all the consumers are parallelized. Each consumer requests continuously the neighbors list.

The averages values for the static and dynamic scenarios are presented in Fig. 4.10. For the static experiment, it is noticed a response time for subscription request between 4.5ms and 8.5ms for *SDfR* and *SDfR*-binary variants. The time for binary variants is 8ms and 12ms. The biggest time response was for *SDfR*-mqtt-coap at an average of 64ms. In the dynamic scenario (Fig. 4.10) the response times are under 14ms for all *SDfR* variant except *SDfR*-mqtt-coap.

Since the request from the consumer in case of HTTP/1.1 and HTTP/2.0 has a small header (having all the information needed in the URL of the request) it is normal to have a small response time in both static and dynamic cases. On the other hand, *CoAP* is using *UDP* in a connection-less state and acknowledging each request through a separate connection. It has a larger response time because requests get re-transmitted due to packet-loss in burst mode.

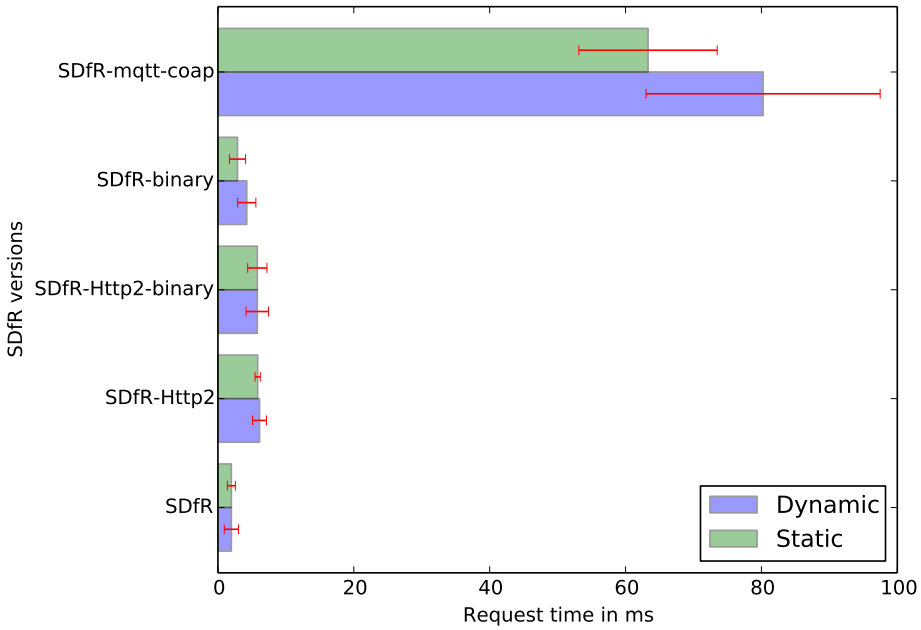
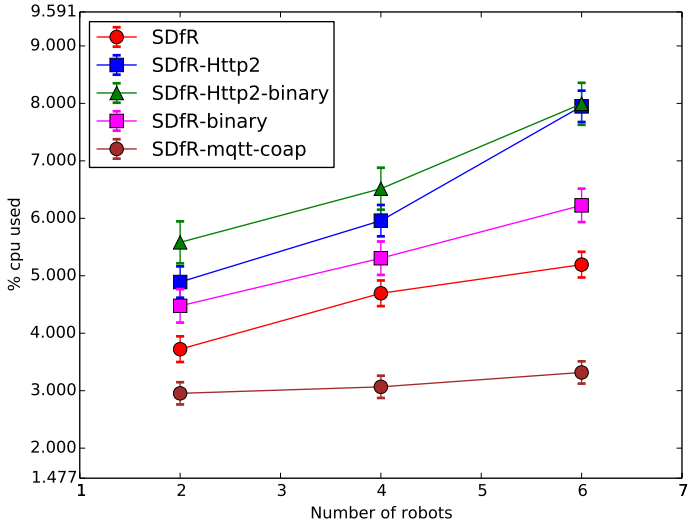
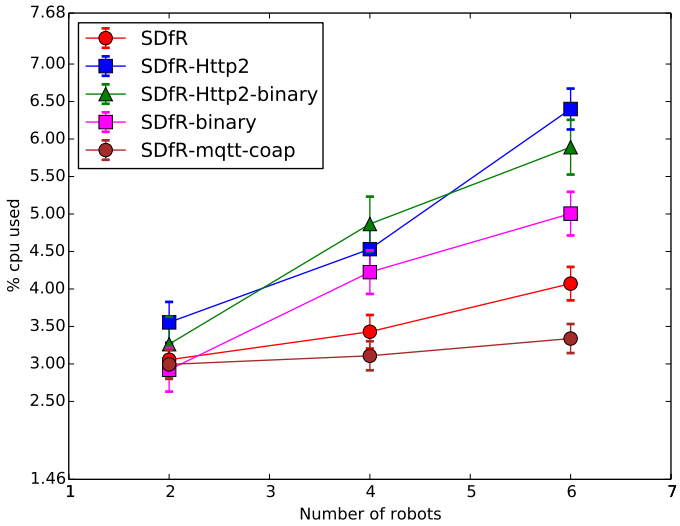


Figure 4.10 – Average request time for subscribing a consumer.

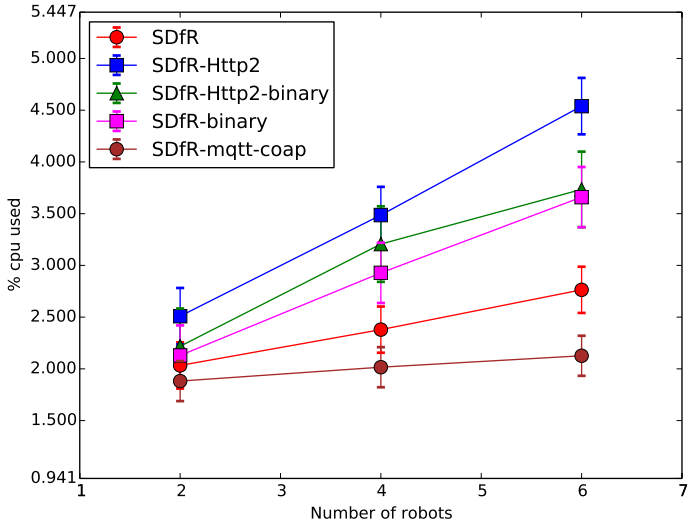
Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés



(a) Static with 30% publishers



(b) Static with 50% publishers



(c) Static with 70% publishers

Figure 4.11 – % of cpu usage.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Usage overhead

In robotic applications, the computation power is critical. Fleet of robots are heterogeneous and can include different types of robots with different computational factor. Having a low *CPU* consumption discovery service benefits the other processes involved in performing the fleet mission.

Figure 4.11 presents the results for % of *CPU* used during the bench-marking for each *SDfR* variant in the static scenario while varying the number of robots and the pub/sub ratio. *CPU* consumption varies between 1.7% to 8.4% of CPU usage. It is observed that the *CPU* usage evolves linearly to the number of robots. Another interesting fact is that the *CPU* usage is reduced if more producers are used. This is explained by the reduction in number of consumes which are performing burst request in parallel.

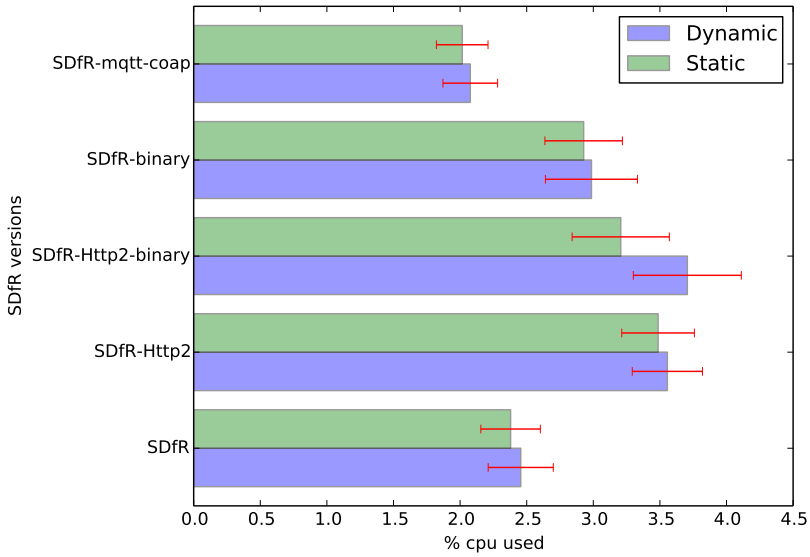
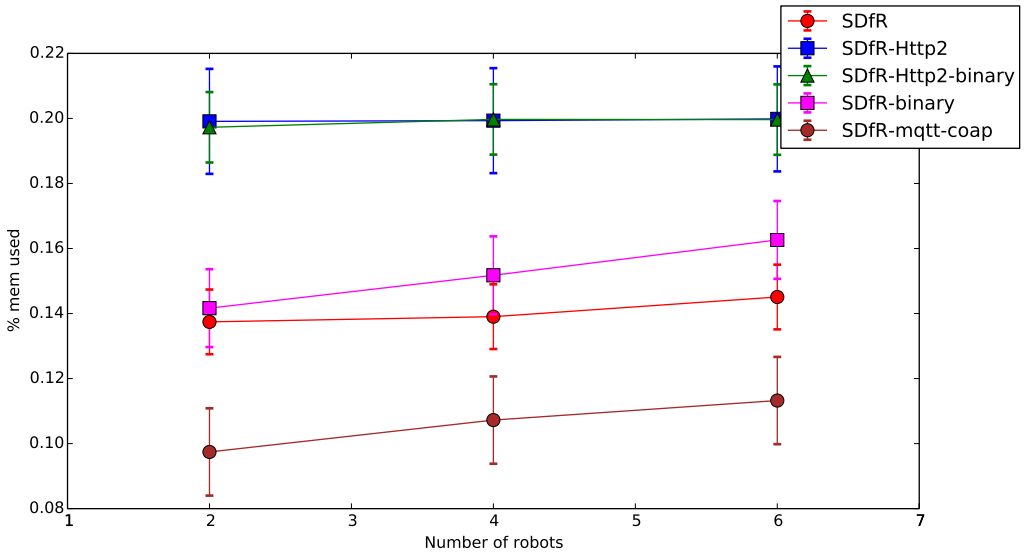


Figure 4.12 – % of cpu usage for 6 robots with 70% publishers.

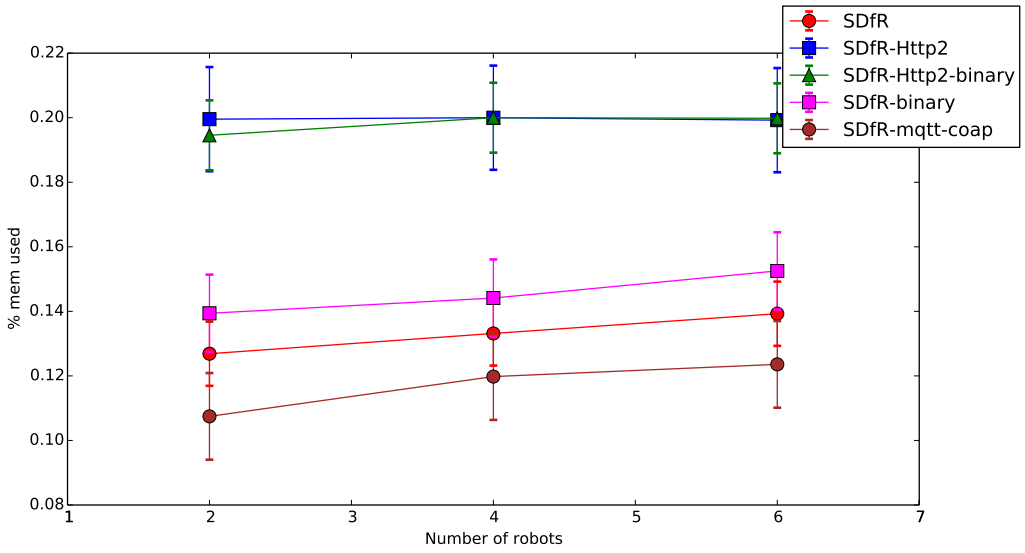
Figure 4.12 presents the evolution of *CPU* usage for all *SDfR* variants for both static and dynamic scenarios for 6 robots using a 70% pub/sub ratio. The usage is between 1.8% and 4.4%. It is observed that the *CPU* usage is higher for the dynamic scenario because the CPU time is consumed by the mobility management and the number of CPU slots is less for other processes.

Unsurprisingly, those results show that the *SDfR* variants that are using HTTP/2.0 as *API* layer protocol have a higher CPU consumption due to the encryption and compression phase while the plain-text variant have a lower CPU usage. The best result in all the scenario is obtained by *SDfR*-mqtt-coap. It can be said also that the *CPU* usage increased with the number of robots in the fleet with a rate of maximum 1% per robot.

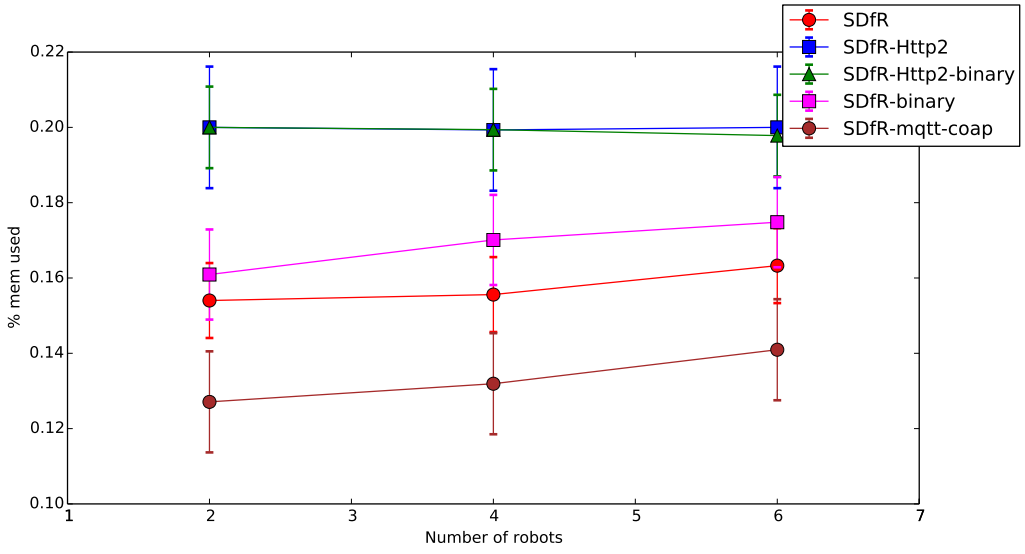
Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés



(a) Static with 30% publishers



(b) Static with 50% publishers



(c) Static with 70% publishers

Figure 4.13 – % of memory usage.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Besides the *CPU* usage, another critical resource in fleets of robots is the memory. As an example, robotic fleets may include visual sensors like 3d cameras which are in high demand of memory. A service discovery protocol needs to have a low usage of the robot memory.

A memory usage evaluation for all of *SDfR* variants was performed. Figure 4.13 shows the percentage of the memory used for static scenarios. For the HTTP/2.0 protocols the memory used tends to be constant to the number of robots used, but higher than the *SDfR* plain-text *API* protocol variants. Furthermore, the protocols that use a binary compression for the lower layer of the service have an increase memory usage. Figure 4.14 presents the results for both static and dynamic scenarios with 6 robots with 70% pub/sub ratio. It is noticed that the differences in memory usage between scenarios is less 1% of the total memory of the robot.

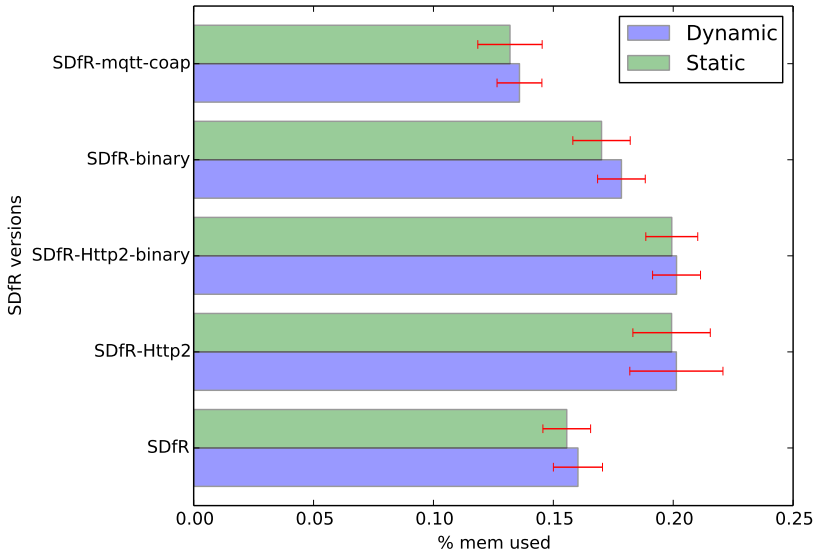


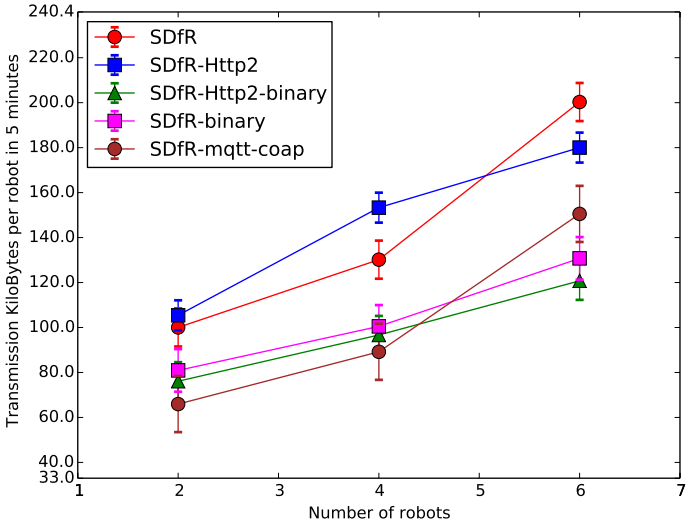
Figure 4.14 – % of memory usage for 6 robots with 70% publishers.

Network overhead

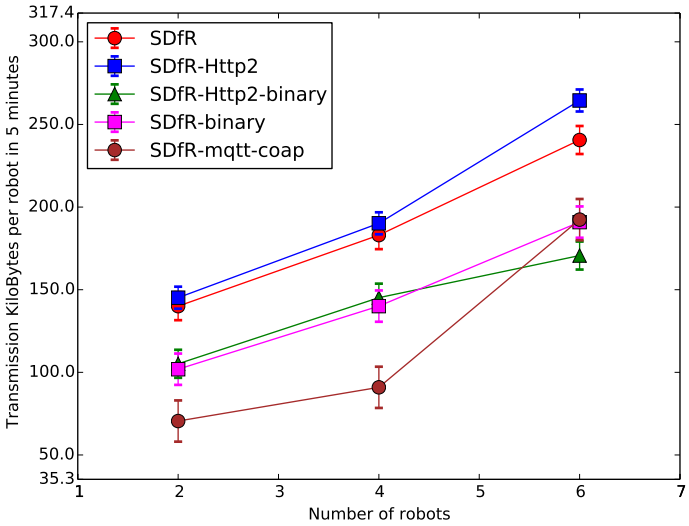
In a fleet context, the communication between peers in ad-hoc network are very sensitive. The transmissions can be unreliable due to the mobility of the robots. This is why the network overhead needs to be as limited as possible in order to allow services to exchange information. The network overhead generated by the different variant of *SDfR* was analyzed.

The measurements include the average of transmitted and received bytes per robot in a test run of 5 minutes. To increase the quality of the measurements, intermediary check-points for each metric at each 10 seconds were used.

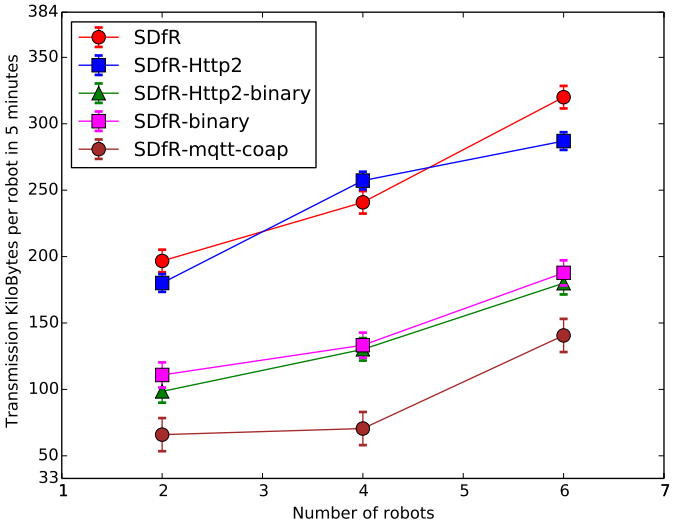
Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés



(a) Static with 30% publishers



(b) Static with 50% publishers



(c) Static with 70% publishers

Figure 4.15 – Average number of kilobytes transmitted per robot in 5 minutes.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Figure 4.15 presents the quantity of transmitted kilobytes per robot on each test-case in static scenarios. The number of bytes varies from 55 kilobytes for 2 robots with 30% providers to 320 kilobytes for 6 robots with 70% providers. This remains very limited considering the time of 5 minutes. Figure 4.16 compares the dynamic with the static scenario for 6 robots with 70% providers.

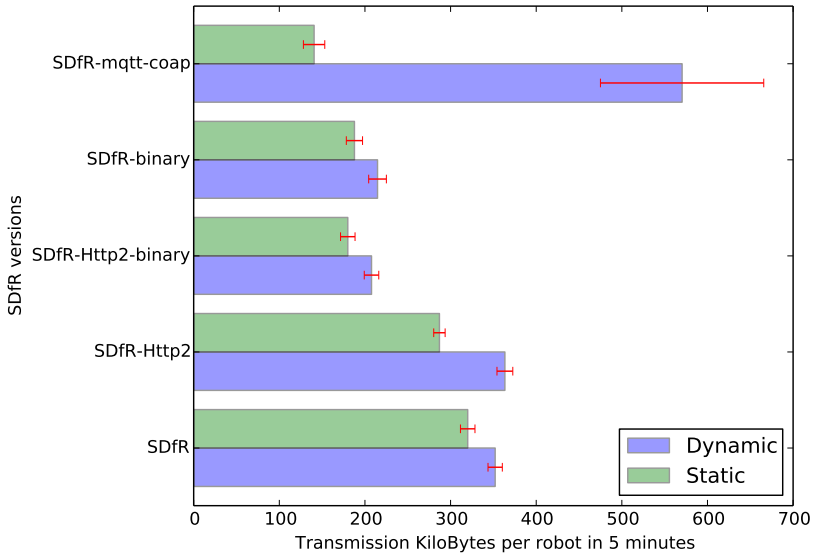
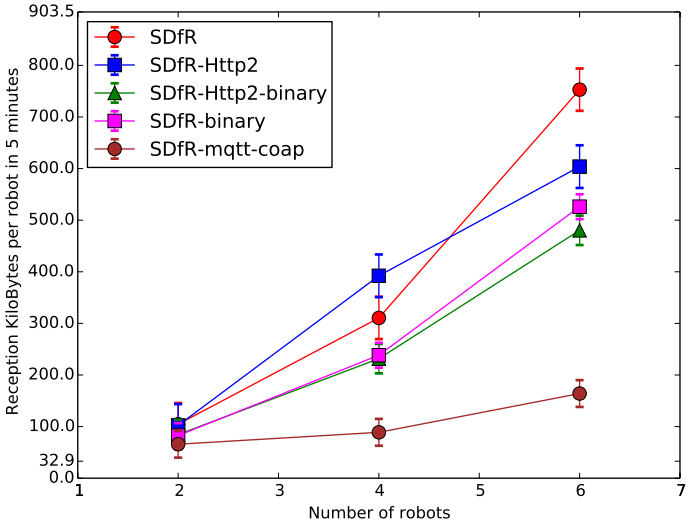


Figure 4.16 – Average number of kilobytes transmitted per robot in 5 minutes with 70% publishers using 6 robots.

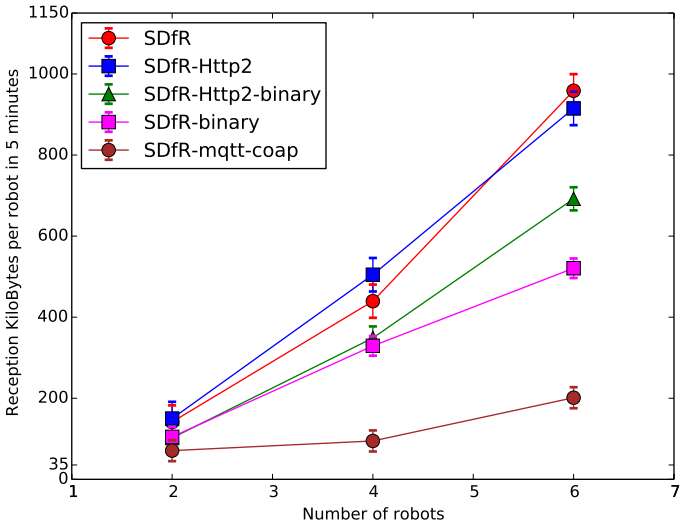
It is noticed that the use of binary protocols with Proto Buffers in the lower layer of *SDfR* can reduce the quantity of transmissions up to 50%. An interesting behavior is showed by *SDfR*-mqtt-coap variant when the number of robots increase to 6. Since this variant is using a MqTT broker on a central peer, the increase in number of robots generates packet loss and re-transmission which increases the quantity of kilobytes sent. More interesting, the fact of having this central point has a significant impact on the dynamic scenario since the quantity of kilobytes sent explodes to almost 700 kilobytes (Fig. 4.16). Under all circumstances, the quantity of kilobytes sent is less than 160 kilobytes/minute/robot which is reasonably for an ad-hoc WiFi network.

Figure 4.17 presents the average quantity of kilobytes received by a robot in 5 minute in the static scenario. As for the quantity of transmitted kilobytes, there is a correlation between it and the protocol used at the lower layer of *SDfR*. The quantity of kilobytes received is proportional with the number of robots and with the quantity of kilobytes sent by each robot.

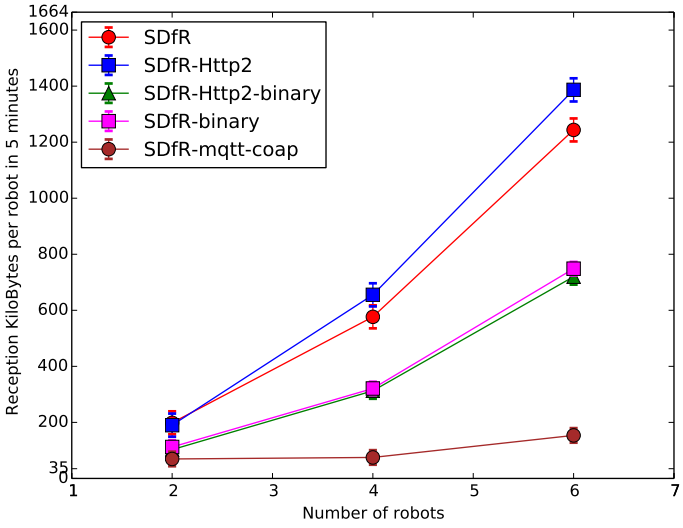
Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés



(a) Static with 30% publishers



(b) Static with 50% publishers



(c) Static with 70% publishers

Figure 4.17 – Average number of kilobytes received per robot in 5 minutes.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

The mobility (Fig. 4.18) has a great impact on the quantity of kilobytes received with a variation of more than 66,66% between dynamic and static scenarios for lower layer plain-text variants. *SDfR*-mqtt-coap has special behavior since it communicates with the central peer broker which re-transmits lost packages and it receives more in the dynamic scenario than in the static one. For the other variants of *SDfR*, the communication is unidirectional and in multi-cast UDP and the behavior of the service is not affected by the loss of packages.

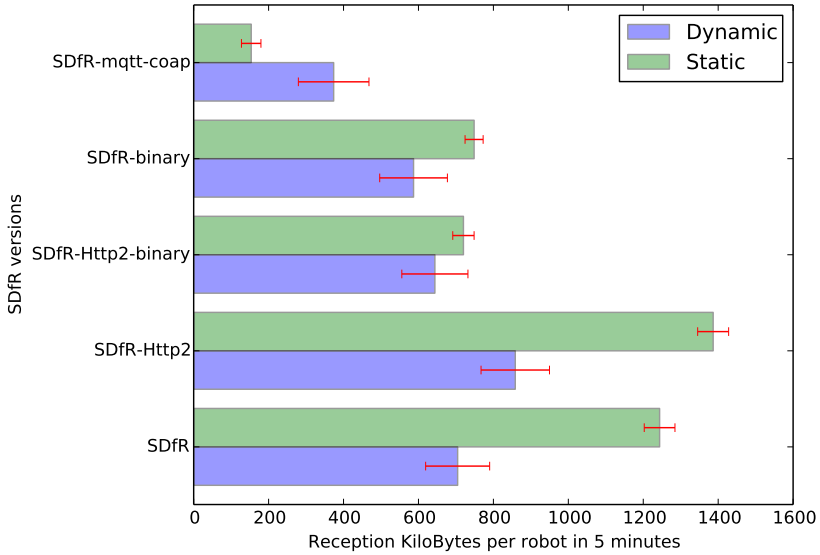


Figure 4.18 – Average number of kilobytes received per robot in 5 minutes with 70% publishers using 6 robots.

4.5 Summary

This chapter presented the challenges to define a service discovery protocol for robot fleet systems. It discussed the limited applicability of existing service discovery protocols in the context of robot fleets and then, it proposed a new protocol called *SDfR* that is suitable for service discovery inside an ad-hoc networked fleet. *SDfR* includes a two-layer service that provides neighbors and service discovery in both multi-cast and unicast communications. It includes a memory map that limits the overhead on the network. We made an extensive evaluation of different text and binary alternatives to implement *SDfR*.

The results show that using HTTP/2.0 as binary protocol for the *API* layer of *SDfR* increases the load on the robots as well as the response times. The gain of having a binary protocol using Proto Buffers in the lower network layer is less significant compared to the benefits of maintaining the retro-compatibility with *UPnP*. While the MqTT and *CoAP* variant performs better in a centralized context, *SDfR* with plain-text protocols shows to be a better fit for robots service discovery in decentralized environments. The results are encouraging, although benchmarking with a larger number of robots as yet to be made.

SDfR is further used in tooling provided with the timed automata model based programming methodology contribution detailed in the next chapters.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

5 ROSMDB: Development methodology

This chapter presents a toolchain, called Robot operating system Model Driven Behavior (ROSMDB)*, that provides a* MDD *over* SOA *approach to design with time properties, develop, validate, deploy and monitor multi-robot applications.*

5.1 From component services to fleet applications 84

5.2 Modeling component external interactions with timed automata 88

5.3 Validating service compositions 95

5.4 The ROSMDB toolset 105

5.5 Summary 123

The main objective of this chapter is to provide a methodology and a toolset that improve the process of creating new multi-robot applications. It proposes a software that allows the user to conceive a model, validate it, develop the code related to the model, deploy, run and monitor the resultant application inside a fleet of robots.

In our opinion, a robotic application in a fleet context can be designed starting from a behavioral model. We propose a development methodology adapted to multi-robot context which can be expressed based on the life-cycle of application development, as shown in fig. 5.1. The process starts with the design phase where each part of the application is modelled using the appropriate formalism. Next, we introduce a new step in the life-cycle of application development where these models are analyzed and verified against predefined properties. Only after this step has been completed, the development phase can start. Once the application has been fully developed, it can be deployed and executed inside the fleet of robots. During the run-time traces of the execution are collected in order to verify again the correct mapping between the model and the executed code. These traces are then analyzed and verified against

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

the same properties. Once the entire process ended, a new iteration can start in order to refine both the model and the source code used.

We think that this methodology can be automated in order to accelerate the process of developing new multi-robot application using a *MDD* over *SOA* approach. The following sections argue the software architecture and class of formalism chosen in order to provide such tool.

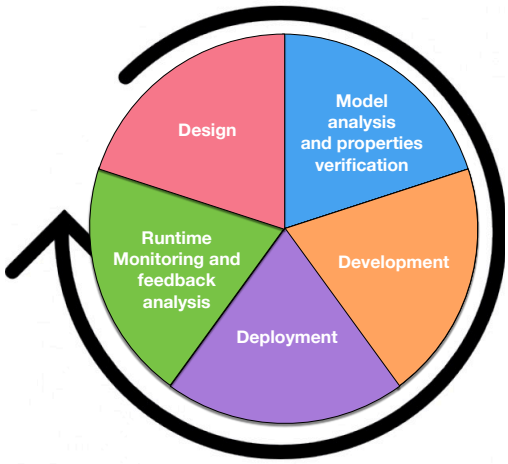


Figure 5.1 – Proposed robotic application life cycle

5.1 From component services to fleet applications

As already mentioned in section 2.4 of chapter 2, we believe the two most suitable robot middleware that can be applied to a fleet of robots are *ROS* and *MRDS*. Nowadays, these frameworks are often used for their service-based robotics packages and libraries. Both frameworks use distributed computing paradigm as their core architecture. These two are, however, very different: *ROS* is a open-source framework that is designed to run on *UNIX* based devices where *MRDS* is a *Windows* based framework supported by Microsoft. However, both frameworks use the same software architecture paradigm: *Service Oriented Architecture (SOA)*. The following subsections argues the use of *SOA* in the services developed with the toolset proposed in this chapter.

5.1.1 Service oriented architecture as root for model based robotic software development

SOA has been used with success in web services [Ponge, 2008]. In that context, *SOA* is referred as a collection of paradigms, standards and technologies such as *XML*, *SOAP* or *Web Services Description Language (WSDL)*. In the robotic context, services are the basic blocks of complex robotic behaviors and applications. This provides sand-boxing for each software component which renders the robotic application more robust and tolerant to failure and still disposing

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

of the flexibility in developing new components. Let's take an example where a robot needs to recognize an object while performing a collision avoidance movement. In case of service failure of the object detection component, if provided with the isolation of the services, the robot can still move without hitting obstacles.

A robot is a device composed of various sensors and actuators, each with their own micro-controller as low level processing units. On the highest level, vision processing, mapping and navigation, speech processing, and behavior selection may require enough resources, thus dedicated *CPUs*. All those components are interconnected in a distributed system to form a robot. Imagine that an object recognition service needs to be written in a programming language that offers a robust and complex level of computing like *C++*. Meanwhile, the collision avoidance service can be written in a prototyping interpreted language like *Python*. Each of the two services operates on different computational units. This example consolidates the need of a *SOA* because it increases the ability to develop distributed software components in various programming languages and for heterogeneous target devices.

At fleet level, the robots represent a series of multi-level interconnected processing units. This implies a large number of different systems that need to exchange data and those exchange mechanisms are provided by the *SOA* paradigm. Extrapolating the previous example, a series of robots can look up for specific objects cooperating together. The movement service in each robot needs to exchange information in order to avoid robot-to-robot collision. Even inside of each robot, the movement can be piloted by the detection service. All the exchanges need a communication infrastructure. The components message exchange mechanisms in *SOA* include data transfer via two types of messaging schemes: *request/reply*, and *publish/subscribe*, which is also the case of core components in *ROS* or *MRDS*.

SOA can improve performance in any general distributed application that may run on an elastic number of devices, even on a single *CPU* node. If the device provides only a single core processing unit, the runtime of the services is sequential. Its performance is equal or greater (due to overhead of process changing) than monolithic approach. Using specialised micro-applications (services) on a single processor machine can improve performance by taking advantage of parallelism. As *CPU* speed is reaching its upper boundaries due to overheating effect, computer engineers are increasing the *CPU* efficiency by increasing the number of cores per processor (as shown in fig. 5.2) in order to validate Moore's law[1] [Schaller, 1997]. In fig. 5.2, we can see that the single threaded performance, the frequency and the typical power have reached an upper boundary due to the overheating of *CPU* since 2010. But the performance of *CPUs* has continued to increase because the number of transistors continues to increases by increasing the number of core in each unit. With more cores, a larger number of threads and processes can be run in parallel up to certain limits (e.g. a *CPU* that manages a 1000 network concurrent connections in 1000 threads has a significant overhead generated by

[1] Moore's law is the observation that the number of transistors in a dense integrated circuit doubles approximately every two years. The observation is named after Gordon Moore, the co-founder of Fairchild Semiconductor and Intel, whose 1965 paper described a doubling every year in the number of components per integrated circuit, and projected this rate of growth would continue for at least another decade

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

the context switching between the threads). This is also true even for micro-controller robotic applications [Zhang, 2012].

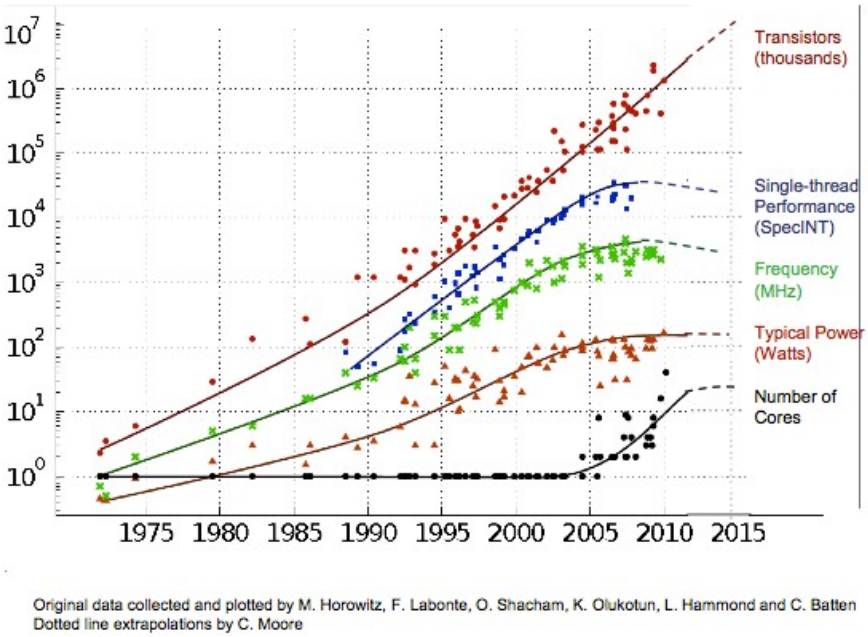


Figure 5.2 – Computing speed developments [2] over the years

Most of nowadays robots are equipped with *CPUs* that have at least 4 cores that allows at least two applications to run in parallel. As an example of a simple robotic service, a *SLAM* can be composed of two services that run in parallel: one that maps the environment and the second that controls the movement of the robot. These two services can run in parallel allowing the movement of the robot to be executed concurrently with the mapping function outside of a round-robin *CPU* scheduler.

With the development of grid computing, it turns out that the performances of a single super-computer is inferior to many slower computers working in parallel [Raicu et al., 2008]. The elastic number of machines interconnected inside a robotic fleet can allow for an increase in computational power [Hazelhurst, 2008]. If a complex task can be split in subtasks, it can be distributed across the fleet in order to accelerate the execution time, thus taking advantage of the parallel execution of subtasks.

[2] https://www.nextplatform.com/2015/08/04/future-systems-pitting-fewer-fat-nodes-against-many-skinny-ones/

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

5.1.2 Alternative approach for service oriented architecture

In contrast with *SOA*, traditional software development process like waterfall[3] usually results in developers working on a single monolithic application. A monolithic architecture implies that the software is written as one cohesive unit of code whose components are designed to work together, sharing the same memory space and resources.

The key advantages of monolithic software compared to *SOA* architecture include:

- The large number of cross-cutting concerns such as rate limiting, security features (audit trails or *Deny of service (DOS)* protection), logging, etc. As an example, in the logging component of a robotic application, the monolithic serial execution of the software ensures that the order of log messages is equivalent to the sequence of the code execution. In a distributed service context, the order relies on the synchronization between nodes as well as the concurrent execution of code.
- Simplified mechanism to bind modules to these cross-cutting concerns since the entire code is running in the same application. For a *SOA*, a messaging system is required to allow services to exchange data asynchronously. In the case of monolithic development, inserting new components in robotic application does not require a messaging framework because the hookups are done at development level and not at runtime.
- Possible performance advantages given by the shared-memory access which is faster than *IPCs*. As everything resides inside of a single robotic application memory space, the overhead of data exchange is none.

However, there are some lurking issues in monolithic approaches:

- Components in monolithic software tend be become tightly coupled with the evolution of the software. This makes very difficult the isolation of components for purposes such as independent scaling, evolution or code maintainability. On the other hand, each service in *SOA* is a self-contained component that has its own life cycle, allowing for isolated benchmarking, improving and monitoring.
- Components code reuse is very limited across monolithic applications. Due to its tight hookups with the main application, modules reuse is limited. In *SOA*, components can be easily reused since their connection to the main application relies only on the data exchange schemes.
- Scaling monolithic applications gets harder with the evolution of software during time since other components are stacked on top of existing ones. In *SOA* the code sandboxing allows the independent life-cycle of each service.

[3] The waterfall model is a sequential (non-iterative) design process, used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of conception, initiation, analysis, design, construction, testing, production/implementation and maintenance.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

- Monolithic applications are platform dependent and stack dependent. Usually they are implemented using a single development stack (i.e. *Java* or *.NET*). *SOA* paradigm ensures the ubiquity of the software and hardware involved in the robotic application.

Before the acceptance of *ROS*, monolithic architecture was mostly used in robotics applications."[...] For many applications, creating a monolithic entity that can address all aspects of a problem can be very expensive and complex; instead, creating multiple, more specialized entities that can share the workload offers the possibility of reducing the complexity of the individual entities [...]" [Parker, 2008]. In what follows, the entities are denoted as services inside a *SOA* approach.

As already mentioned in section 2.2.2, *SOA* greatly simplifies the implementation of highly-adaptive, constantly-evolving applications [Frénot et al., 2010]. It also reduces the process of developing and deploying new robotic applications as well as the execution time of complex task by taking advantage of task parallelization. Services are platform independent and they can be described, discovered and composed dynamically. In addition, higher levels of functionality provided by service-oriented programming reduce the implementation of redundant software. As a conclusion, *SOA* paradigm is very suitable to develop model driven robotic software.

5.2 Modeling component external interactions with timed automata

In mobile robotic fleets, a key feature of each robot is the ability to operate and interact with a highly dynamic, constantly changing environment as well as with the evolution of the fleet configuration. A successful method for creating such robotic software that handles this problem is to use a behavior *MDD* [Arkin, 1998]. Different elementary behaviors can be mapped as actions to different inputs from internal sensors or data exchange with other robots or environment. Combining elementary behaviors can generate more robust robotic control applications. Some examples of elementary behaviors are: the detection of a target by an optical sensor which results in the robot movement towards the target or the arrival of a network message from the fleet leader resulting in the task execution. This type of modular behavior approach, combined with *SOA* in a *MDD* offers the robotic application the possibility of using decentralized controlling modules mapped to specific tasks. This concept of using model base service composition has been already applied for dynamic environments in tools like Apache Felix iPOJO [Chollet et al., 2015]. Combining the ideas from the field of distributed artificial intelligence exposed by [Ferber, 1999] with the fleet robotic behavior based software, the application can be divided in subparts significantly simplifying the process of design and development. This also allows new behaviors to be easily added to the system.

As already mentioned in section 2.4 of chapter 2, nowadays multi-robot applications are mostly based on distributed computing paradigm that enables subsystems (e.g. nodes in *ROS*) to perform dedicated tasks. A first layer of complexity is represented by the task execution

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

code itself. All those subsystems are communicating with each other in order to exchange data and execute behavior control, thus adding an inside robot communication layer of complexity. At the fleet level, a third layer of complexity is added by the inter-robot communications. These layers of complexity in multi-robot distributed computing application make a software hard to debug at both development (code compilation) and monitoring (runtime), even if the general behavior model is simple.

One of the objectives of this chapter is to provide the user with a novel methodology to design the robotic fleet application using *MDD* by first designing the general robotic behavior as composition of elementary tasks. It provides a way to analyze if the interconnections between the elementary blocks do not violate specific guards and if the model is correct in both developing phase and monitoring at runtime.

5.2.1 Motivating example

Modelling robotic behaviors appears in countless scenarios. The reader is shown here two examples of robotic fleet application where modelling the robotic behavior using a design formalism like *MDD* prior to developing the software can reduce pitfalls.

A: Obstacle detection and avoidance navigation

One key module of a mobile robot fleet application is real time obstacle detection and avoidance. In nowadays context, most of the mobile robots are featured with some type of collision avoidance, starting from less complex algorithms which will stop the robot immediately when an obstacle is detected, towards more complex algorithms that will recompute the path in order for the robots to detour the obstacles as shown in fig. 5.3. Those latter algorithms involve not only the means of detecting the obstacle, its size and dimensions, but also they include a more resourceful computational unit, since they need to drive the robot around the obstacle and resume the path to the initial target. These algorithms are being part of the autonomous navigation concept. In general, in autonomous navigation, the environment may have known and unknown obstacles. All these assumptions are taken into account in the global path planning algorithm that plans the robot initial path in order to avoid known obstacles as well as in local path planning involved in unknown obstacles avoidance.

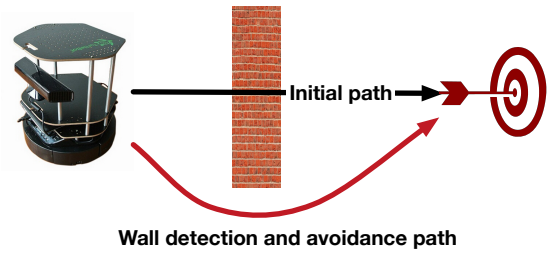
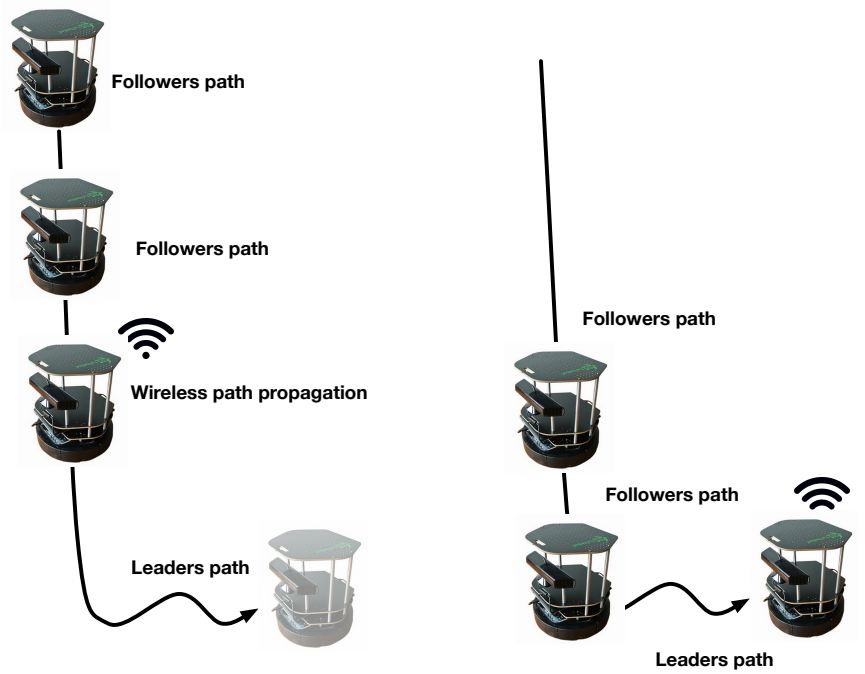


Figure 5.3 – Obstacle detection and avoidance

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

B: Fleet platooning

A fleet platoon is a group of robots that move in a coordinated way. A platoon is defined as a convoy of robots (i.e. train of robots) that move together in order to increase the throughput of circulation lane [Coelingh and Solyom, 2012]. There exist two types of platooning: centralized platooning with one leader and a decentralized platooning where each robot follows its predecessor.



(a) Leader path computation (b) Fleet robots following the leader path

Figure 5.4 – Centralized fleet platooning via wireless communication

Typically, in a centralized fleet platoon, there is one robot that leads the platoon while all other robots are following it with the same speed and within certain boundaries for inter-robot distance. As shown in fig. 5.4, the leader can decide to accelerate, to brake or change direction (see fig. 5.4a) and the following robots will mimic its actions (see fig. 5.4b). Coupled with autonomous navigation of an unknown map, the leader can avoid an obstacle leading to the entire fleet avoiding the same obstacle. Such systems that are found on the cooperation between peers (in the platooning case, cooperation between the leader and the other fleet members or between two adjacent robots) relay on wireless communication or on other sensors that can estimate the actions of the leader (e.g. an optical sensor like a 3d camera). In the case of wireless communication, the network should have standardized, efficient protocols with a minimum loss of packets.

In the second case of decentralized platooning, the mechanism of perceptions via sensors changes the general scenario because each robot is considered the local leader for the robot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

behind him. In this case, each robot needs to analyze the movement of the robot in front of it with information form the optical sensors and perform the movement to follow its local leader. This makes the propagation of the first robot (initial leader) longer and less fault tolerant.

5.2.2 Timed automata

In order to model robotic behaviors, multiple formalisms can be used. *Petri nets* have been successfully used to model Sensory-Based robots [Lyons and Arbib, 1989] as well as unmanned vehicles [Jaulin et al., 2012]. *Process algebras* usage in robotics includes specifications and planning of robotic missions [Karaman et al., 2009], distributed control architecture for robotics [Petersson et al., 2001] and definition of robotic behaviour [Košecká et al., 1997]. But, as mentioned in [König et al., 2009], [Egerstedt, 2000] and in [Ferber, 1999], the most used formalism in modelling *Artificial intelligence (AI)* and robotic behaviors is *FSM* and its extensions. Its applications include modelling autonomous navigation [Sales et al., 2010], path planning [Choset, 2001], mission planning and control [Pirjanian et al., 2000], defining the entire robotic behavior based on *FSM* [Martinoli et al., 2004, Bautin et al., 2012].

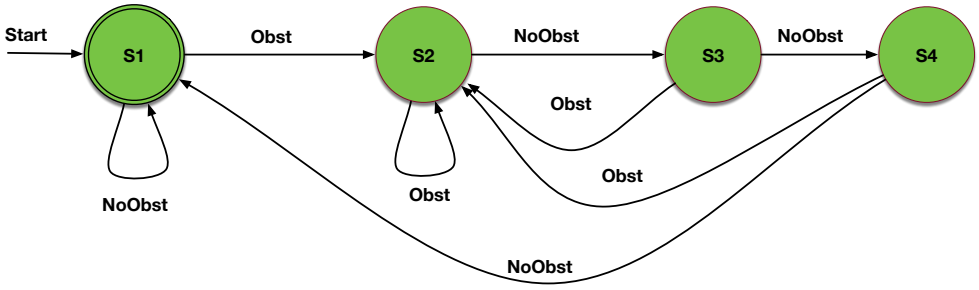


Figure 5.5 – Simple collision avoidance modelling as a *FSM*

Figure 5.5 shows the formal modelling of a simple collision avoidance system (example A) as a *FSM*. In this case, the robot is going straight in state $S1$. If the robot detects an obstacle, it will change its path with a 30 degrees angle until the obstacle is avoided ($S2$) and then will continue its linear movement in $S3$. Finally returns to its initial trajectory in $S4$ and resumes the movement in $S1$. The *FSM* definition based on this example is:

- The *alphabet*: $\sum = \{Obst, NoObst\}$ where *Obst* means that an obstacle was found and *No obstacle* means that no obstacle was found.
- Finite set of *states* (or locations): $Q = \{ S1, S2, S3, S4 \}$
- The set of *initial states*: $Q^0 = \{ S1 \} \subseteq Q$
- The set of *final (accepting) states*: $F = \{ S1 \} \subseteq Q$

In the example above $\xrightarrow{\text{Start}} S1 \xrightarrow{\text{NoObst}} S1 \xrightarrow{\text{NoObst}} S1 \xrightarrow{\text{Obst}} S2 \xrightarrow{\text{Obst}} S2 \xrightarrow{\text{NoObst}} S3 \xrightarrow{\text{NoObst}} S4 \xrightarrow{\text{NoObst}}$ $S1$ is a valid execution over A which recognizes the word $NoObst \cdot NoObst \cdot Obst \cdot Obst \cdot$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

$NoObst \cdot NoObst \cdot NoObst$ of the timed language L_A. A word like $NoObst \cdot Obst \cdot Obst \cdot Obst$ is not recognized by the automaton A because there is not a path starting from Q^0 towards F.

This simple *FSM* allows for displaying the interactions and component behavior of the system (in this case obstacle detection via a sensor and robot movement) but it is ignoring that all the actions/data collection happen in a time sensitive fashion. Different from software, where time is discrete and depends on the *CPU* cycles, in hardware, the time is continuous and events/actions can happen anytime. Since the robots are a complex combination of software and hardware, their behavior should be modelled with time into account because all the events, message sending and receiving inside the robot and outside happen in continuous time. Timing is an important abstraction in state change protocols.

[Alur and Dill, 1994] defined an extension of classical finite state automata [Hopcroft, 1979] called timed (finite) automata and was introduced to model real-time systems. It provides simple and powerful annotations of state-transitions timed constrained graphs by using real-valued clocks [Alur, 1999]. The previous example can be modelled using a timed automaton for better defining the real-time behavior of the robot as shown in fig 5.6.

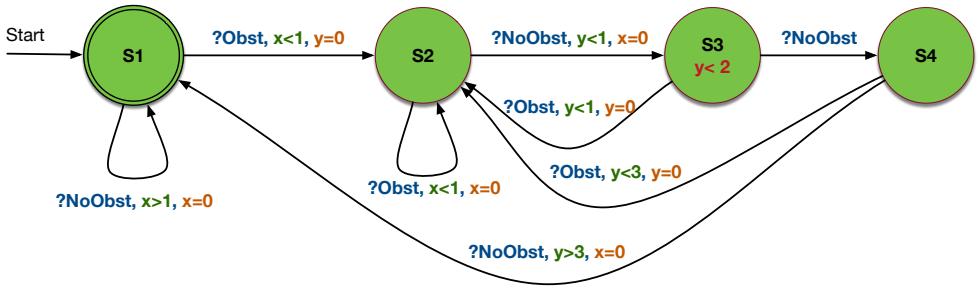


Figure 5.6 – Simple collision avoidance modelling as a Timed Automaton

The timed automata definition based on this example is:

- The *alphabet*: $\sum = \{Obst, NoObst\}$ where *Obst* means that an obstacle was found and *No obstacle* means that no obstacle was found. In this example, the state change is synchronized on the arrival of a notification from the optical sensor that detects the obstacle. In fig. 5.6, the arrival of a message is noted with ? in front of the alphabet element. In general the arrival of the message is market with $? < symbol >$ and the departure is mark with $! < symbol >$.
- Finite set of *states* (or locations): $L = \{ S1, S2, S3, S4 \}$.
- The set of *initial states*: $L^0 = \{ S1 \} \subseteq L$.
- The set of *final (accepting) states*: $L^f = \{ S1 \} \subseteq L$.
- The finite set of clocks: $X = x, y$. This example presents a set of 2 clocks: x being a global clock that counts the total time of the movement action and y which measures only the

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

time when the automaton enters the phase of obstacle avoidance (states: $S2$ to $S4$). The clock x resets when no obstacle is found in state $S1$ while y resets each time an obstacle is found and the automaton enters the state $S2$.

- $I : L \rightarrow C(X)$ the function that associates an *invariant* to each state. $C(X)$ represents the set of clocks constrains over X and it is formed using an arbitrary number of combinations of atomic expressions $x \# c$ where $x \in X$, $\# \in \{<, \leq, =, \neq, \geq, >\}$ and $c \in \mathbb{Q}$. The set of the clocks constraints $\in C(X)$ of the form $x < c$ or $x \leq c$ is noted $C(X)$.

- $E \subseteq L \times C(X) \times \sum \times 2^X \times L$ is a finite set of transitions where $e = (l, g, a, r, l') \in E$ is a transition from state l to l', where g is the guard, r is the set of clock to be reset and a is the label.

Time elapses in the locations, while the switches are instantaneous. A requirement for a timed automaton is that time must always progress. Visible in fig. 5.6, the state S3 presents an annotation $y < 2$ called invariant. An invariant denotes a boundary of the time spent in a state. It forces the trigger of a state change when the time has elapsed its value. In our case, the transition to $S4$ will be forced after the elapse of 2 time units.

In the example above, $\xrightarrow{\text{Start}} S1 \xrightarrow{1.23} S1 \xrightarrow{\text{NoObst}} S1 \xrightarrow{0.4} S1 \xrightarrow{\text{Obst}} S2 \xrightarrow{0.3} S2 \xrightarrow{\text{Obst}} S2 \xrightarrow{0.2} S2 \xrightarrow{\text{NoObst}} S3 \xrightarrow{\text{NoObst}} S4 \xrightarrow{4.3} S4 \xrightarrow{\text{NoObst}} S1$ is a valid execution over A which recognizes the word $(NoObst, 1.23) \cdot (Obst, 0.4) \cdot (Obst, 0.3) \cdot (NoObst, 0.9) \cdot (NoObst, 0.9) \cdot (NoObst, 5.11)$ of the timed language L_A. A word like $(NoObst, 1.1) \cdot (Obst, 1.1) \cdot (Obst, 0.4) \cdot (Obst, 2.4)$ is not recognized by the automaton A because there is not a path starting from L^0 towards L^f and it also violates the time guards.

In the example, the state is changed based on external observation of the environment. Different classes of timed automata propose different external behaviors. The external behavior, also called observable behavior, is given by its sequences of external actions. It also considers the passage of time as an externally observable event.

5.2.3 Event recording timed automata

In the toolchain that is proposed in this chapter, each state is seen as a black box where the logic and the actions inside the state are transparent for the system. It focuses on how the states transitions are done in response to external stimuli, called events (e.g. the detection of an obstacle by an optical sensor). It monitors how the system reacts to those stimuli and how the robotic application is composed from different timed automata that are synchronized on reciprocal events.

Event recording automata (ERA) is the class of timed automata that is the most suitable for analyzing such behavior [Alur et al., 1999]. In an *ERA*, each input symbol is mapped to a clock. Every time a symbol is recognized, its assigned clock is reset. $\perp$ symbol signifies that a given

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

input symbol has not been recognized yet (i.e. the initial value of all clocks is set to $\perp$). The time domain T of the clocks in an *ERA* is represented by $\{v | v \in \mathbb{R}_{\geq 0}\} \cup \{\perp\}$.

In this chapter work, the timed automata alphabet is represented by the set of externally observable stimuli, called events. An event is represented by arrival or departure of a message. The events can be a *ROS* notification (i.e. a new entry on a topic or a data exchange on a service) or a network message. The robotic application behavior is expressed by the product of all services formalized as synchronized timed automata. (i.e. the state change of each service happens only on an event). The property of the *ERA* that resets the clocks each time the events mapped to them are triggered, allows for monitoring and measuring the interval between two consecutive occurrences of reciprocal events.

The example represented in fig. 5.6 is modelled as an *ERA* in fig. 5.7. The reader should notice that, compared to a general timed automata, the set $\{X\}$ of clocks is mapped to each symbol becoming $X = \{XObst, XNoObst\}$. Those clocks are set to 0 each time the associated event is triggered. Even on state changes that are not subject to clock guard (e.g. transition from $S3$ to $S4$), the clock associated to the event is reset. In this context, the guard evaluates to the last time an event was observed.

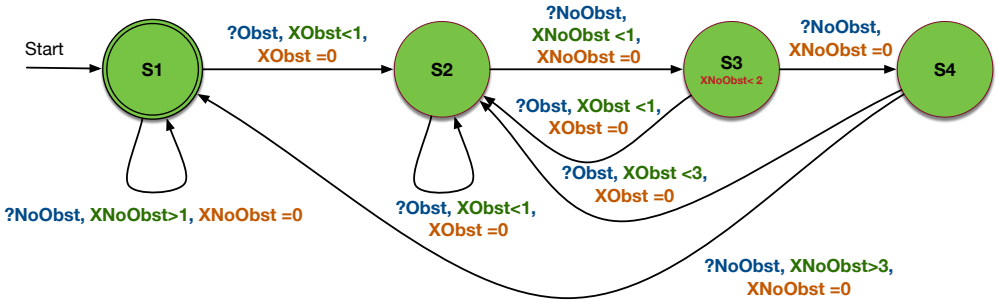


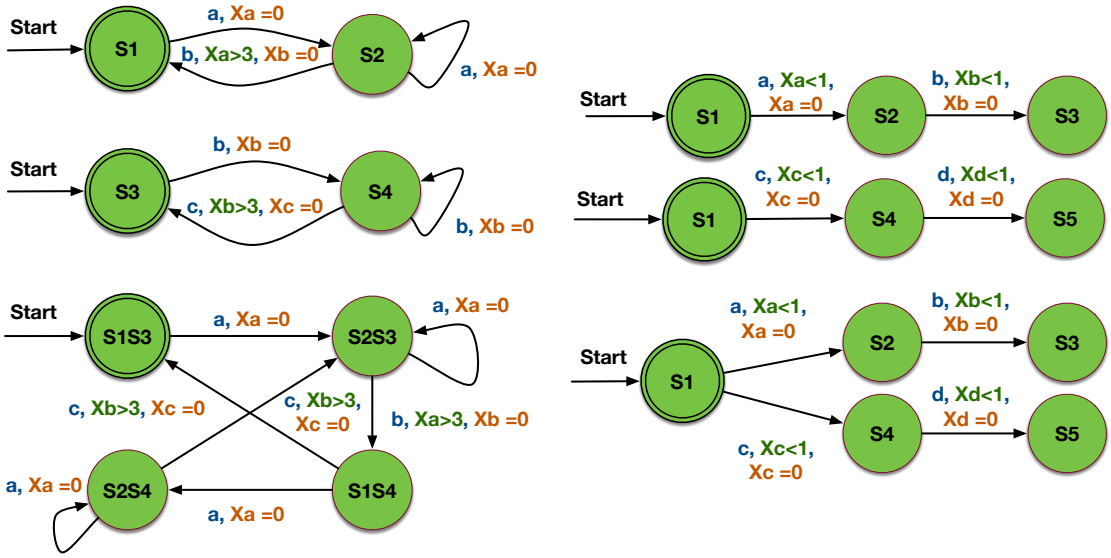
Figure 5.7 – Simple collision avoidance modelling as an Event Recording automata

The Event Recording Timed automaton definition based on this example is:

- The *alphabet*: $\sum = \{Obst, NoObst\}$ where *Obst* means that an obstacle was found and *No obstacle* means that no obstacle was found. In this example, the state change is synchronized on the arrival of a notification from the optical sensor that detects the obstacle. In fig. 5.6, the arrival of a message is noted with ? in front of the alphabet element. In general the arrival of the message is market with $? < symbol >$ and the departure is mark with $! < symbol >$.
- Finite set of *states* (or locations): $L = \{S1, S2, S3, S4\}$.
- The set of *initial states*: $L^0 = \{S1\} \subseteq L$ defined as a singleton. The execution of an *ERA* needs to be deterministic, thus it can only have a single initial state.
- The set of *final (accepting) states*: $L^f = \{S1\} \subseteq L$.
- The finite set of clocks: $X = XObst, XNoObst$.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

As already mentioned, the input symbols create a tight influence on the value of the mapped clocks. This fundamental property of an *ERA* makes the automata *complementable* (i.e. *ERAs* are closed under complementation) and *determinable* (i.e. for each in-deterministic *ERA* there is a transformation to a deterministic *ERA* with the same language). *ERAs* can be extended as long as the values of the clocks only depend on the symbols they are mapped to. Those properties ensure that the *product* and *union* of two *ERAs* are internal (closed) operations. Fig 5.8a presents a product of two automata: A - the automaton formed by $S1$ and $S2$ and B - the automaton formed by $S3$ and $S4$. In fig 5.8b represents the union of the automaton formed by the states $S1$, $S2$ and $S3$ with the automaton formed by the states $S1$, $S4$ and $S5$.



(a) Event Recording Automata Product

(b) Event Recording Automata Union

Figure 5.8 – Event Recording Automaton operations

5.3 Validating service compositions

In order to propose a solution to apply *MDD* to multi-robot application, our approach is combining a model based construction over a *SOA*. As mentioned in [Hilaire et al., 2008], formal driven prototyping and composition can be applied to *Multiagent Systems (MAS)*, thus to multi robot systems.

The formalism that we think fits such *ROS* based software is *ERA* since it allows modelling our robotic external behavior as timed automata where the leaps of time between the arrivals of messages can be monitored and conditioned.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

5.3.1 Applications, services and components

The examples presented in subsection 5.2.1 can be represented as services that can be combined inside of a same robotic application that is running on each robot inside a fleet. The robotic application, in this case, consists of a fleet platooning capable of avoiding collisions. Each robot is running the application in order to form the distributed behavior of the fleet.

Each of the application services are specialized on a specific task. Example A (Obstacle detection and avoidance navigation) represents a robotic service that allows for navigation without colliding with the environment (i.e. with objects and with other robots from the fleet). Example B (Fleet platooning) allows for a designed leader to send the path constructed by the service in example A via a *IP* network. The other fleet members (i.e. followers) will use the information to control the navigation system in order to follow the leader.

Each one of the two services is composed of dedicated components that deals with a particular part of the robot. The components include managers for optical sensors, actuators for movement, *IP* communication, etc. Each of these components are modelled using an *ERA* and represents the building blocks of the multi-robot application.

In order to model the component, the design begins by specifying the complex robotic application and then dividing it into successively smaller pieces called services. Each service is divided again in components. This approach of design, called *top-down*, is often found in software programming where the developing starts with the main procedure that names all the major functions it needs. Later, the developing focuses on the requirements of each of those functions and the process is repeated.

Multiple components can be composed in order to form a service. Multiple services can be associated to form a robotic application that will run on the fleet members. This approach is called *bottom-up* approach where the building blocks are first modelled in great detail. These elements are then linked together to form larger subsystems, which, at their turn, are linked, sometimes in many levels, until a complete top-level system is formed.

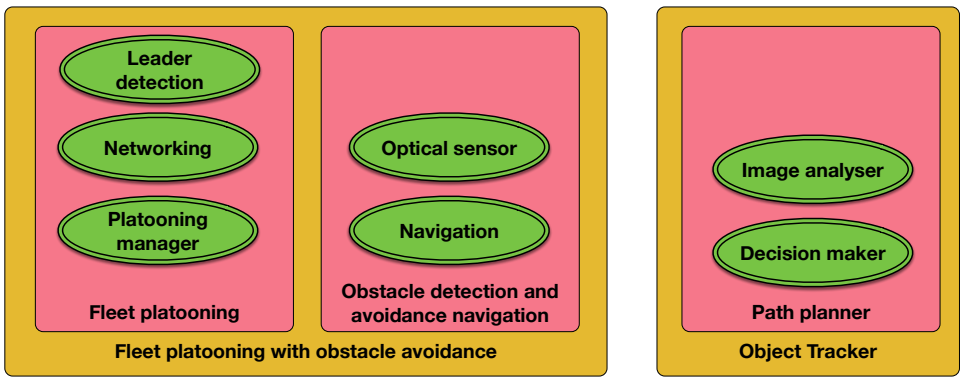


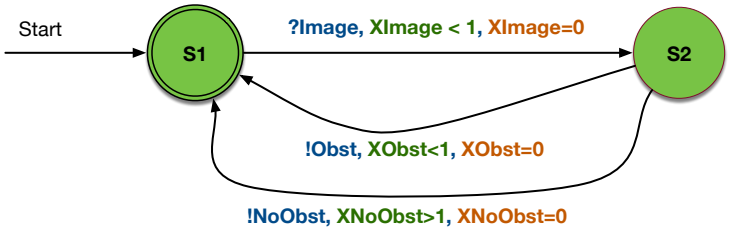
Figure 5.9 – Bottom-up approach of Event Recording automata composition

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

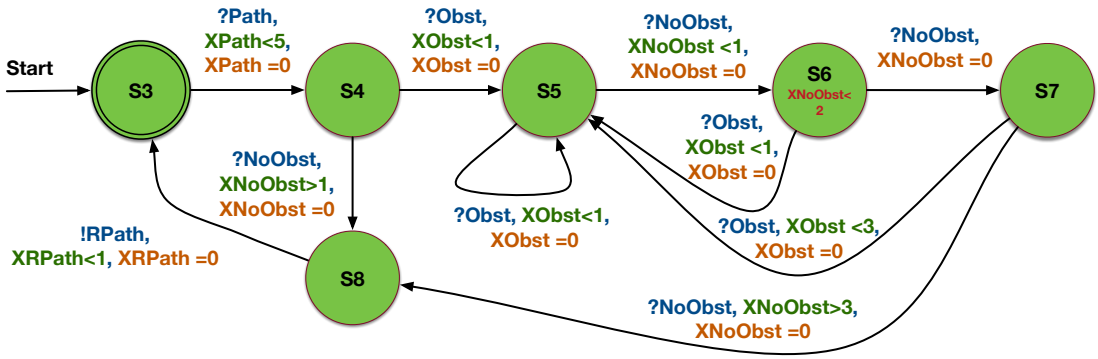
Our *MDD* proposal combines both techniques. First the *top-down* approach is used. A robotic application is structured in task specialized services. Each service is organized in components that manage a specific sensor or actuator. Once the organization of the robotic application is done, each of the components is modelled using an *ERA*. The *bottom-up* approach is then used in order to model the general behavior of the application. The component's timed automata are composed in order to model services which join into robotic application model.

The example presented above, fleet platooning capable of avoiding collisions illustrated in fig. 5.9, is composed of the two services presented in subsection 5.2.1 in a *bottom-up* approach. Each of the service is composed as follows:

Obstacle detection and avoidance navigation The model is constructed, as shown in fig. 5.10, from two components:



(a) Optical sensor component Event Recording Automaton



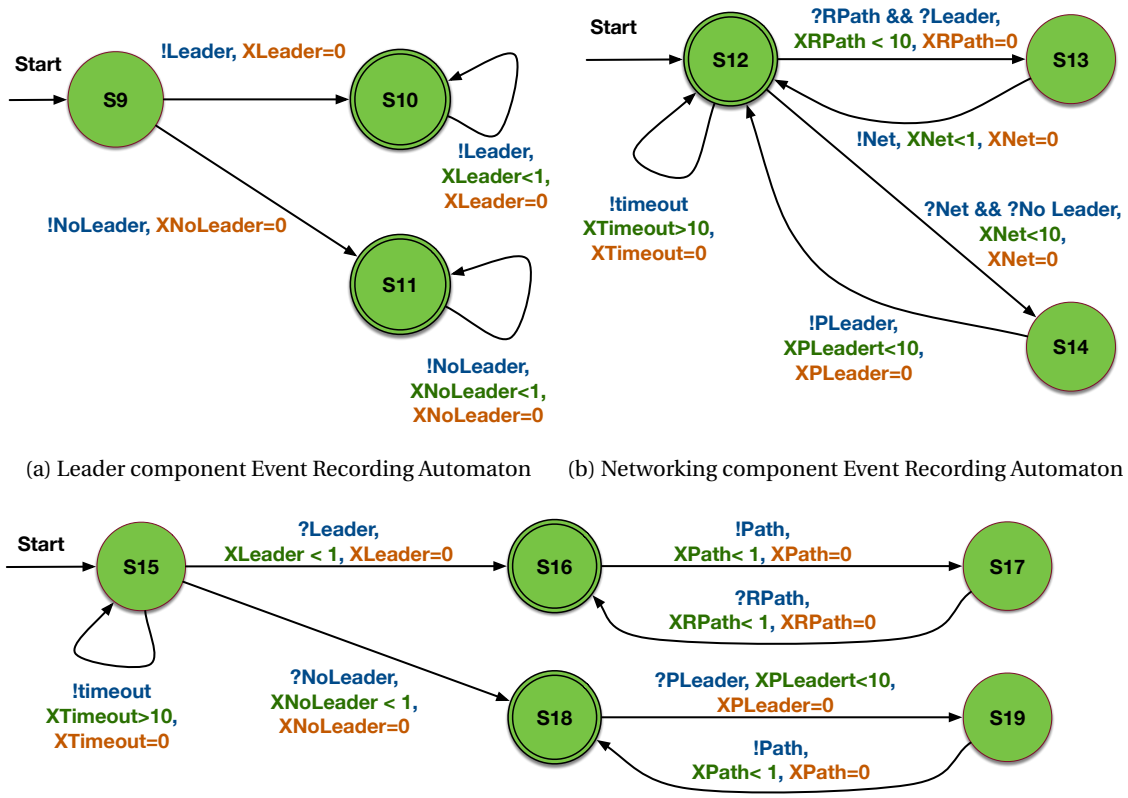
(b) Navigation component Event Recording Automaton

Figure 5.10 – Components of service: Obstacle detection and avoidance navigation

Optical sensor component Its dedicated task is to get a depth and RGBA[4] image from the optical sensor in order to analyse if an obstacle is present on the robot trajectory. As shown in fig. 5.10a, the set of states is formed from only two states. In the initial state, $S1$, the model waits for the arrival of a image from the optical sensor that should arrive with a frequency smaller than 1 time unit. When the *image* message arrives, the model changes state to $S2$. In this state, the image is analyzed for the

[4] RGBA stands for red green blue alpha. While it is sometimes described as a color space, it is actually simply a use of the RGB color model, with extra alpha channel information

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés



(a) Leader component Event Recording Automaton (b) Networking component Event Recording Automaton

(c) Platooning manager component Event Recording Automaton

Figure 5.11 – Components of service: Fleet platooning

presence of obstacle. If an obstacle is found, the system returns in $S1$, which is also the final state, by firing an *Obst* message. If no obstacle was found a *NoObst* message is triggered.

Navigation component This component performs the actual movement of the robot and avoids obstacles. The *ERA* is formed from several states (see fig. 5.10b.). In the initial state, $S3$ (which is also an acceptance/final state), the robot waits for a *Path* message. When the message arrives, the robot executes in state $S4$ the trajectory specified in the message. If an obstacle is found (i.e. arrival of a *Obst* message), the model switches to state $S5$ where the robot turns around in order to avoid the obstacle. If no new obstacles are found on the new trajectory, the system will avoid the initial obstacle in states $S6$ and $S7$. If new obstacles are found during the transitions from $S5$ to $S7$, the system returns in state $S5$. In all cases, the system will enter in state $S8$ where the corresponding executed path, which include path for avoided obstacles is observed (i.e. messages from the odometer services of the robots which can result in a slightly different path compared to the leaders path) and then sent via a *RPath* message. In the end, the system will switch to $S3$ and the execution will loop again.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Fleet platooning The model is constructed, as shown in fig. 5.11, from three components:

Leader detection component The main task of this model is to decide if a robot is a leader (first robot in the platooning row) or not based on a configuration file. The example can be future detailed with a leader election process, but it is out of the scope of this example. The model, visible in fig. 5.11a, is composed of an initial state $S9$ where the decision is made. If the robot is a leader, the system will translate in $S10$ by triggering a *Leader* message. If not, the system will end in $S11$ and a *NoLeader* (i.e. not a leader) message will be sent. In both cases, the model will end in a final state. When switching from $S9$ to $S10$ or $S11$, the model has no time constrains. Once in one of the final states, the system will send in a loop the corresponding message to the state in order to inform the other components of the robot role.

Networking component This component is managing the *IP* messages that are exchanged between to robots in the fleet. It starts in state $S12$ (see fig. 5.11b) where it loops with a *timeout* of 10 time units if no other message is produced. When a *RPath* (i.e. real path) message arrives to the leader, the model will switch to $S13$ where it prepares the *IP* message, broadcasts it via *Net* symbol and then returns to the initial state $S12$. If the robot is not a leader, the model will switch to $S14$ when a network message arrives and transforms it to a *PLeader* (i.e. Path from leader) message.

Platooning manager component This component represents the main logic of the service. As shown in fig. 5.11c, it waits in $S15$ until the role of the robot is decided by *Leader detection component*. If the robot is a leader, it will switch to state $S16$ where it computes and sends via a *Path* message the trajectory of the fleet. Then it translates into $S17$ where it waits for the execution of the trajectory via *RPath* (i.e. real path). Then the real executed path is taken into account in $S16$ in order to repeat de process. If the robot is a follower, in $S18$ it will wait to execute the path coming from the fleet leader via *PLeader*(i.e. path from leader). In $S19$ the model integrates the trajectory to execute and sends a *Path* message to its internal movement service.

All the *ERA* presented above represent the models for individual parts after the breakdown of the entire robotic application in task specific components following a *top-down* approach. In order to prove that this *MDD* approach respects the initial behavior of multi-robot application, those components *ERA* need to be combined in models for each service, followed by the composition of service models into global application behavior model.

5.3.2 Event recording automata composition

The composition of *ERA* (and timed automata in general), called product construction for timed automata, is used to define a complex model as a product of subsystems. Let A_1 =

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

$\langle L_{A_1}, L^0_{A_1}, \sum_{A_1}, X_{A_1}, I_{A_1}, E_{A_1} \rangle$ and $A_2 = \langle L_{A_2}, L^0_{A_2}, \sum_{A_2}, X_{A_2}, I_{A_2}, E_{A_2} \rangle$ where the set of clocks X_{A_1} and X_{A_2} are disjoint. The product construction of two timed automata is presented in fig. 5.12. Each of the automaton have two states ($L_{A_1} = \{S1, S2\}$ and $L_{A_2} = \{Sa, Sb\}$).

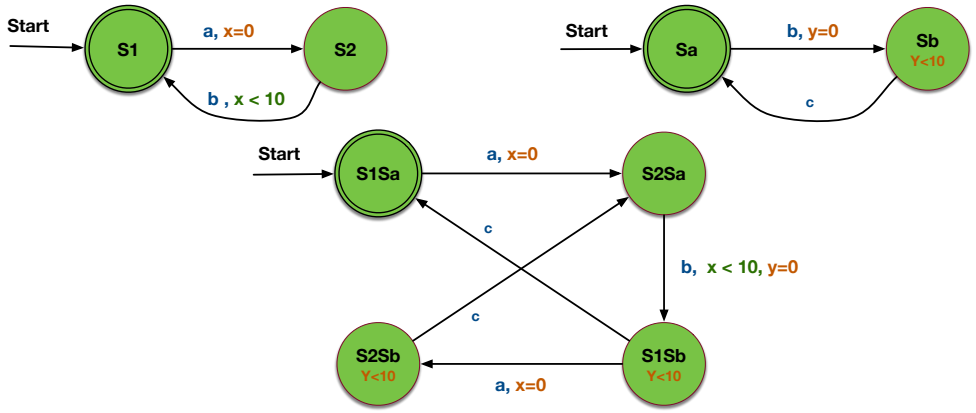


Figure 5.12 – Timed automata product construction example

$A_1 \parallel A_2$ represents the product of timed automata A_1 and A_2. The product is defined as the automaton: $A_1 \parallel A_2 = \langle L_{A_1} \times L_{A_2}, L^0_{A_1} \times L^0_{A_2}, \sum_{A_1} \cup \sum_{A_2}, X_{A_1} \cup X_{A_2}, I, E \rangle$ where $I(l_1, l_2) = I(l_1) \wedge I(l_2)$ and the transitions are defined by:

- for $a \in \sum_{A_1} \cap \sum_{A_2}$, for every $\langle l_1, g_1, a, r_1, l'_1 \rangle$ in E_{A_1} and $\langle l_2, g_2, a, r_2, l'_2 \rangle$ in E_{A_2}, E contains $\langle (l_1, l_2), (g_1 \wedge g_2), a, r_1 \cup r_2, (l'_1, l'_2) \rangle$.
- for $a \in \sum_{A_1} \setminus \sum_{A_2}$, for every $\langle l_1, g_1, a, r_1, l'_1 \rangle$ in E_{A_1} and for every l_2 in L_{A_2}, E contains $\langle (l_1, l_2), g_1, a, r_1, (l'_1, l_2) \rangle$.
- for $a \in \sum_{A_2} \setminus \sum_{A_1}$, for every $\langle l_2, g_2, a, r_2, l'_2 \rangle$ in E_{A_2} and for every l_1 in L_{A_1}, E contains $\langle (l_1, l_2), g_2, a, r_2, (l_1, l'_2) \rangle$.

The locations of the product ($L_{A_1 \parallel A_2} = \{S1Sa, S2Sa, S1Sb, S2Sb\}$) are pairs of component locations and the invariant of product location $S1Sb$ is the conjunction of the invariants of the component location ($S2$ and Sb). The transitions are obtained by synchronizing the transitions with identical labels and different types: emission of the event market with ! and reception of the event, marked with ?.

In the case of black box states, where the focus is to analyze the external behavior of the model, the labels represent the arrival or departure of a message (e.g. *ROS* or *IP*). In order to construct the product of *ERA*, the synchronization of the events is done on transitions with the same event, but with different directions (i.e. the departure of a message is synchronized with the arrival of the same message in another automaton or vice versa).

The components of the service Obstacle detection and avoidance navigation produce the product in fig. 5.13. Even if the clocks of same message type have the same name in both *ERA*, the clocks are disjoint. (i.e X_{obst} from the *Optical sensor component* is not the same clock with X_{obst} from *Navigation component*). Let's suppose that the exchange time between

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

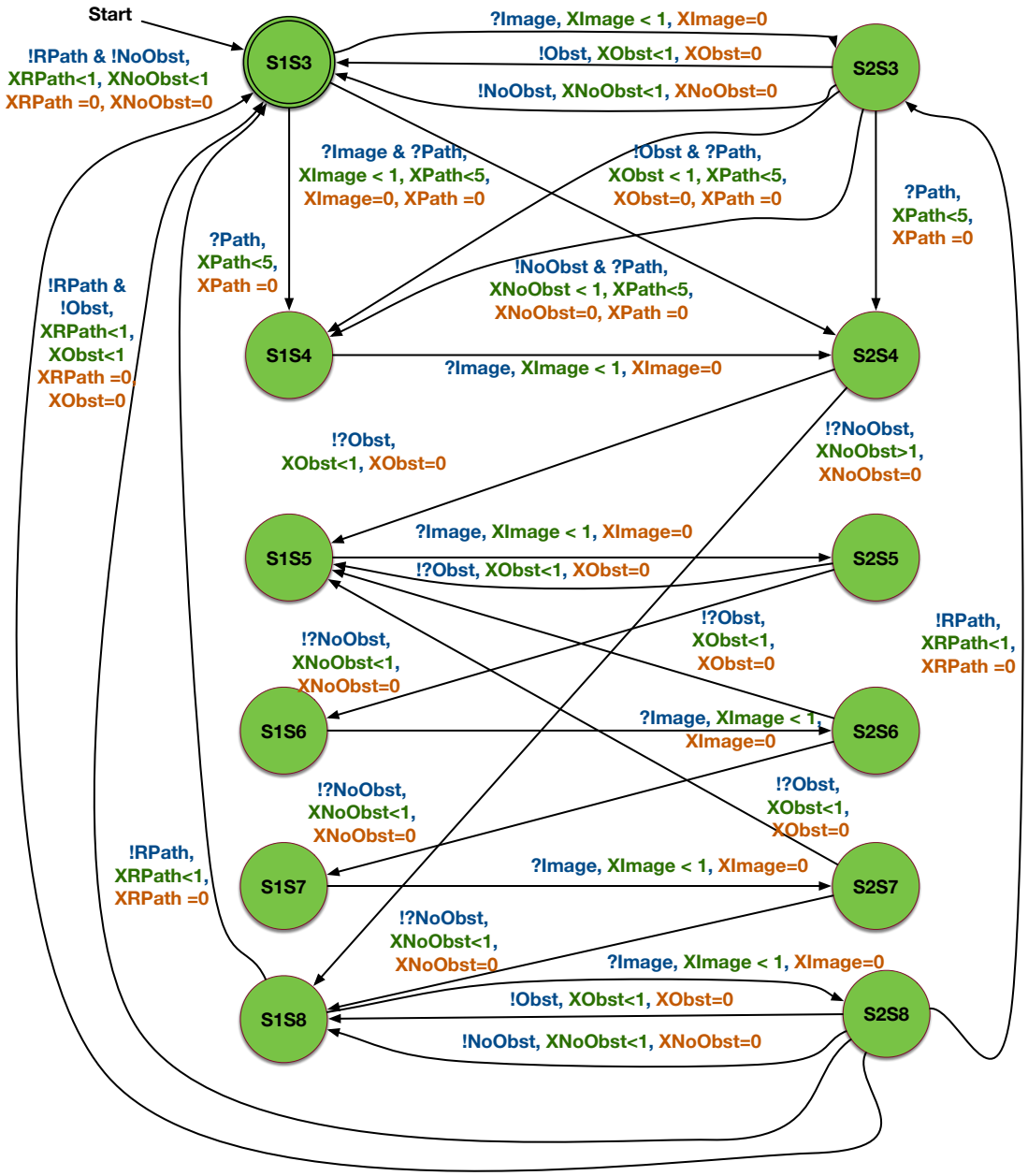


Figure 5.13 – Product construction of components for service: Obstacle detection and avoidance navigation

components (e.g. *ROS* and *IP* messages RRT[5]) is null. For simplification purposes, fig. 5.13 only presents the most strict guard of a $X_{Opticalsensor} \cup X_{Navigation}$. The final product automaton consists of:

[5]Round-trip time (RTT) is the length of time it takes for a signal to be sent plus the length of time it takes for an acknowledgment of that signal to be received.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

- The *alphabet*: $\sum = \sum_{Opticalsensor} \cup \sum_{Navigation}$. The reader should also notice in this example that the synchronization of events is marked with !? meaning that a message was sent from one automaton and received in the second one.

- Finite set of *states* (or locations): $L = L_{Opticalsensor} \times L_{Navigation} = \{ S1S3, S1S4, S1S5, S1S6, S1S7, S1S8, S2S3, S2S4, S2S5, S2S6, S2S7, S2S8 \}$.

- The set of *initial states*: $L^0 = L^0_{Opticalsensor} \times L^0_{Navigation} = \{ S1S3 \} \subseteq L$.

- The set of *final (accepting) states*: $L^f = L^f_{Opticalsensor} \times L^f_{Navigation} = \{ S1S3 \} \subseteq L$.

- The finite set of clocks: $X = X_{A_1} \cup X_{A_2}$.

- $I(l_1, l_2) = I(l_1) \wedge I(l_2)$.

The product in fig. 5.13 represents the internal state of the entire service. But when this service is composed with other services, the internal behavior of the service is not requested, since the service itself becomes a black box. The full representation of the product can be simplified just to two states showing how the service reacts with the external environment as shown in fig. 5.14.

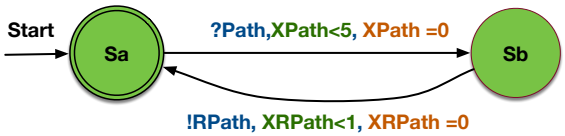
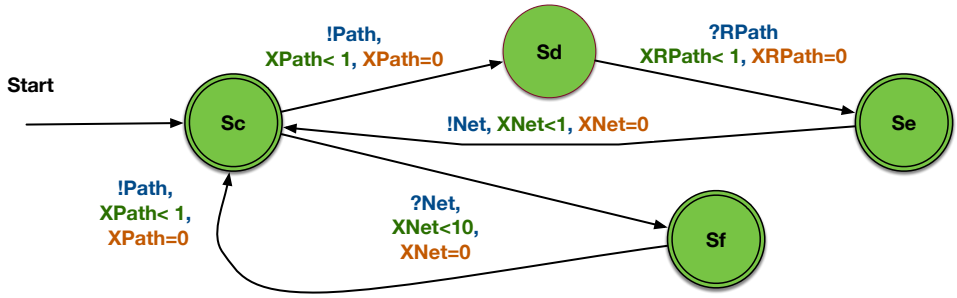


Figure 5.14 – Reduced product construction of components for service: Obstacle detection and avoidance navigation

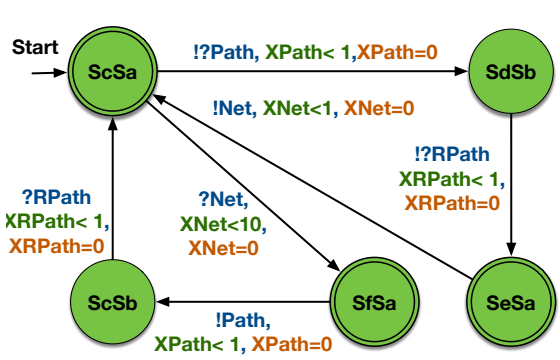
A reminder of *Obstacle detection and avoidance navigation* as well as a detailed product construction for the service *Fleet platooning* can be found in Appendix C. The simplified automaton for *Fleet platooning* service is presented in fig. 5.15a. The two services can be composed again in order to obtain the application automaton. This product is displayed in fig 5.15b and the simplified version is shown in 5.15c. This one state application (i.e. a black box application) is exchanging data with the environment (e.g. with the other peers of the fleet) via *Net* events. A last composition can be made in order to study the entire fleet behavior. It consists of combining the automaton with itself.

All the product construction can be realized based on the properties of *ERA*. [Alur and Dill, 1994] has analyzed the closure of: union and intersection. Closure under union and intersection is based on fact that *ERA* are in-deterministic, thus they can support more than one location. This ensures that the composition of *ERA* still remains an *ERA*. Furthermore, the product of *ERA* is associative: $A_1 \parallel A_2 \parallel A_3 = (A_1 \parallel A_2) \parallel A_3$.

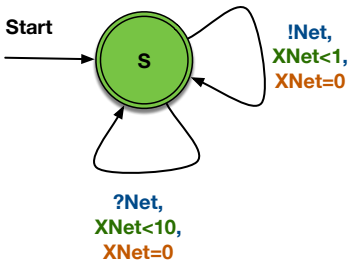
Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés



(a) Reduced product construction of components for service: Fleet platooning



(b) Product construction for entire robotic application



(c) Reduced Product construction for entire robotic application

Figure 5.15 – Product of entire example application

5.3.3 Model validation

Each component *ERA* and their compositions (i.e. the *ERA* of services and application) are checked in the tool presented in the next section against the following properties in order to verify the correctness of the model:

- *Reachability* properties specify if a property can possibly be satisfied by the model. A location S_s of a *ERA* A is reachable if a state S_q with location component S_s is a reachable state of the transition system S_A[6]. In order to verify the reachability property, A and the set $L^f \subseteq L$ (i.e. the final set of states) are considered. The analysis consists of determining if $\forall s \in L^f$, s is reachable or not. In the previous examples, this property can be addressed to verify if an obstacle can be detected by the optical sensor and navigation starts (i.e. optical sensors component in $S1$ and the navigation component in $S3$)

- *Liveness* properties asserts that a model can eventually reach a good state. The analysis

[6]A state of a transition system S_A is a pair $\langle s, v \rangle$ such that $s \in$ locations of A and v is a clock interpretation of X with the property that v satisfies $I(s)$.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

of the liveness properties of a real-time system consists of checking for reachability of cycles containing final states in a *ERA* model, A. The main challenge for such cycle analysis is to handle the infinite domain of clocks. [Alur and Dill, 1994] proposed the first approach to handle real valued clock domain by partitioning the clock domain into a finite set of *regions*, called R. The result of $R \times A$ has a finite symbolic semantics for A. In the examples above, such property can be defined to verify if a leader can send a network message to the followers (i.e. the leader component can switch from $S10$ to $S10$ while the networking component translates from $S12$ to $S13$ and back to $S12$).

The properties of *ERA* used in the model checkers are usually written as form of *temporal logics*. Temporal logic focuses on the qualitative time properties rather than quantitative ones. The main purpose of temporal logics is to verify if there exists a path between the states that will satisfy it. In the analysis of the properties over the composition of component models, *TCTL* is used. Timed computational temporal logics allows the verification of the formula over several time lines.

The syntax of a *TCTL* expression is composed of:

- a set of propositional variables $AP = \{\phi, \psi, ...\}$.
- logical operator like $\neg$, $\vee$.
- temporal modal operator:
  - $A\phi$ - all - ϕ has to hold on all paths starting from the current state.
  - $E\phi$ - exists - there exists at least one path starting from the current state where ϕ holds.
  - $\bigcirc\phi$ - next - ϕ has to hold at the next state.
  - $\Box\phi$ - globally - ϕ has to hold on the entire subsequent path.
  - $\Diamond\phi$ - finally - ϕ eventually has to hold (somewhere on the subsequent path).
  - $\psi U\ \phi$ - until - ψ has to hold at least until ϕ , which holds at the current or a future position.
  - $\psi R\ \phi$ - release - ϕ has to be true until and including the point where ψ first becomes true; ψ never becomes true, ϕ must remain true forever.

A *TCTL* formula can be satisfied by an infinite sequence of truth evaluations of variables in *AP*. These sequences can be viewed as a timed-word over alphabet $\sum$ of an *ERA* A. Let $w = a_0, a_1, a2, ...$ be such a word. Let $w(i) = a_i$. Let $w^j = a_j, a_{j+1}$, which is a suffix of w. Formally, the satisfaction relation $\models$ between a word and an *TCTL* formula is defined as follows:

- $w \models p$ if $p \in w(0)$

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

- $w \models \neg\psi$ if $w \not\models \psi$
- $w \models \phi \vee \psi$ if $w \models \phi$ or $w \models \psi$
- $w \models \bigcirc\psi$ if $w^1 \models \phi$ (in the next time step ϕ must be true)
- $w \models \phi U\ \psi$ if there exists $i \geq 0$ such that $w^i \models \psi$ and $\forall 0 \leq k < i$, $w^k \models \phi$ (ϕ must remain true until ψ becomes true)

[Alur and Henzinger, 1994] presents the reachability problem of timed automata in general as PSPACE-complete. In [Courcoubetis and Yannakakis, 1992], reachability is shown to be PSPACE-complete even with a small number of clocks. For *TCTL*, model checking is also PSPACE-complete [Alur et al., 1993].

These properties for the examples above translate into:

- `reachability property` - an obstacle can be detected by the optical sensor and navigation starts:
$$E\Diamond(S1 \wedge S3)$$
- `liveness property` - the leader component can switch from $S10$ to $S10$ while the networking component translates from $S12$ to $S13$ and back to $S12$:
$$E \bigcirc S10\ U\ \ A\Box(S12 \rightarrow S13 \rightarrow S12)$$

Even if properties of models and their compositions can be verified using query languages like *TCTL* inside model checkers, there is a need to verify those properties against a model refinement based on runtime observations. The models are based on theoretical assumptions which in real time systems could be invalided by factors outside the system.

5.4 The ROSMDB toolset

In order to allow new multi-robot applications to be develop using *MDD* approach, we propose a toolset called *Robot operating system Model Driven Behavior (ROSMDB)*. *ROSMDB* provides the ability to create new *SOA* applications designed for fleet environment. It allows for design of the services based on *ERA* and provides a complete tool-chain from the model verification up to the execution (runtime) of applications. With *ROSMDB* it is possible to compare the theoretical model checking with real-time observation collected at runtime and to iterate until both give the same behavior. Furthermore, this contribution includes also a framework that will be enabled in each robotic software issued from the toolset that will shadow the management of generic events non-related to robotic behavior.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

5.4.1 Global overview of the environment

ROSMDB tool-chain is a software designed to accompany the process of development of new multi-robot applications from the conception phase through the lifecycle of a software: development, deployment and runtime. It was designed based on the idea that output of the tool-chain (i.e. multi-robot services that interact together to form the behavior of the final application) will represent a *ROS* based *SOA* application. In the output, it generates *ROS* nodes which will be executed together at runtime.

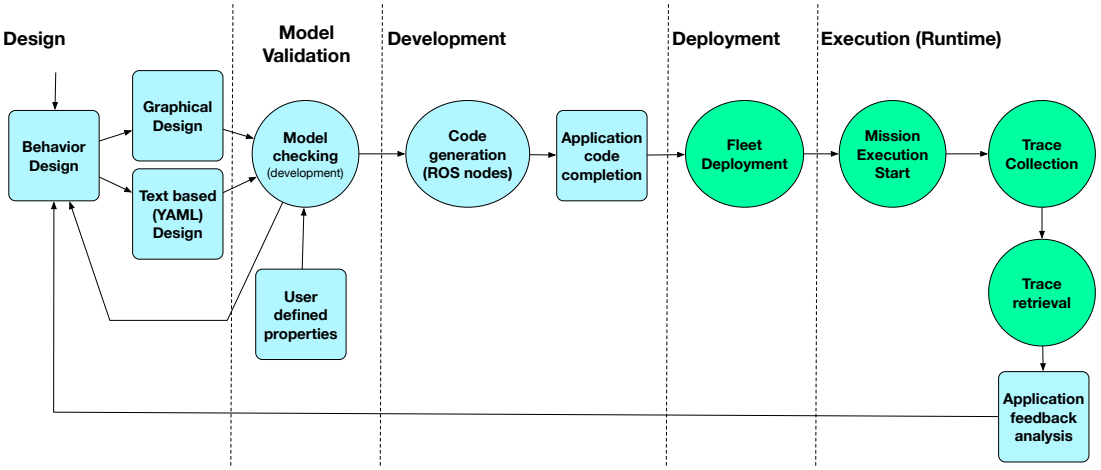


Figure 5.16 – General architecture for ROSMDB

The generic architecture of the toolset is visible in fig. 5.16. The squares in the figure mean that the user action is required while the circles mean that action is automatically executed. The blue colored shapes represent steps taking place in the development phase while the green shapes represent steps executed at run-time on the robot.

The cycle of creating new multi-robot application starts with the design of the software application using a *top-down* method in order to split the behavior in small task oriented components. They can be inputted in two methods: using a drag-and-drop *Graphical User Interface (GUI)* where the automaton can be drown or using a text-based input. This step is called in fig 5.16 *Design*. The next phase is the local offline validation of the models and their composition. The *Model Validation* verifies the model against predefined properties and against user specific rules. In the *Development* phase, the model is translated into *Python ROS* based code and the link to the *ROSMDB* framework is injected. In this step, the specific code of the application needs to be filled in. The tool-chain provides a method of provisioning all the fleet robots in the *Deployment*. At the *Execution (Runtime)*, the application is executed and traces of the model execution are collected in order to allow the user to analyze the real behavior of the application and compare it to theoretical one. This mechanism allows for multiple iterations of the design process in order to refine the software robotic behavior.

The tool-chain is implemented as *Python* web based application. The core application is

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

developed on top of Django[7], Bootstrap[8] and jQuery[9] frameworks and includes technologies like: *Hypertext Markup Language 5 (HTML5)*, *Cascading Style Sheets 3 (CSS3)*, *JavaScript (JS)*. It is cross-platform and cross-version. All the steps can be executed even from mobile devices (e.g. tablets, etc.) allowing for a better reactivity in the development process. All the projects developed in the tool-chain are stored on the server which needs to be able to exchange data with the robots.

ROSMDB provides a file management tool (see fig. 5.17) for multiple multi-robot application *projects*. Each *project* is stored on internal workspace on the server running the tool-chain. A *project* example can be `Package Transportation` that allows for a fleet of robots to form a platoon and move freight from point A to point B. Each *Project* is composed of one or multiple multi-robot *application(s)* that are needed, but which can be independently used in other projects. The example project can be formed from the `fleet platooning with collision avoidance` presented in the previous sections (which can be used as stand-alone application) and from a `Package handling` application. Each *application* represents a combination of *service* that have dedicated tasks (e.g `Obstacle avoidance`, `Platooning`) and each service is composed of *components* that manage a specific sensor, actuator or behaviour logic (e.g `Navigation`, `Optical Sensor`, `Networking`).

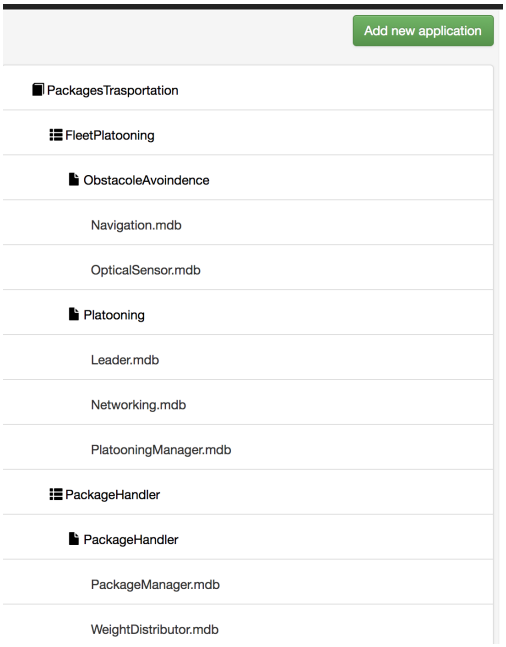


Figure 5.17 – File manager in ROSMDB

5.4.2 Design

In order to design and later verify the correctness of the behavior of a robotic application inside a project, the division of the application into services and components needs to be done by using a *top-down* approach. This structuring is translated into model organization via the file management presented above. As a reminder, the structure is represented in *ROSMDB* as follows:

[7] Django is a free and open-source web framework, written in Python, which follows the model-view-template (MVT) architectural pattern.

[8] Bootstrap is a free and open-source front-end web framework for designing websites and web applications. It contains HTML5 and CSS3 based design templates for typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions

[9] jQuery is a cross-platform JavaScript library designed to simplify the client-side scripting of HTML

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

- 1st layer consists of projects: Multiple projects can coexist in the same workspace.
- 2nd layer consists of applications: Multiple applications form a project, but each application can be used as standalone application or in different projects.
- 3rd layer consists of services: Multiple services form a robotic application.
- 4th layer consists of components: Multiple components form a service

In order to use later in *model validation* and *execution* phases a *bottom-up* composition approach, the only layer that a user needs to address is the 4th layer. Each component needs to be designed independently as an *ERA*. The design is persisted as a *.mdb* file in order to be used later for component compositions and model validation.

ROSMDB allows the design of each component *ERA* in two ways:

- a *GUI* interface that allows for the automaton to be drawn using a drag-and-drop interface.
- a text based interface that allows the design in *YAML Ain't Markup Language (YAML)*[10] format.

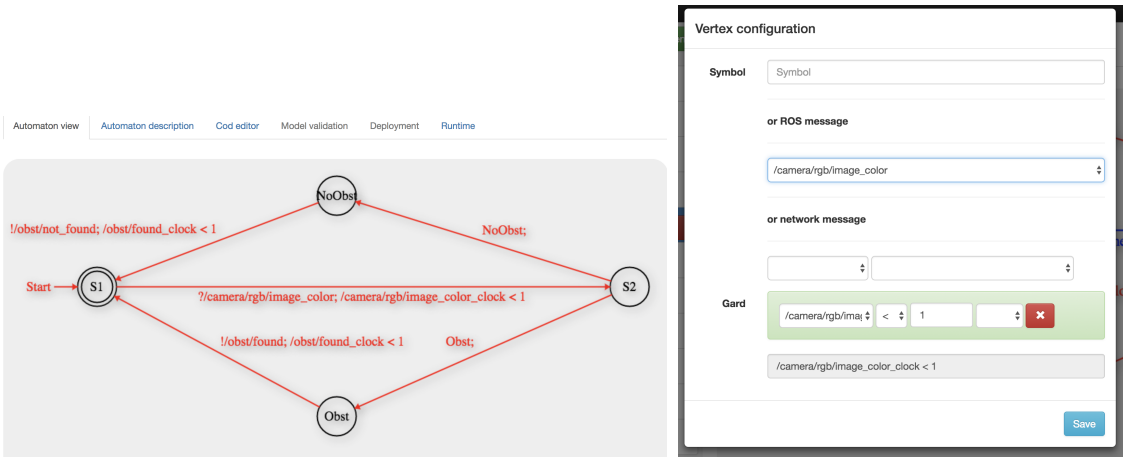
Each of the ways of inputting the component *ERA* is automatically converted into the other type (i.e. the *ERA* designed with the *GUI* is converted to *YAML* and vice-versa). The automaton is persisted as *YAML* in the *.mdb* file corresponding to the component.

As shown in fig. 5.18a, the *GUI* allows the design of states and transitions as a graph. Each state is represented as a circle with its name. As shown in fig. 5.18b, a transition label can be a local symbol specific for the automaton, a *ROS* message or a network message. For each symbol in the automaton, a clock is assigned. The transitions are decorated with the symbol (message) that needs to be recognized (sent or received) and the clocks guards. The guards are represented as boolean logic composition of multiple clocks (if needed). The clock reset is omitted from the representation because, in an *ERA*, the clock corresponding to a given symbol is automatically reset when the input is recognized.

In the text based mode, the states and transitions can be declared as shown in the *YAML* listing 5.1. The information required for states include the state name, the flag *isAcceptState* if the state is a final state, and other optional information like global variables and constants needed by the state. A transition is defined by the origin and destination state (*nodeA* and *nodeB*), by the label (which can be one of *ros_message,network_message* or *symbol* values) and by the time *guard*.

[10] Initially called Yet Another Markup Language, YAML is a human-readable data serialization language. It is commonly used for configuration files, but could be used in many applications where data is being stored (e.g. debugging output) or transmitted (e.g. document headers).

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés



(a) Graphical representation of automaton in ROSMDB (b) Graphical symbol assignation in ROSMDB

Figure 5.18 – Graphical user interface in ROSMDB

```yaml
automaton:
  states:
  - constants: ''
    isAcceptState: 'True'
    name: S1
    variables: ''
  - constants: ''
    isAcceptState: 'False'
    name: S2
    variables: ''
---
  vertex:
  - guard: /camera/rgb/image_color_clock < 1
    network_message: ''
    nodeA: S1
    nodeB: S2
    ros_message: /camera/rgb/image_color
    symbol: ''
```

Listing 5.1 – States and transitions YAML in ROSMDB

The design tool-chain allows the components to declare communication channels that will be future used to interconnect components and services via both *GUI* and *YAML* editors (see listing 5.2). In order for the output robotic application to be *ROS* compliant, the tool-chain allows the registration on *ROS* topics to provide an asynchronous communication and on *ROS* services to offer a synchronous exchange mechanism. Furthermore, the robotic application will be executed in a fleet environment with multi *ROS* master nodes (i.e. one master node

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

per robot). The tool-chain allows for the declaration of network broadcast communication scheme that will be used later by the *ROSMDB* framework to open and handle network messages transparently.

```
---
ros:
  services: []
  topics:
  - function: listen_for_image
    message_type: /sensor_msgs/Image
    name: /camera/rgb/image_color
    type: listen
network:
- callback: callback_function
  name: NetworkMessageExxample
  port: '10001'
  push: push_function
---
```

Listing 5.2 – Communications channels YAML in ROSMDB

After the design of each components *ERA* in one of the previous editors and the definition of all the communication channels, the models can be composed and validated by verifying certain properties of them.

5.4.3 Validation

In order to validate the properties of components and services *ERA*, *ROSMDB* is integrated with a part of an external model checker, called *UPPAAL*. *UPPAAL* is an integrated tool box for modelling (via a graphical simulator), validation and verification (supported by an automatic model-checking) of real-time systems designed as compositions (networks) of timed automata, extended with various data types that include bounded integers and arrays.

The philosophy behind *UPPAAL* is to model a real system using timed automata or classes of timed automata, simulate it and then verify a set of properties on it. Multiple models can be constructed in order to form a system which consists of a network of processes that are composed of locations (or states). The transitions between these states define how the system behaves. *UPPAAL* is running the system interactively in the simulation step in order to check if the systems behave as intended.

UPPAAL is formed of two parts: a *GUI* and a model-checker engine. The *GUI* is written in Java and is executed independently on Java Runtime Environment. The model design and the simulation component are part of the *GUI*. The use of this *GUI* is not subject to *ROSMDB*.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

The model-checker is written in *C* and can run on the same machine as the *UPPAAL GUI*, but also on a dedicated server. This latter component is used in *ROSMDB* on the same server. This verifier can check reachability properties, (i.e. if a certain state is reachable or not) as well as liveness properties (i.e. if there is a state where the system blocks). It represents a complete search that covers all possible dynamic behaviors of the system. The engine reduces the verification to solving simple constraint systems [Larsen et al., 1995] by combining a symbolic technique with on-the-fly verification.

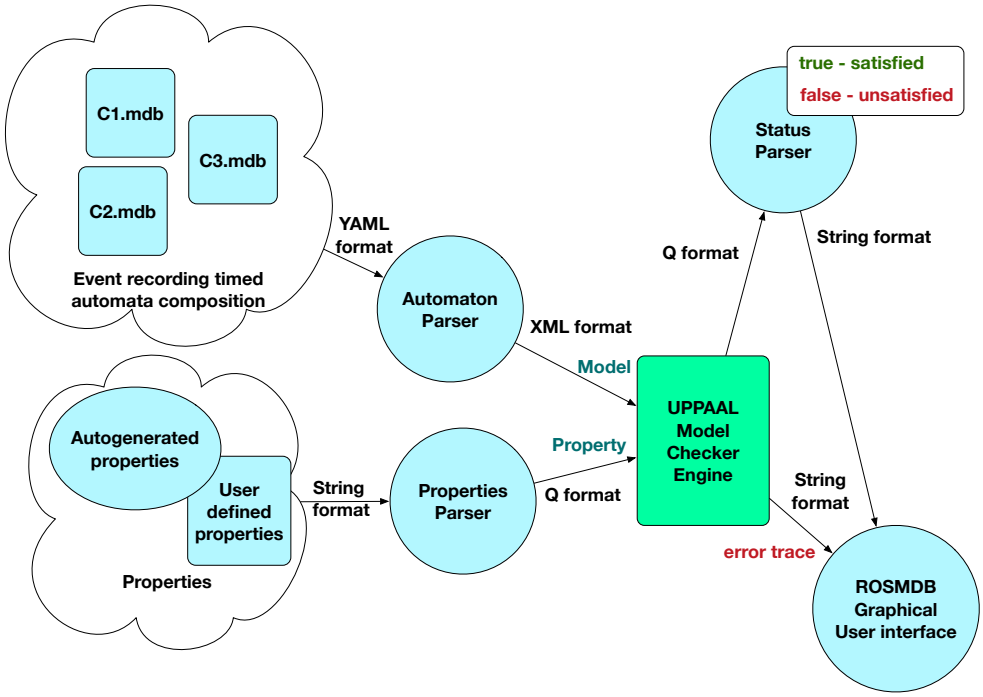


Figure 5.19 – Integration of ROSMDB with UPPAAL model-checker engine

In order for ROSMDB to use the model-checking engine, format change is needed. As shown in fig 5.19, the *ERA* of each automaton is automatically parsed in a *XML* format. The user inputted properties combined with the autogenerated verification rules are parsed into *Q*[11] format. At this step, *ROSMDB* injects into the *UPPAAL* model-checking engine both the model and the list of properties to verify. When the validation ends, a parser transforms the properties results from *Q* format to *JSON* format. Both the results and the error trace are displayed in *ROSMDB* front-end as shown in fig. 5.20.

The *ERA* model validation can be executed at any layer of granularity: starting from component validation at 4th layer, to service model validation by automatically making the product of components automata, towards entire application model validation. At each layer of the granularity, the user can select the number of components involved into the behavior. This allows the validator to mimic the use of multiple robots running the same application.

[11] A UPPAAL specific text format for properties

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

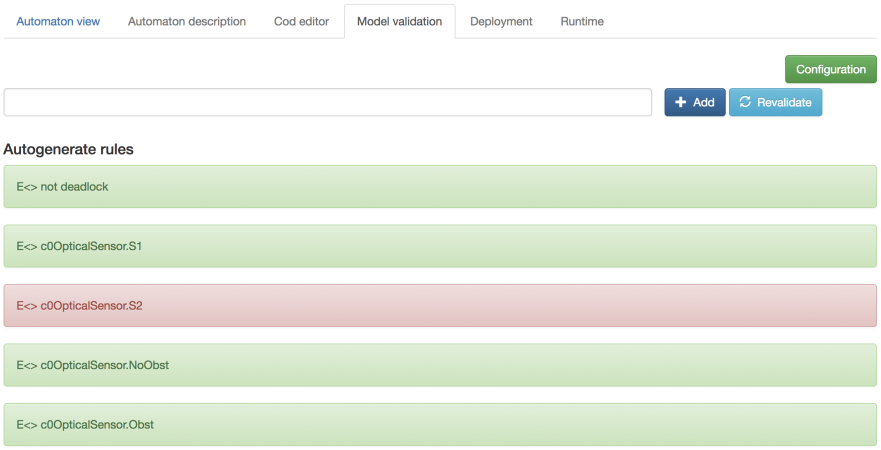


Figure 5.20 – ROSMDB model-validation front-end

When a new component *ERA* is designed, *ROSMDB* tool-chain autogenerates rules for the Validation step of the chain. These rules verify two kind of properties:

- liveness property: the rule queries the model checker for the existence of deadlocks.
- reachability properties: $\forall s \in L_{component}$, the rule verifies that $\exists$ a path p starting from $s_i \in L^0_{component}$ that can reach s.

These autogenerated rules, as well as the user inputted rules, are required to be in *UPPAAL* specific query language which is based on *TCTL*. The queries available in the verifier are:

- `E<>` p: there exists a path where p eventually holds.
- `A[]` p: for all paths p always holds.
- `E[]` p: there exists a path where p always holds.
- `A<>` p: for all paths p will eventually hold.
- `p -> q`: whenever p holds, q will eventually hold.

p and q represent state formulas that are logical combination of `<process>.<state>` and `clocks guards` (e.g. `OpticalSensor0.ObjectDetected and XObjectFound<1`). A special form `E<> not deadlock` that checks for deadlocks. This notation is the *ROSMDB* generated rule for liveness. The other autogenerated rules for reachability are formed on the pattern `E<> component.state` $\forall$ component $\in$ { product of components}, $\forall$ state $\in L_{component}$.

The autogenerated rules for the *OpticalSensor* are:

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

- E<> C0OpticalSensor.S1 - it exists a path from the initial state towards S1
- E<> C0OpticalSensor.S2 - it exists a path from the initial state towards S2
- E<> not deadlock - the model has no deadlocks

In addition to these generated rules, the user can also add their own TCTL rules like: *E[] C0OpticalSensor.S1 and XObjectFound<1* that translates into there exists a series of events where an obstacle will be detected in less than 1 time unit after a previous obstacle was detected.

This phase of development can be iterated any number of times, thus allowing for a refinement of the initial model. The user can go back to the design phase, improve the *ERA* model and verify again against the same or different properties. When the resulting behavior is correct, the model can advance in the lifecycle of the robotic application to development step.

5.4.4 Code generation

When applying a *MDD* methodology, the difficult part is to translate the model design to software code. The task is further complicated when the model evolves and the source code needs to be updated. Keeping the equivalence between the code and the model is a complex task that can be managed by *ROSMDB* at a component level.

In the development phase of a robotic fleet application designed with *ROSMDB* tool-chain, the equivalent *Python* source code is generated automatically. It presents itself as a *ROS* compatible model. Each component is associated with a *ROS* node and it is generated as a *Python* object class that extends a *Base* class from the *ROSMDB* framework. For each component, this is the only file that needs to be edited in order for the application to run.

The name of the generated skeleton class is similar with the component name. It consists of the various routines that will be executed automatically at runtime by the framework. An extract from a generated skeleton can be seen in listing 5.3. The routines are decorated with a *Python* decorator and their type can be:

- `@Transition` decorated routine: it is executed each time the associated event or symbol is recognized by the *ERA*. The header of the routine includes a *data* object that carries the eventual payload of the event message into the function. Those types of routines can return the event payload if the event should be triggered by the transition. The event push and pull are not managed by the routines themselves because these tasks are delegated to *ROSMDB* framework.
- `@State` decorated routine: it is executed each time the internal *ERA* translates to the associated state. This function should be edited with the specific code of the state. (e.g.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

for the optical sensor example, the routine associated with the state *S2* analyses the depth image and decides if an obstacle is present or not).

```
"""..."""
# This code was autogenerated. Please modifiy only the parts marked as #TODO

class OpticalSensor(BaseClass):
    def __init__(self):
        BaseClass.__init__(self)

    @Transition("?/camera/rgb/image_color-/camera/rgb/image_color")
    def listen_for_image(self, data):
        # @param - sensor_msgs/Image - the data for the callback function
          from the topic /camera/rgb/image_color
        # TODO
        pass

"""..."""

    @State("S1")
    def state_S1(self):
        # This function should be called when the state changes into this
          state
        # TODO
        pass

"""..."""
if __name__ == "__main__":
    runner = OpticalSensor()
```

Listing 5.3 – Example of code skeleton

The reader can remark in the listing 5.3 the presence of (TODO) metadata. The tool-chain only generates the skeleton of the component based on the states and transitions in the model and guarantees the equivalence to the model based on it. The source code of each transition or state function it is not generated and it is neither guaranteed for model equivalence nor taken into account for the model validation. The validation of the model at the runtime relies on the structure of the skeleton. If the functions headers are changed or remove, *ROSMDB* does not guarantee anymore the equivalence to the theoretical model. As already mentioned, this contribution looks only at the exchange of a model with the environment (sensors, actuators and software components). Even if the tool-chain allows the user to input the specific source code logic, it will only be used at runtime and not into evaluation of the model behavior.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

This feature allows not only for *MDD* methodology, but also reduces the time to develop a *ROS* node by removing from the user space redundant *ROS* code because all the nodes registration and listeners or pushers mapping are delegated to *ROSMDB* framework.

5.4.5 ROSMDB framework

ROSMDB framework is a self-contained *Python* package that allows the robotic application issued from *ROSMDB* tool-chain to be executed at runtime. It is designed to abstractize the model handling of an *ERA* in a *MDD* approach. Furthermore, it allows a transparent management of *ROS* nodes and *IP* network connections.

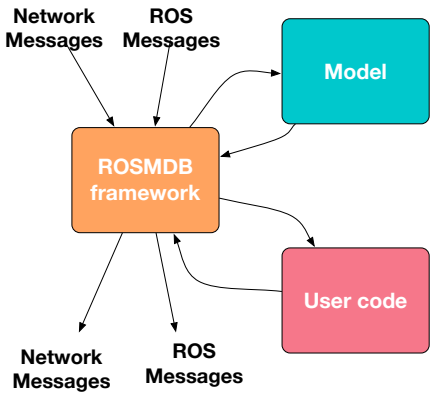


Figure 5.21 – User code interaction with ROSMDB

Figure 5.21 presents how the `framework` interacts with the `user code` of the robotic application that is based on the skeleton issued from *ROSMDB* tool-chain. An application developed and deployed with this contribution contains an automatically generated *JSON* representation of the `model` among the actual *Python* files of the user code. The `framework` loads transparently this configuration file and declares on-the-fly all the *ROS* nodes in the user code. Moreover, the `framework` registers all the network listeners. Each function call in the `user source code` triggers a transition and a state change in the `model`. This mapping between the `model` and the `user source code` is transparent since the `framework` is managing all the interactions. In general, a transition can happen when a `message` arrives or when it needs to be published. The arrival of a `message` (*ROS* or *network* message) fires a function call in the `user code` as well as a state change in the `model`. If the `user code` needs a message to be published, it will forward it to the `framework` which will send it on the right communication channel. The `framework` will also reflect this in the current state of the `model`.

ROSMDB framework makes the link between the user robotic application, *ROS* and networking service of the operating system as shown in fig. 5.22. This package needs to be installed on each robot that is included in the projects developed with *ROSMDB* tool-chain. Internally, the framework is composed as follows:

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

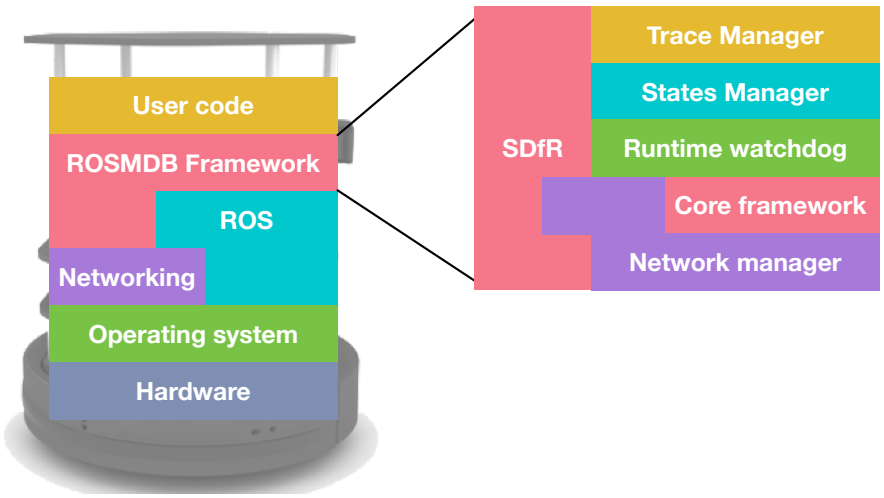


Figure 5.22 – ROSMDB framework architecture

- `Trace manager` is logging all the events that happen at runtime. It inserts data collection probes in the stack execution of the application via *Core framework* in order to provide the feedback for the tool-chain. It also relies on *SDfR* integration in order to provide these traces back to the *ROSMDB* tool-chain.
- `State manager` is handling the memory representation of each components *ERA*. It maps each function from the user code space to the model and keeps track of the transitions based on the memory stack evolution of the components runtime. It is also used to determine the current state of the system by the *Runtime watchdog*.
- `Runtime watchdog` decides to trigger an alert based on the information from the model description and the current state of the system. It uses the traces generated by the *trace manager* and relies on *SDfR* to push in real time alerts to *ROSMDB* toolchain *GUI*.
- `Core framework` is orchestrating all the internal components together. It is the main communication point between the framework and the user code space because each of the components skeleton extends a base class from it. It allows messages to be send to and from the user code to other components. Furthermore, this framework registers all the *ROS* nodes and performs the data exchange between *ROS* topics and/or services and the robotic application.
- `Network manager` is managing all the network connections of the robotic application. Transparently for the user, it sets up listening workers on the ports specified in the model. When a network message arrives, it pushes the data to the user code space and calls back the function configured in the model description with the help of *Core framework*. If the user space needs to broadcast a message, the *network manager* receives the payload from the *Core framework* and broadcasts to all reachable peers (i.e. all the neighbors visible through *SDfR* protocol) in a point-to-point mechanism exchange in order to avoid flooding the network.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

- `SDfR`, the robotic service discovery protocol presented in chapter 4, integration is registering each service on the fleet network and allows the framework to have a list of reachable peers.

Each state transition is monitored at runtime by *trace manager* via collection probs. The main reason for this logging system is to allow the user to analyze the runtime behavior of the application and compare it to the validation via model-checker. An example of a trace can be found in listing 5.4. The *state manager* can reset the associated clocks of the event (the reader should remind that the behavior model is an *ERA*, thus when a symbol is recognized, the associated clock is reset) based on the trace.

```
'symbol': '/camera/rgb/image_color', # - The recognised symbol
'current_state':'S1', # - The current state when the trace was recorded
'destination_state': 'S2', # - The desired destination state
'clocks_value': 0.345465,  # - The value of associated clock
'time': 1510586388.771781, # - The unix time when the trace was recorded
'is_valid': True, # - Is the transition possible ? Updated by the watchdog
'error': None # - Does the clock violates any guard? Updated by the
  watchdog
```

Listing 5.4 – Example of transition trace

The alerting system pushes alerts in in real time, thus the developer station will receive the message if it is visible in the *SDfR* neighbours table. There are two types of alerts:

- `Time guard violations`: This logging of each state switch allows the *watchdog* to measure the value of the clocks associated with transition and to trigger an alert if the guards form the model are violated.
- `Transition violations`: Coupled with the *State manger*, the *watchdog* is capable of detecting if incorrect transitions (i.e transitions that are not designed in the model) are produced due to the arrival of an event. In this case, it does not block the execution, but it triggers an alert.

5.4.6 Summary of SDfR usage

All the subcomponents in *ROSMDB* framework relay on *SDfR* to provide information about reachable neighbors [Chitic et al., 2016]. The applications designed and developed with *ROSMDB* tool-chain are running in a fleet context where each service of these application needs to be advertised in order to exchange data with peers.

To allow the framework to register the services on *SDfR* protocol, each of the service needs to be described in the tool-chain *GUI* as shown in fig. 5.23. This metadata is added to each

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

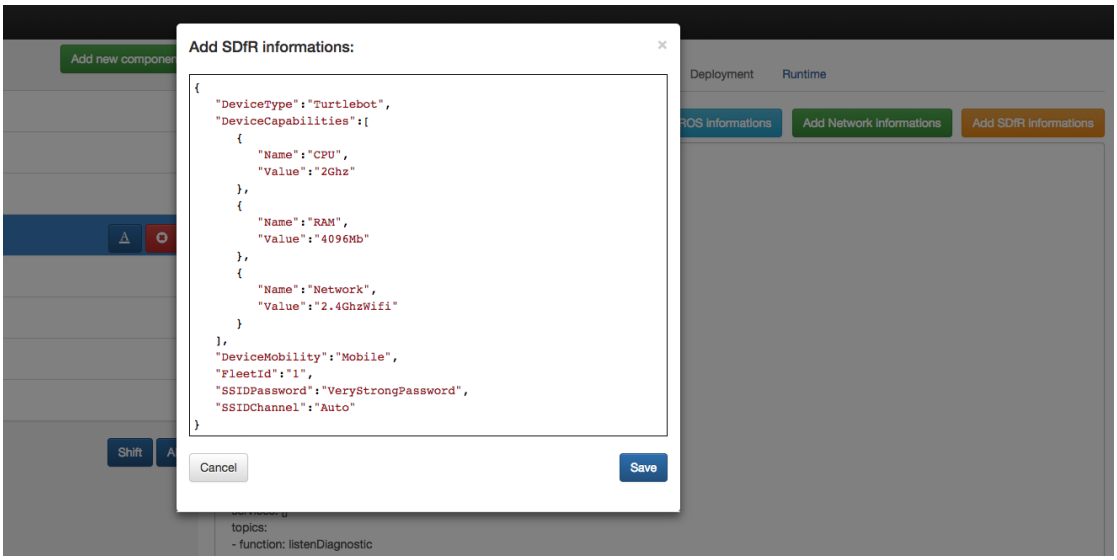


Figure 5.23 – SDfR metadata input

service *ERA* and packed together with the source code. At runtime, the *SDfR* integration of the framework will automatically fill the missing information and register the service. It also provides the support and manages the auto-description *URL* needed by *SDfR*. Finally, it maps each of the *SDfR service REST* endpoint in order abstractize the usage of *SDfR* protocol by the *ROSMDB* framework and by the user code.

SDfR is also used by the *ROSMDB* tool-chain *GUI*. The applications running on the fleet robots need to push the alerts to the developer station (if it is visible in the *SDfR* neighbors table) in real-time and all the collected traces at the end of the mission (i.e. runtime end). Moreover, the tool-chain provides an application deployment and execution scheduling system that relies on *SDfR* to provide the reachable fleet robots.

5.4.7 Fleet deployment

When testing a multi-robot application, deploying it in order to have a real test is very time consuming since each robot of the fleet needs to be provisioned with new software version. In our vision of using *MDD*, we propose multiple iterations in order to refine an initial model. Once an iteration is developed, it needs to be tested in a real environment in order check if the runtime behavior is similar to the modelled one. In order to reduce the time to deploy the new version, *ROSMDB* tool-chain propose an automatically deployment feature.

The automatically deployment feature packages each of the component source code and the model metadata in a tarball [12]. The user can deploy directly this archive to the reachable robots from the fleet as shown in fig. 5.24. The discovery of such robots is done by using

[12] In computing, tar is a computer software utility for collecting many files into one archive file, often referred to as a tarball, for distribution or backup purposes.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

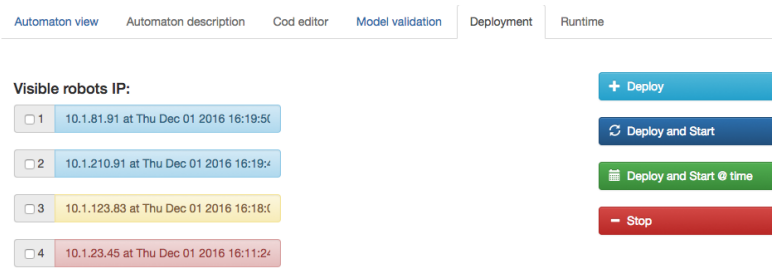


Figure 5.24 – Deployment interface in ROSMDB tool-chain

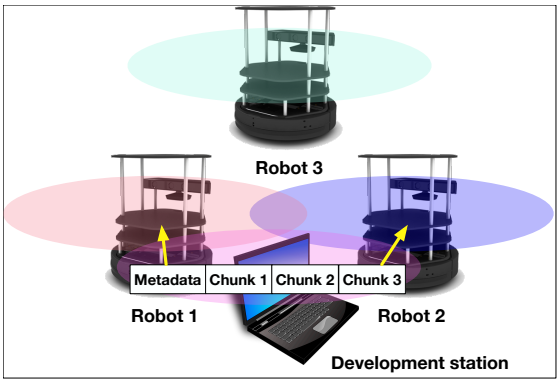
the tool-chain integration with *SDfR*. The system is tracking when the last time a robot was reachable in order to filter out unreachable robots. As a disclaimer, we are not taking into security and integrity of the packages. We are aware of the security vulnerabilities but the solutions to these problems are out of the scope of this chapter.

The fleet deployment component has an add-on feature that allows the developer to trigger the execution of the application immediately after the deployment or it can be scheduled to start at later time. This metadata is sent to all robots with a higher priority than the tarball. The tool-chain also includes a mechanism to abort the execution of a user code directly from the *GUI* that uses *SDfR* in order to directly command any reachable robot from the fleet.

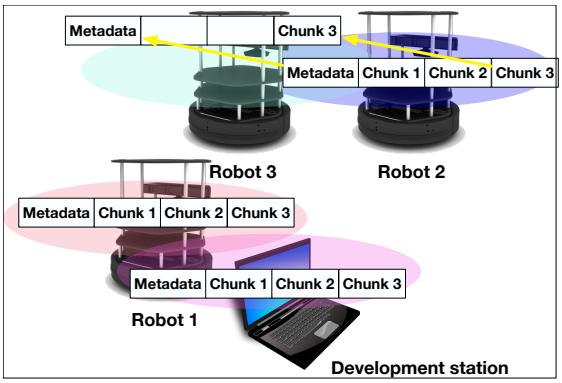
The tarball and metadata transfer is taking into account the mobility of the robots and is designed to exchange the payload in a chunk-by-chunk approach as shown in fig. 5.25. When the deployment starts in fig. 5.25a, the robots *Robot 1* and *Robot 2* are in the WiFi communication area of the development station. They receive the new package and the metadata before they start moving. At $t = 5$, in fig. 5.25b, *Robot 2* discovers *Robot 3* via *SDfR* and starts sending the new package. First, the metadata is sent with the highest priority. *Chunk 3* of the package is the only part sent because the *Robot 3* moves outside of the communication area. At $t = 10$, in fig. 5.25c, *Robot 3* encounters *Robot 1* and continues to receive the missing chunks of the package.

This mechanism allows for a fast propagation of the new version inside of a fleet even if not all the robots are in communication with the development station. This process combined with the mechanisms of designing *ERA* based applications, developing and executing the associated code allows *ROSMDB* tool-chain to be fully integrated with a robotic application lifecycle. A model can be designed, verified, developed, deployed and executed. There is still the need to compare the executed software and its behavior to the theoretical model. In order to answer this issue, *ROSMDB* tool-chain proposes a runtime validation feedback tool.

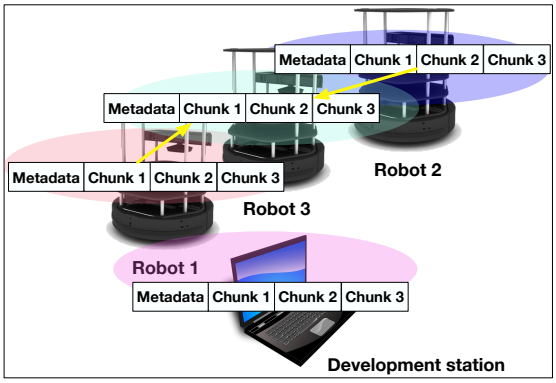
Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés



(a) Step t=0



(b) Step t=5



(c) Step t=10

Figure 5.25 – ROSMDB fleet deployment process

5.4.8 Runtime validation feedback

When a mission is executed by running a multi-robot application using *ROSMDB* tool-chain, the user has the possibility to retrieve traces of what happened during the runtime as well as alerts in the *GUI*. As shown in fig. 5.26 and in fig. 5.27, the alerts monitor shows a complete image of the system when the event happened. This includes information about the source state and the destination state of the transition associated with the event and the value of all the *ERA*s clocks.

The information provided by the monitor includes a global image on the number of correct transitions compared to the number of switches that violated the guard and the percentage of visits in each state. Furthermore, this monitor includes measurements of the evolution of internal clocks and the delay between the recognition of the same event. Even-more, the system is computing the average and the standard deviation of each event symbol for the current execution and for the total execution of the code. All the data is pulled from each robot reachable via *SDfR* protocol and displayed in a per robot view.

Let's go back to the example presented in 5.2.1. The component *Optical sensor* was modelled,

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

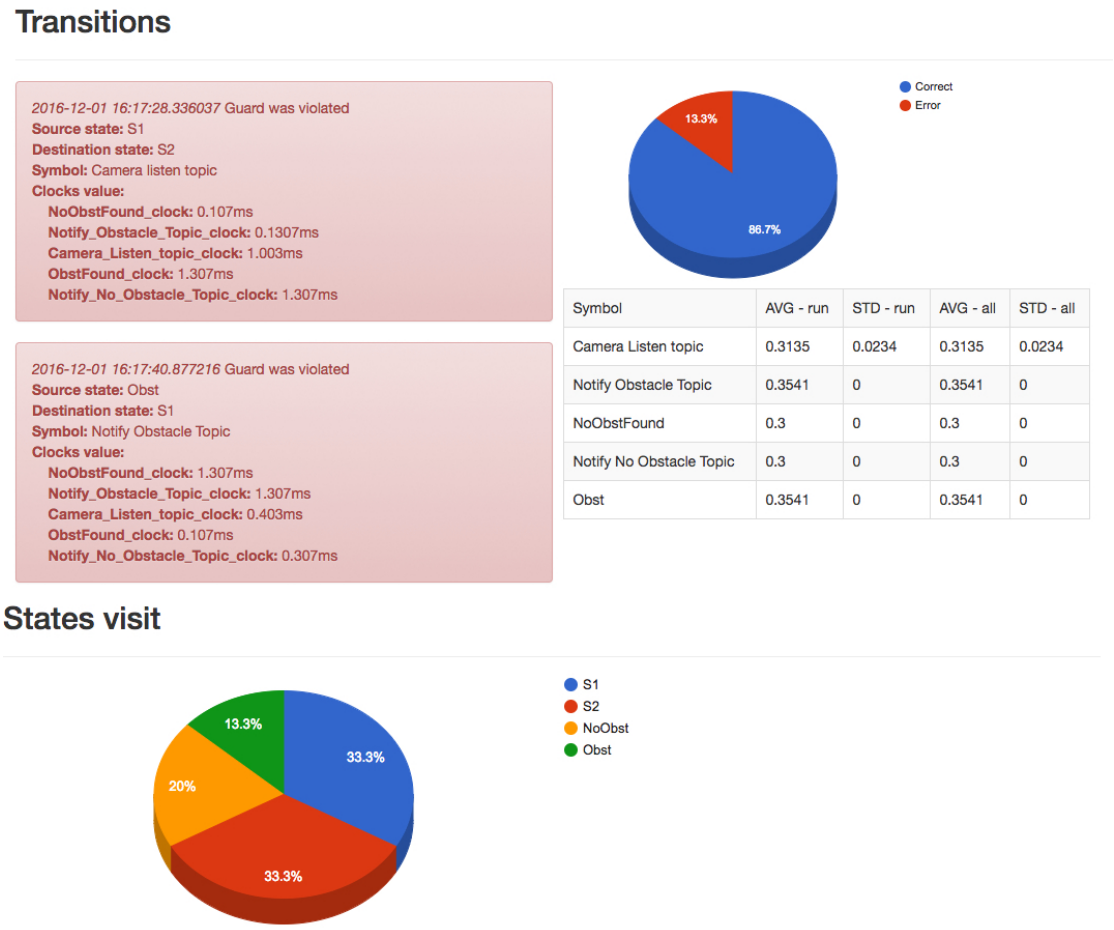


Figure 5.26 – ROSMDB trace feedback for example A (part 1)

verified, developed and deployed using *ROSMDB*. During the model checking step, all the reachability properties (i.e. every state is reachable) and liveness property (i.e. there is not a deadlock) were fulfilled. The results of the traces collected at runtime are displayed in fig. 5.26.

The reader should first notice the two alerts that were displayed. The first violation happens on the transition from state $S1$ to $S2$ because the clock associated with the *Camera listening topic* of the optical sensor overpassed 1 second. Being in an *ERA*, this clock only resets when a new image arrives. The model did not take into account that while processing the image for obstacles, new arrivals are suspended. Even if the optical sensor is providing the image at a constant rate, the time analysis of the image in $S2$ may vary due to the *CPU* usage, thus the transitions from $S2$ to $S1$ can overpass 1 second which makes the clock associated with *Camera listening topic* to violate the guard. However, the average value of the clock is 0.3135 seconds which makes this alert a rare event.

The second alert concerns the detection of an object. In the model, this transition from $S2$ to $S1$ (via *Notify Obstacle found*) when an object is found should happen after 1 second since the

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

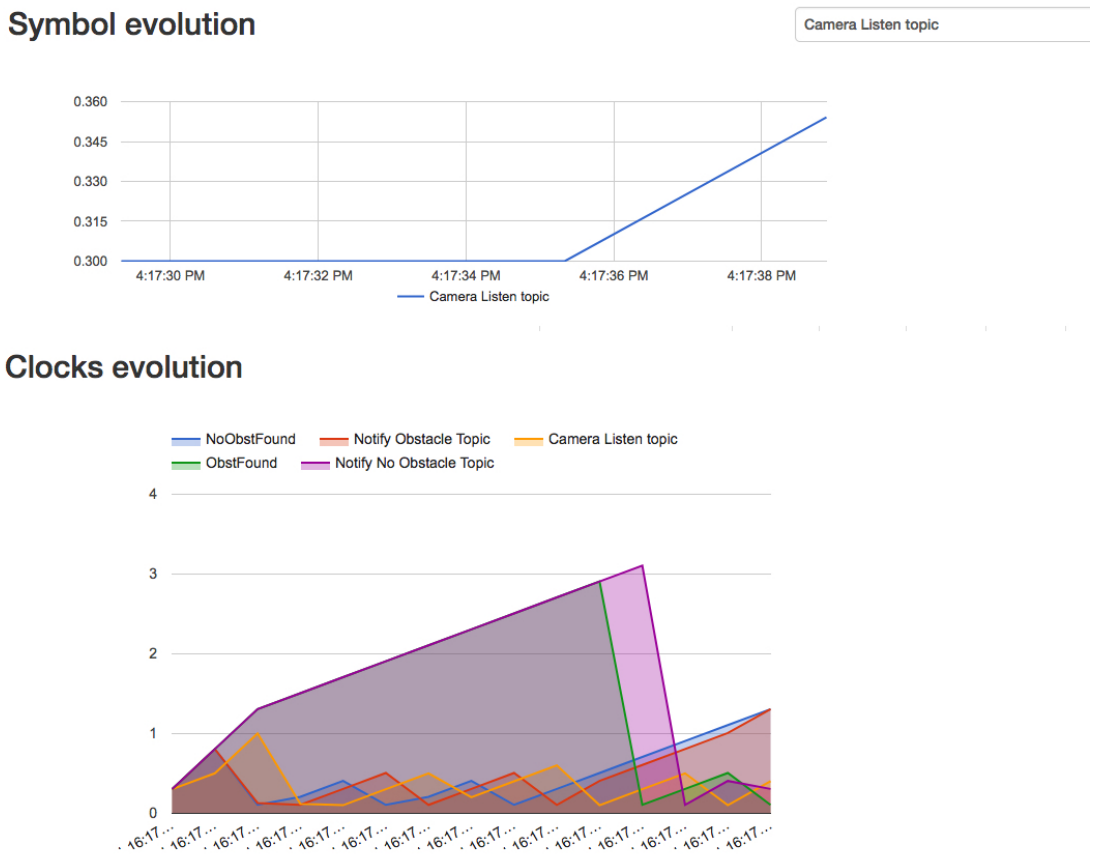


Figure 5.27 – ROSMDB trace feedback for example A (part 2)

last object was found. The average of this clock value is 0.3451 seconds at the runtime which makes this violation to happen every time an object is found because the clock will always be reset. This means that the initial model has a flow: the frequency of the object detection is greater than the ability of the robot to move in order to avoid the object, so the same obstacle is present in the next loop of the component.

The kind of analysis presented above can be applied by the user based on the data collected at runtime to compare the validated *ERA* in the model checker with the actual behavior of the application at runtime. In this way, guards can be refined. The upper limit of a guard can be modified from the theoretical value to the average of the real value. Moreover, the lower limit of the guards can be estimated in order to avoid deadlocks and unpredicted behavior of the system.

This mechanism of refinement can be done in iterative steps that bring back the lifecycle of the application from runtime to design. From this later phase, the application goes again through *ROSMDB* tool-chain in order to obtain the same behavior both in the model-checker and at

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

execution time. These iterations, called sprints in software development, allow to define the finite multi-robotic application by granularly improving the behavior of an *ERA* based model using *MDD* in a *SOA*.

5.5 Summary

This chapter presented the challenges to define a complete tool-chain to develop *MDD* multi-robot application over a *SOA*. The most appropriate formalism to design such application is, in our opinion, timed automata. In this context, we have proposed *ROSMDB* that interacts with the entire lifecycle of a multi-robot application: design of the behavioral model, theoretical validation using timed automata formalism, code development, application deployment and runtime monitoring and feedback retrieval.

Furthermore, a *ROS* compliant framework has been proposed that allows the user to focus less on *ROS* and networking modules development. The *ROSMDB* framework allows for transparent states and transition manager and lets the user to only define the specific code logic for the multi-robot application.

In order to validate our proposal, we have defined a series of scenarios and we have implemented and benchmarked sample applications of these scenarios with *ROSMDB*. The following chapter presents our results.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

6 ROSMDB: Experimentations

This chapter presents a series of case-studies and sample experimentation of parts of these case-studies in order to validate the usage of Robot operating system Model Driven Behavior (ROSMDB) *and how errors that can be present at runtime even if everything is correctly validated at design phase, can be detected via* ROSMDB.

6.1 Package delivery by drones swarm 125

6.2 Guest welcoming and management with intrusion detection system 132

6.3 Summary 142

In these chapter, the reader is presented first with a series of case-studies that involves complex multi-robot software projects composed each one of multiple applications. These scenarios are: *Package delivery by drones swarm* and *Guest welcoming and management with intrusion detection system*. We present the experimental results of our implementation of sample robotic applications that answer to at least one of the problems (i.e *Flight synchronization based on N pole* for the first scenario, *Random movement object search* for the second one) from these projects using *Robot operating system Model Driven Behavior (ROSMDB)*.

6.1 Package delivery by drones swarm

In 2013, Amazon announced its intention to deliver packages by drone[1] in a US TV show. At this time, it was suggested that the company would begin delivery in 2018 which started a large number of concerns around theft, liability and safety. What is more interesting is that they

[1] https://www.amazon.com/Amazon-Prime-Air/b?node=8037720011

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

are designing drones capable of delivery payloads up to 5 pounds (2.26 kg)[2] which represents small parcels.

In the case of large parcel delivery, the total weight of a parcel can be devised by using a swarm of drones instead of only one. In this case, only of a fraction of the weight will be assigned to each drone. This multi-drone project needs to handle not only the navigation inside urban areas, but also the coordination mechanism between drones in order to load, fly and deliver a parcel.

First, the drones need to be assigned a fly plan from centralized system which computes it based on the total weight and distance. Based on this computation, the exact number of drones needed for the flying will take off from their base station to the loading system.

The loading system will assign each drone an exact position where to hook their payload cables. After this process is done, the drones will fly in a coordinated way towards the delivery point and will unhook their cables at the destination.

This complex fleet of drones project faces several challenges which can each be expressed as a multi-robot problems:

- Flight synchronization
- Localization and navigation of a flight plan
- Network communication in a wide area network (i.e. networks that cover a city or a region; e.g. 3G/4G networks)
- Decision making based on environmental changes
- Local synchronization with external systems to the fleet.

In the next subsection, we have designed, developed and analyzed using *ROSMDB* one of the application from this project: *Flight synchronization based on N pole*.

6.1.1 Flight synchronization based on N pole: Description

One of the applications needed by the case study *Package delivery by drones swarm* is represented by a coordinated formation flight of drones in an indoor environment like a wear-house. In this case, the *Global Positioning System (GPS)* position is unavailable. Furthermore, this system is not highly reliable even in outdoor flights since its accuracy is over 1m [Shepard et al., 2012]. To solve this issue, we have experimented in *ROSMDB* a formation flight based on the detection of the North Pole which can be done in both indoor and outdoor environments.

[2] https://consumermediallc.files.wordpress.com/2015/04/amazon_com_11290.pdf

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

The experiment provides an application that runs on a fleet of Parrot Bebop drones developed using *MDD* over a *SOA*. Its purpose is to allow drones to take off simultaneously, synchronize their orientation based on the North Pole and fly for a given distance synchronously.

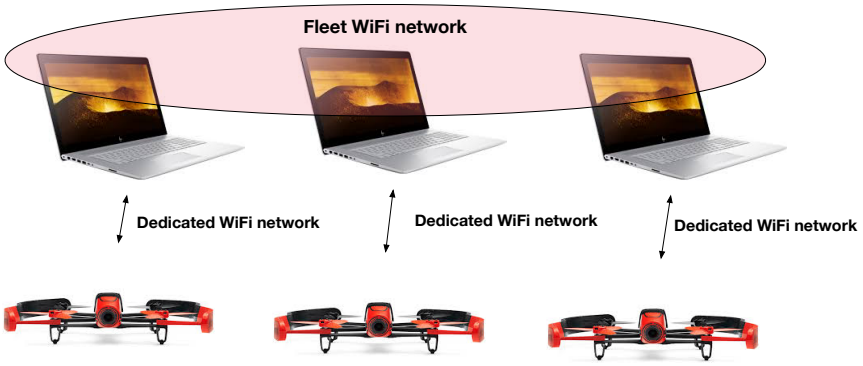


Figure 6.1 – Fleet of drones network

In order to test this scenario, we have used 3 Bebop drones[3]. Since the embedded software on the drones only allows for data exchange over *HTTP* over a point-to-point *WiFi*, the experiment includes a controller pc for each drone as shown in fig. 6.1. Each drone is connected to its controller PC via a dedicated *WiFi* connection and all the laptops are connected in a fleet network. In this case, each laptop has 2 network cards. The benchmarks were performed on laptops equipped with Intel Core 2 Duo, 2.1 GHz CPU, 4Gb of Ram, WiFi enabled (supporting Ad-Hoc networks) running on Ubuntu 13.04.

The communication between the laptop and its corresponding drone is managed by *ROS* Ardrone package that is converting *ROS* messages to *HTTP* messages and proving the data exchange. For this case, the tuple drone-laptop is represented as a single robot. The application developed with *ROSMDB* is hosted on the laptop and fleet network is represented by the laptops network.

6.1.2 Flight synchronization based on N pole: Models

In order to model this scenario, we propose 2 applications: a *controller application* that will be used on a separate laptop in order to trigger the execution of the mission and a *Commander application* that manages all the commands for the drone, the fleet network communication and the synchronizations between the drones. A series of screenshots from *ROSMDB* corresponding to this scenario design, validation, development and feedback can be found in Appendix D.

The *Controller application* is composed of only *Command sender service*. As shown in fig. 6.2, it waits for a keyboard command to trigger the mission start using the networking service.

[3]https://www.parrot.com/fr/drones/parrotbebopdrone#in-the-box

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

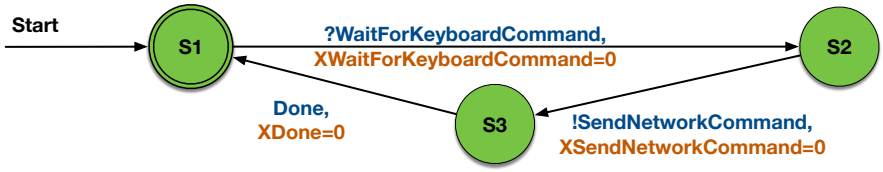


Figure 6.2 – Command sender Model executed on a separated control laptop

The *Commander application* is composed of several services:

Networking service is managing all the communications between drones (i.e. the laptops corresponding to each drone). The model is represented in fig. 6.3. The models waits for a local command in *S*2 and it broadcasts it using *SDfR* over the network.

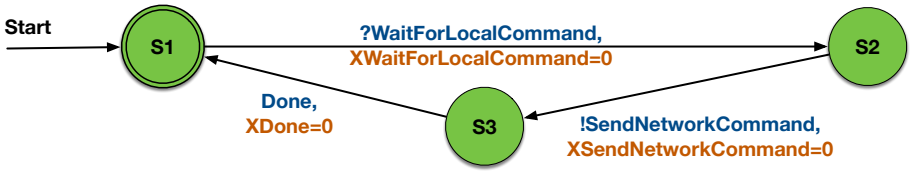


Figure 6.3 – Networking Model executed on each drone assigned laptop

Take off manager service is managing the synchronous take off of all the drones and their stabilization before the coordinated flight. The model is represented in fig. 6.4. In state *S*1, the model waits for a network command coming from the controller pc to start the mission and when the command is given, it will trigger the *take off* command in *S*2 and notify the *Lock North* service in *S*3.

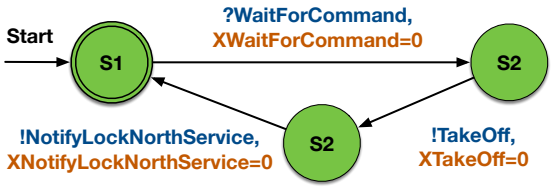


Figure 6.4 – Take off Model executed on each drone assigned laptop

Lock North manager service is taking care of drone's position in order to face the North Pole. Each drone waits for the other drones to finish locking the North pole before the flight in formation. The model is represented in fig. 6.5. The model is waiting for the take off command to be finished in *S*2 and starts the process of rotating the drone. In *S*2, it waits for a position change in altitude or in rotation. This information is requested by the source code on the laptop from the drone internal *CPU* over *HTTP*. The model needs to wait for the response to the request to be propagated. Then it checks if the angle between the current position and the North Pole is 0 ($+/-\epsilon$ since the angle is cast to an

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

integer). If the angle in $S3$ is not 0, it switches to $S4$ where is stops the current movement of the drone and applies the new rotation in $S5$. The rotation movement ends when the angle in $S3$ is $0(+/-\epsilon$ since the angle is cast to an integer). The *Movement* service of each drone is notified that the drone faces North in the transition between $S6$ and $S1$.

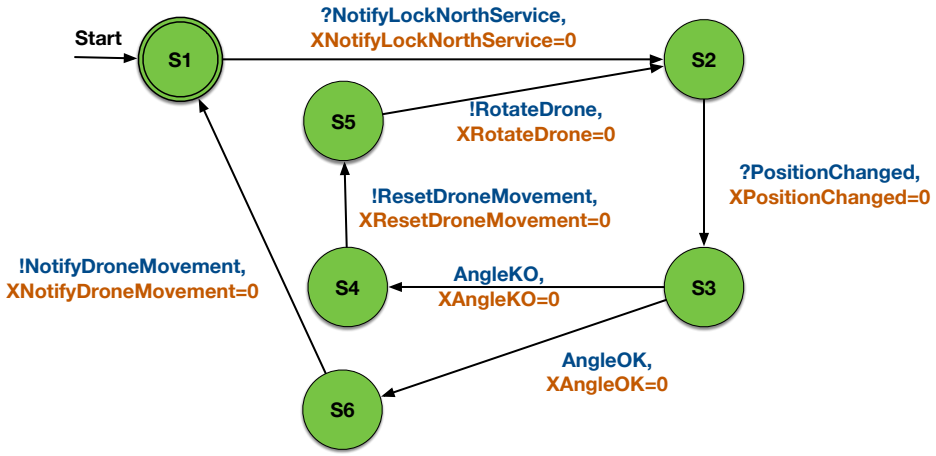


Figure 6.5 – Lock North Model executed on each drone assigned laptop

Drone Movement Manager service is executing the synchronous flight of the drones for a linear distance of 2m. When the drones reach their destination, the service triggers the landing command of the drones. The model is represented in fig. 6.6. The model waits for the north pole to be locked in $S1$ and starts the drone movement in $S2$. After a movement command is applied in $S2$, the model passes through the states $S3$, $S4$ and $S5$ where it waits (i.e. blocking states) for information about drone speed, altitude and position. In $S6$ it computes the position of the drone and resets the movement if the final position is not reached in $S7$. If the final position is reached, the model triggers the landing command and notifies the system in $S8$.

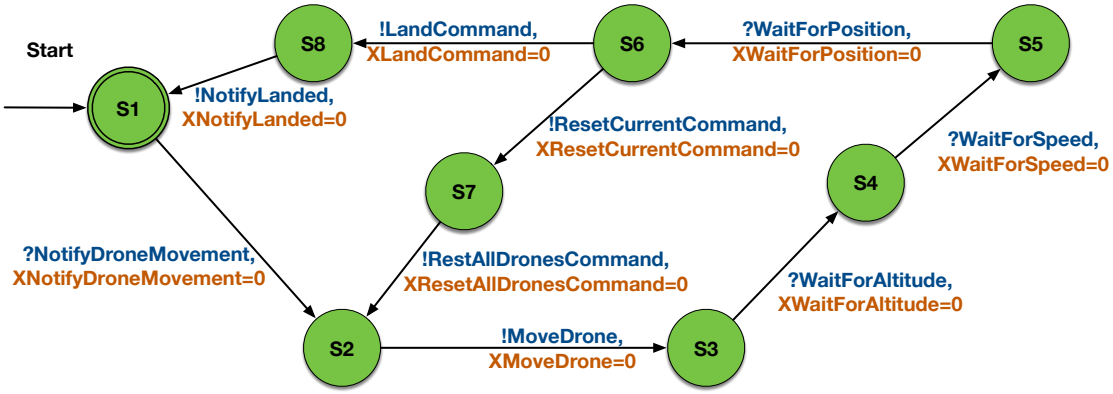


Figure 6.6 – Drone Movement Model executed on each drone assigned laptop

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Before filling the python skeleton, all the models and their composition were validated using *ROSMDB* model checker component. The reader may find in appendix D a series of screen-shots from the tool-chain which includes also the validation of each model. We have checked if all the states are reachable from the initial state as well as if the final state is reachable (i.e. reachability properties) and for *deadlocks* in the models (i.e. liveness property). Below are listed some of the properties that were validated:

- E<> not deadlock *(liveness property)*
- E<> c0DroneMovement.GetAltitude *(reachability property)*
- E<> c0DroneMovement.GetSpeed *(reachability property)*
- E<> c0DroneMovement.stopDrone *(reachability property)*

The reader should notice that in all the previous models, no time guard is set. In this case, it was difficult to set a time guard since we could not anticipate the behavior of the system.



Figure 6.7 – Experimenting Flight synchronization with Parrot Drones

6.1.3 Flight synchronization based on N pole: Experimental Results

The results in the first iteration of the applications give us an idea of how the time evolves between each state. The reader can watch a movie of the experimentation at https://www.youtube.com/watch?v=BqvcCYOnyGs. Figure 6.7 shows a in illustration of the experiment.

After running the first iteration of the application 10 times (each iteration is defined below as a Run), we have noticed that the inter-arrival time of events for simple services like *Take off manager* or *Networking* is constant. This was not the case for more complex services where a full interaction with the drone was needed.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

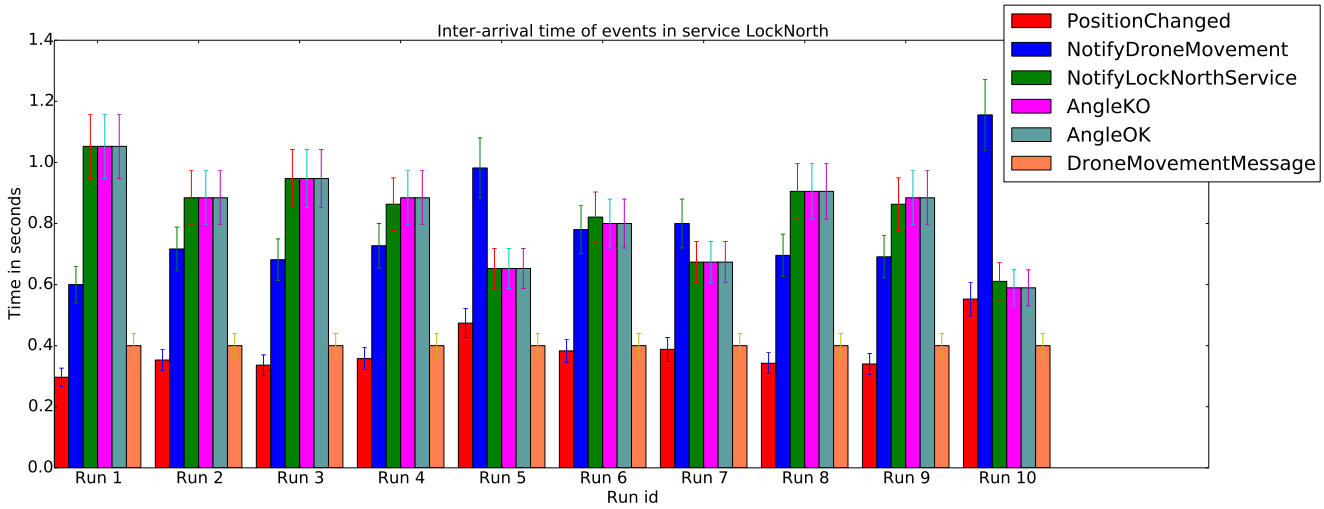


Figure 6.8 – Lock North messages inter-arrival time

For the *Lock North* service, as shown in fig 6.8, the notification message from the *taking off* service arrives (i.e. arrival of NotifyLockNorthService) with an average of 0.7368 seconds which means that the time for a drone take off depends on the drone hardware parameters like battery, position, luminosity[4], etc. Another interesting event is the *PositionChanged* which arrives with an average of 0.4 seconds. This value shows us the time to collect the information from the drone. The fact that is not constant can be explained by the workload of the *CPU* of the drone. The internal symbols *AngleOK* and *AngleKO* are recognized under the same time per run with an average of 0.8 seconds. The *DroneMovementMessage* is almost constant at an average of 0.3914 seconds. The final message after the *north pole* is locked, is sent with an average of 0.7789 seconds, meaning that this service is running in average 0.7789 seconds in order to lock the north pole.

In the case of the *Drone movement* service (see fig. 6.9), we can see that the inter-arrival time for the information coming from the drone is almost constant within multiple runs: *WaitForAltitude* has an average of 1.2475 seconds, *WaitForSpeed* has an average of 1.3745 seconds while *WaitPosition* has an average of 0.6475 seconds. The average for a *DroneMovement* message to be executed is 0.3158 seconds. Those values show us that the time to get information from the drones via *HTTP* is, on average constant.

These results allow us to refine the initial model by adding the time guards to our initial model. This initial application allows us to test the usage of *ROSMDB* as a complete tool-chain to develop a multi-robot project.

Figure 6.10b shows the refined model for Networking service (fig. 6.10a) and for Take off service (fig. 6.10b) after 3 iterations of the entire life-cycles. In fig. 6.10 the reader can notice

[4] A bebop drone uses a camera to detect the altitude and its position

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

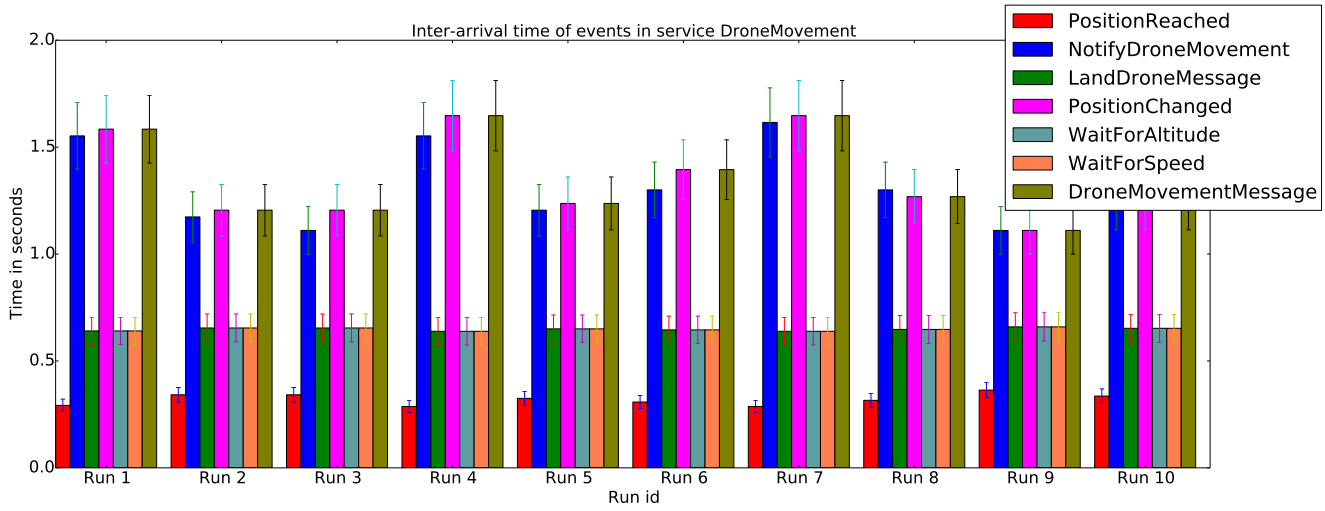
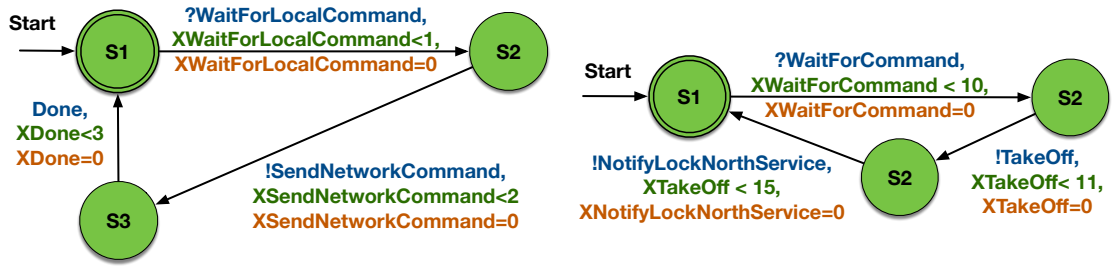


Figure 6.9 – Drone movement messages inter-arrival time

the refine model for Lock North service while in fig. 6.11 it is displayed the refine model for Drone Movement service. Compared to the initial versions, all the models have been updated with time guards that were extracted form the run-time observations.



(a) Refined Networking Model executed on each drone assigned laptop

(b) Refined Take off Model executed on each drone assigned laptop

6.2 Guest welcoming and management with intrusion detection system

In this section, we present a hypothetical example of the problems of a guest welcoming and intrusion detection system operated with a fleet of robots. Let's take the example of a nuclear research facility, like *CERN*[5]. The organization is located in two main campuses and several remote complexes. These campuses house a large number of buildings. The complex is highly secured with different access zones. Each employee has different access clearances based on its role and on its qualifications. But the organization needs to handle a lot of external

[5]The European Organization for Nuclear Research, known as CERN, is a European research organization that operates the largest particle physics laboratory in the world.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

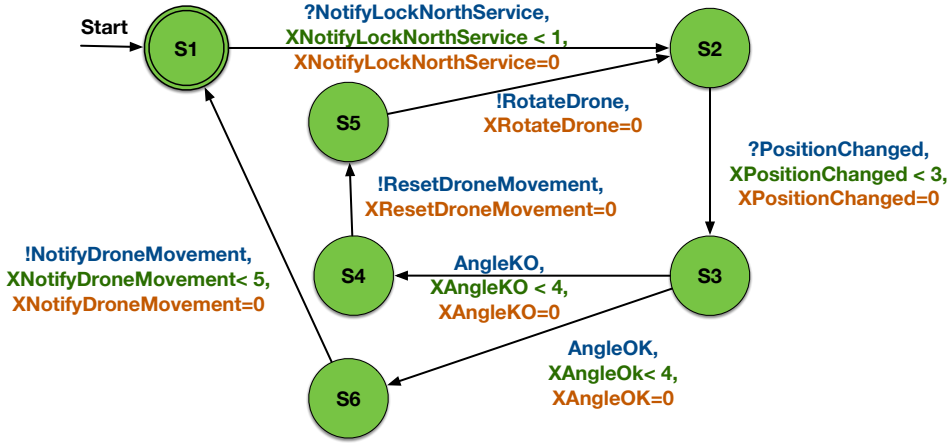


Figure 6.10 – Refined Lock North Model executed on each drone assigned laptop

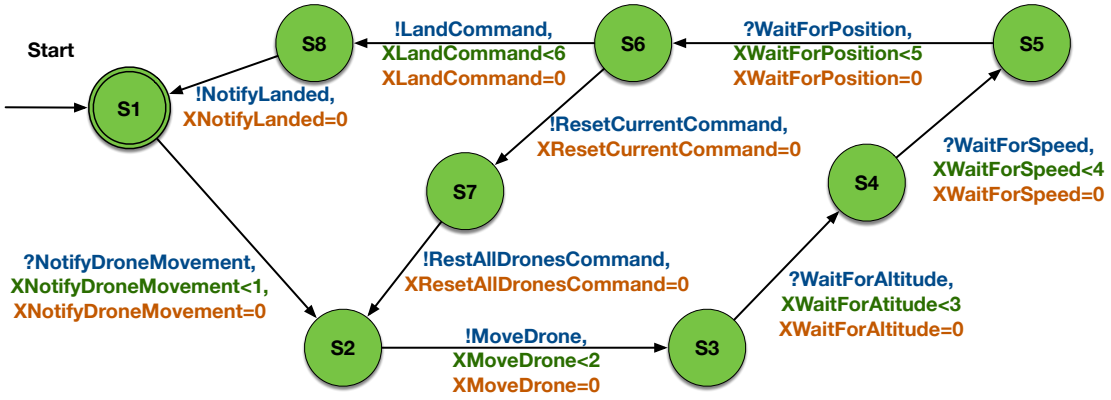


Figure 6.11 – Refined Drone Movement Model executed on each drone assigned laptop

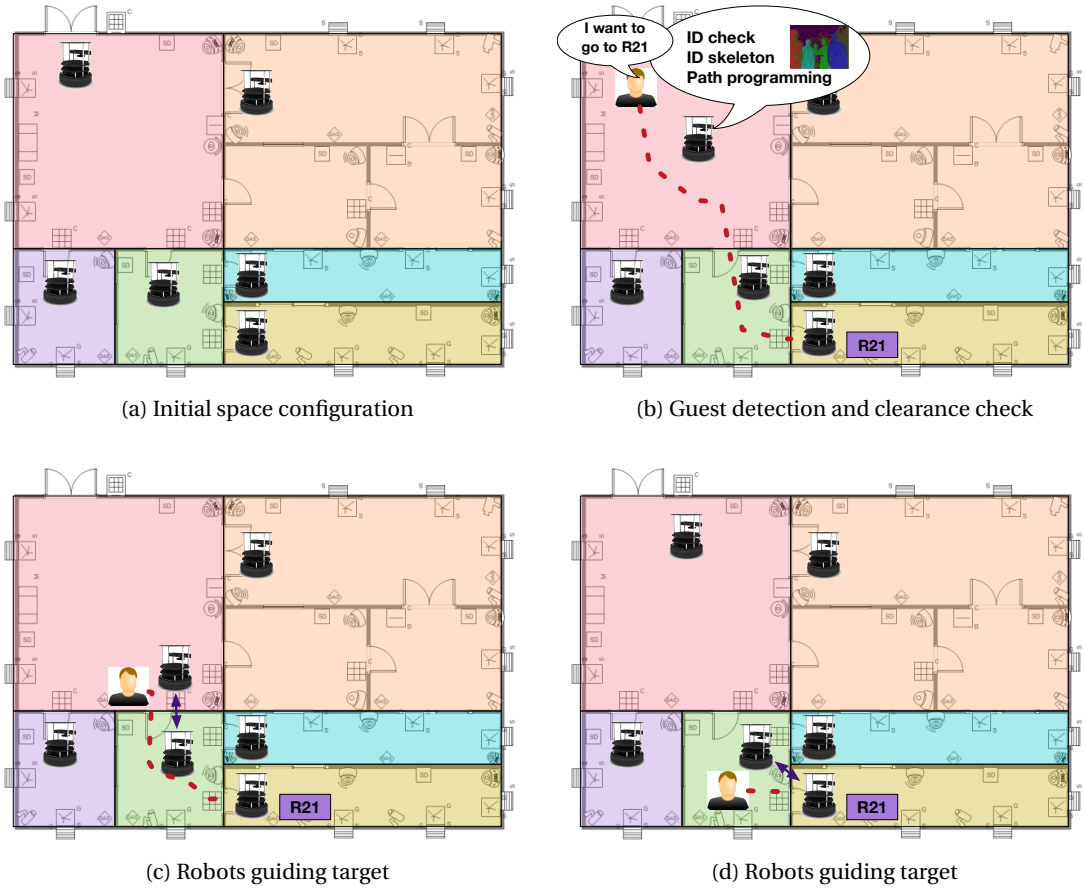
interactions because it is a research facility with collaborations all across the world and it has several public sights inside the campuses where visitors can view the results of the research and the infrastructure in use. This complex task of guiding and handling access to those external people inside the campuses costs the organization more than 9.000.000 CHF in 2017[6].

In our vision, such complex and costly guest management could be fully automatized using a guest welcoming and management with intrusion detection system. The main role of the system will be to create a human-machine interaction and to be able to guide guests in a complex environment. The environment is separated in different access zones and credentials need to be checked. Furthermore, the system should be able to detect undesired intrusion in such restricted areas.

In the design of the system, we make some assumptions. First, the complex needs to be large enough in order to be able to deploy a fleet of robots. Examples include large warehouses,

[6]CERN budget for 2017: https://cds.cern.ch/record/2240771/files/English.pdf

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés



(a) Initial space configuration

(b) Guest detection and clearance check

(c) Robots guiding target

(d) Robots guiding target

Figure 6.12 – Guest handeling

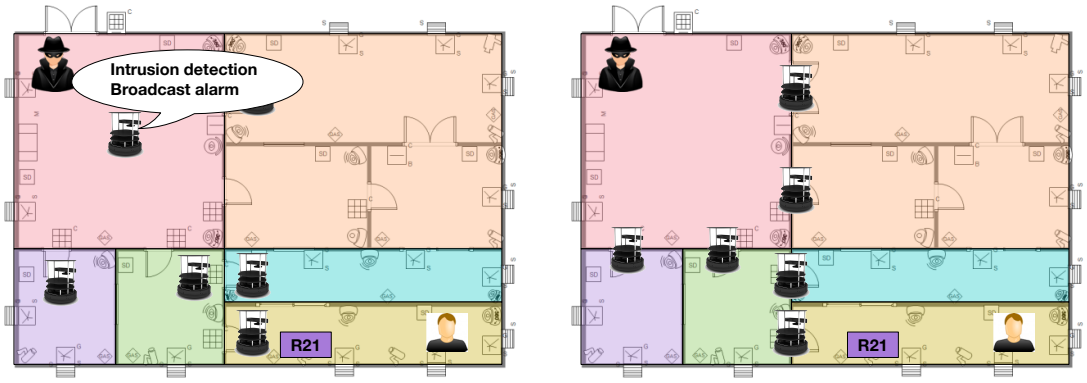
medical centers, university campuses, secured banks or military facilities. The space needs to be devised in several security clearances protected areas and all the accesses between zones should present smart enabled doors (e.g. NFC door openers, biometrical openers, etc.). Finally, those accesses and the paths inside the complex should be robot friendly in order to allow the movement of robot (i.e. ramps instead of staircases, enough space clearance for robots to pass through, etc.).

The fleet robots' main goals are:

- to automatically welcome and interact with guests
- to guide the guests
- to detect human intrusion via their optical sensor

Figure 6.12a presents the initial configuration of a fleet of robots in a research facility. The space is devised in 6 areas with a robot of the fleet in each zone: the access area (i.e. the pink

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés



(a) Intrusion detection (b) Lockdown configuration

Figure 6.13 – Intrusion detection

colored space) and other 5 zones with restricted access. When a guest arrives in the buffer zone area (see fig. 6.12b), the robot in this zone will verify its credentials (e.g. secured access card, fingerprint or retina scan), will compute a skeleton id of the guest in order for the other fleet members to recognize this guest and will compute the path to the guest desired location. In fig. 6.12c, the initial welcoming robot arrives at the end of its assigned controlled area, and will inform the guest that the robot in the green area will continue the guiding. The initial robot passes the information about the guest that includes the skeleton id. The same exchange is performed between the green and the yellow zone in fig. 6.12d and the guest arrives in the desired $R21$ area.

In fig 6.13a the robot in the pink area detects a violation of the access rights in the secured area. In this case the buffered zone is compromised. In order to prevent the intrusion in the secured facilities, an alert is broadcasted to all the fleet robots in the secured zones to create a lockdown. Their reaction is to block the access to those areas by locking the secured doors (i.e. interaction with the environment) or by creating a passage blockage with themselves (fig. 6.13b).

This complex fleet robotic project faces several challenges which can each be modelled as a multi-robot application:

- Human / object detection
- Localization and navigation of a known map
- Face recognition
- Decision making based on environmental changes
- Robot fleet connectivity

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

In the next subsection, we have designed, developed and analyzed using *ROSMDB* one of the application from this project: *Random movement for Human/ Object detection*.

6.2.1 Random movement object search: Description

One of the applications in the project *Guest welcoming and management with intrusion detection system* is represented by the detection of a guest or an intruder. Each fleet member needs to scan its designated control area by performing a random movement in order to avoid pattern recognition by possible unwanted guests. In the experimentation bellow we have simplified the imaging processing application by replacing a skeleton detection with a moving target: a green ball. We propose a solution that allows each fleet robot to move randomly inside a confined space and search for the target. Once found, the robot will approach the target. The solution was designed, developed, deployed and analyzed using *ROSMDB*. Multiple iterations were done in order to obtain the final results.

The benchmarks were performed on Turtlebot 2 robots equipped with an Intel Core 2 Duo, 2.1 GHz CPU, 4Gb of Ram, WiFi enabled (supporting Ad-Hoc networks) running on Ubuntu 13.04. The robots were looking for a green ball in an environment with no green objects.

6.2.2 Random movement object search: Model

The models of the final iteration are described below. A series of screenshots from *ROSMDB* corresponding to this scenario design, validation, development and feedback can be found in Appendix E. Our solution is composed of two independent applications: *Collision avoidance application* which main purpose is to avoid obstacles and other peer members and *Object detection application* which detects if the target is visible and moves towards it or performs a random movement.

The *Collision avoidance application* is composed of the following services:

Engine stopper service responsibility is to detect if an obstacle different from the target is close to the robot and stop the robot motor until a decision on how to avoid the obstacle is taken. The corresponding model can be found in fig. 6.14. In state $S1$, the model loops until an obstacle is found. In this case, it stops the motor of the robot in $S2$ and sends an alert to other services in $S3$. Then it waits in $S4$ for a decision on how the obstacle should be avoided by synchronizing with the other services on *ack ROS* topic.

Avoidance service objective is to compute and execute a path to avoid an obstacle when is detected. The model also verifies if the computed path was physically executed correctly. The corresponding model can be found in fig. 6.15. In the initial state $S1$, the robot waits for an alert to be triggered by *Engine stopper service*. In this case, in the state $S2$, the robot verifies the status of the motor. If the motor is on, the system switches directly to $S4$. If not, the model execution passes through $S3$ where it sends a *ROS* command to

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

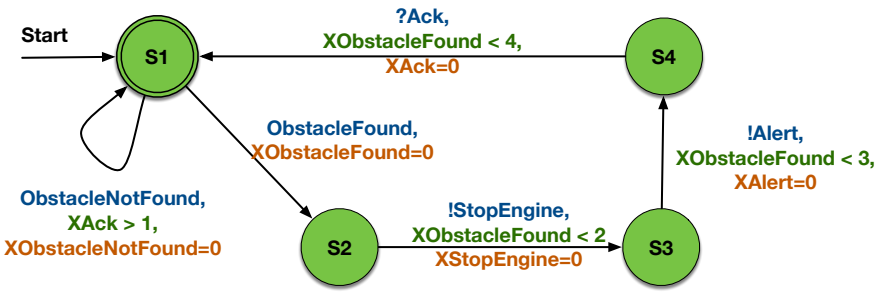


Figure 6.14 – Engine stopper Model executed on each robot

trigger the start of the motor. In *S*4, the robot computes the reverse path to be executed in order to avoid the obstacle and executes it. In *S*5, it waits for odometer data from *ROS* topic and verifies if the path was executed correctly in *S*6. If this is the case, it switches to *S*7 where it informs all the other services that the movement was executed correctly. If an alert arrives while the model is in one of the states related to the movement, the system resets the computed behavior and goes back to state *S*2.

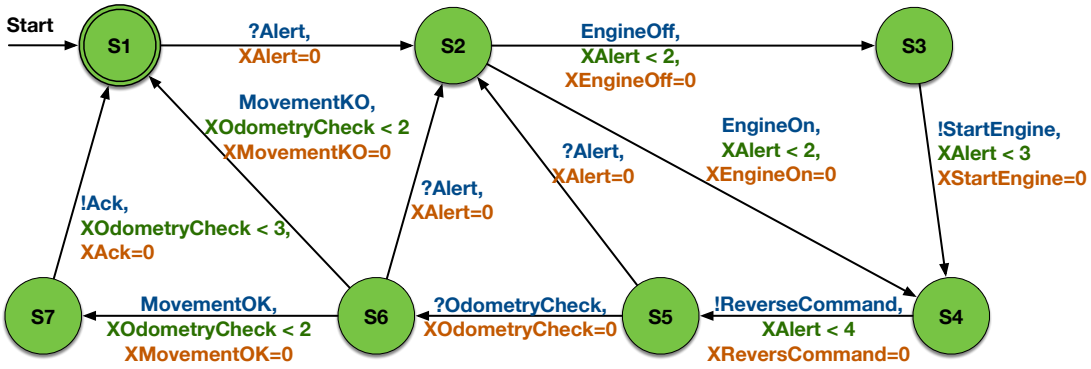


Figure 6.15 – Avoidance Model executed on each robot

The *Object detection application* is composed of the following services:

Image analyzer service is specialized in detecting the moving target (i.e. the green ball) and computing the path that needs to be executed in order to reach it. The corresponding model can be found in fig. 6.16. The model loops in *S*1 until the ball is detected. In *S*2 it computes the path needed to be executed in order to approach it and waits for the movement to be executed in *S*4. The execution is repeated multiple times until the target is reached because the movement can be cancelled by the *Collision avoidance application* or because the target has moved.

Movement service responsibility is to move the robot towards the target if it is detected or to perform a random movement in order to search for it. In *S*1, it waits for a computed

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

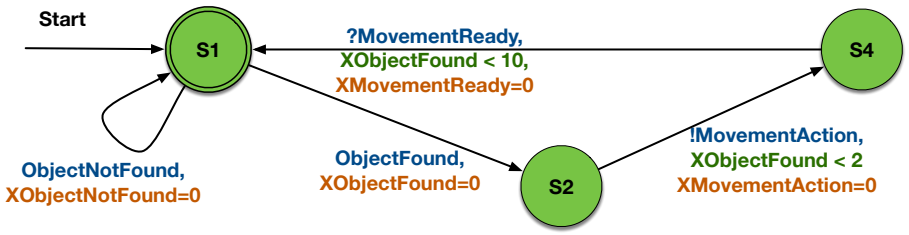


Figure 6.16 – Image analyser Model executed on each robot

path from *Image analyzer service*. If the path arrives in less than 5 seconds, the model applies the path in $S2$ and acknowledges the movement in $S3$. If no path arrives in the time interval, a random path is computed and executed in $S4$. This movement is also acknowledged in $S3$.

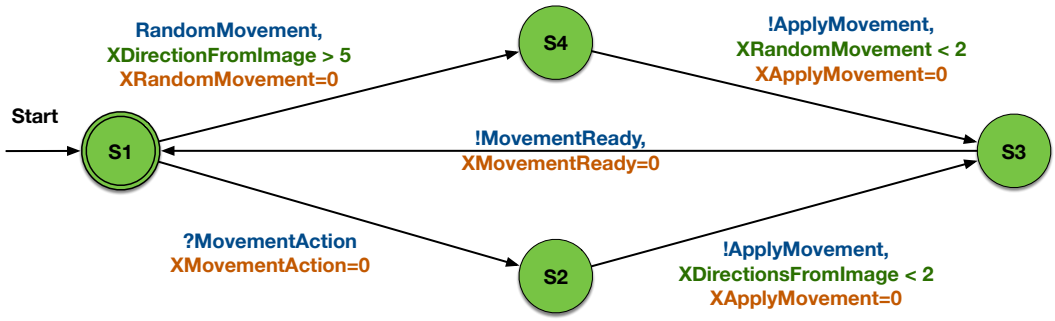


Figure 6.17 – Movement Model executed on each robot

Before filling the python skeleton, all the models and their composition were validated using *ROSMDB* model checker component. The reader may find in appendix E a series of screenshots from the tool-chain which includes also the validation of each model. We have checked for deadlocks in the models (i.e. liveness property) and if all the states are reachable from the initial state as well as if the final state is reachable (i.e. reachability properties). Below are listed some of the properties that were validated:

- E<> not deadlock *(liveness property)*
- E<> c0EngineStopper.WaitForKinnectImage *(reachability property)*
- E<> c0EngineStopper.Stop motor *(reachability property)*
- E<> c0EngineStopper.brodcastAlert *(reachability property)*

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés



Figure 6.18 – Experimenting Random movement object search with Turtlebots

6.2.3 Random movement object search: Experimental results

What is more interesting for the use of *ROSMDB* is that all the models were valid in the model checker. The initial approach was to set all the clocks guards to 1 second which failed in practice because the code execution in each state depended on processing images that was time consuming. The initial iteration helped us understand the real guards that we need to assigned. We have refined the model and have checked it again with the model checker. Figure 6.18 shows a in illustration of the experiment.

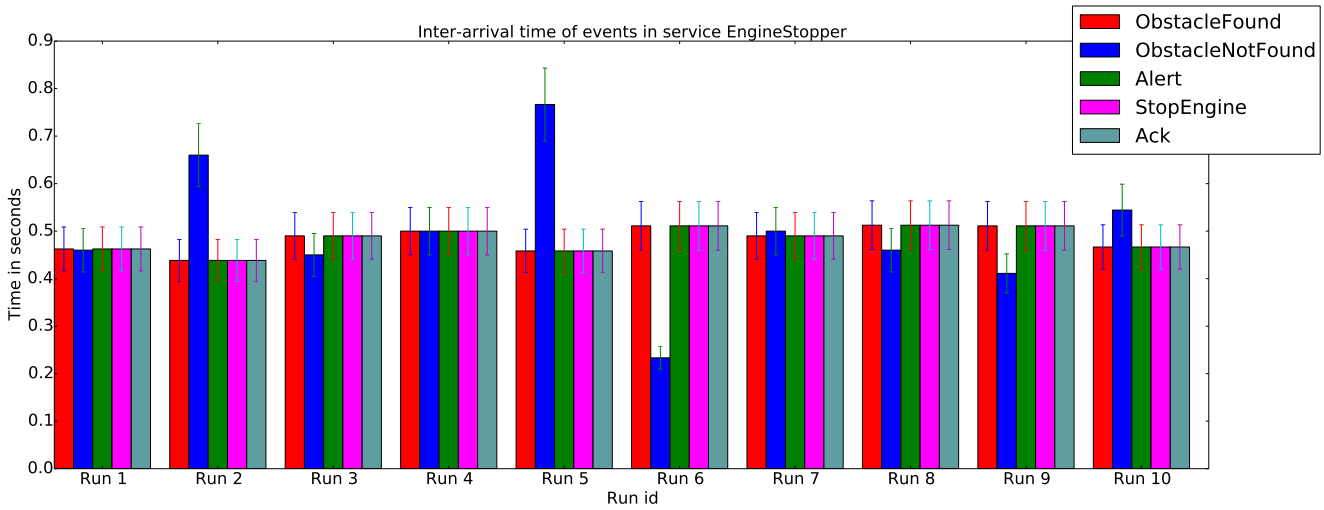


Figure 6.19 – Engine stopper messages inter-arrival time

Even if the time guards were correct and the model checker did not detect any deadlocks in the theoretical model, a deadlock was found at runtime as shown in E.6 of appendix E. This is

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

explained by the delay of message propagation in *ROS*. When the *Engine stopper* detected a possible collision with an obstacle, it stopped the motor of the robot and triggered an alert event. The *Avoidance service* received the alert, checked the status of the engine. The delay in *ROS* stopping the motor made the *Avoidance* service to see that it was still on and triggered computing an avoidance path, but in reality, the motor was shutting down. Once the path was computed and the motor was shut down, the movement message could not succeed in moving the robot, thus when the system checked if the execution was correct, it was not the case. The system blocked because the service *Avoidance* was waiting for an alert in *S*1 and the only model that could sent this alert was *Engine stopper* which was waiting for an *ack* event from the *Avoidance* service. This problem, that could not have been seen by the model checker due to its dependence on the physical system, was corrected in the latest iteration by simply waiting for the motor state to fully change state.

The results for the final refined code that avoided deadlocks collected in a benchmarking of 10 runs are shown below. In the case of *Engine stopper*, as shown in fig. 6.19, the *ack*, *alert* and *StopEngine* events are happening with the same frequency in the same run. The average of these events across all the benchmarking is 0.5125 seconds. The average inter-arrival time of *ObstacleFound* (i.e. 0.5125 seconds) and *ObstacleNotFound* (i.e. 0.46 seconds) is environment related and vary from run to run. We should remark that when an obstacle is found, the following events in the model have the same inter-arrival average.

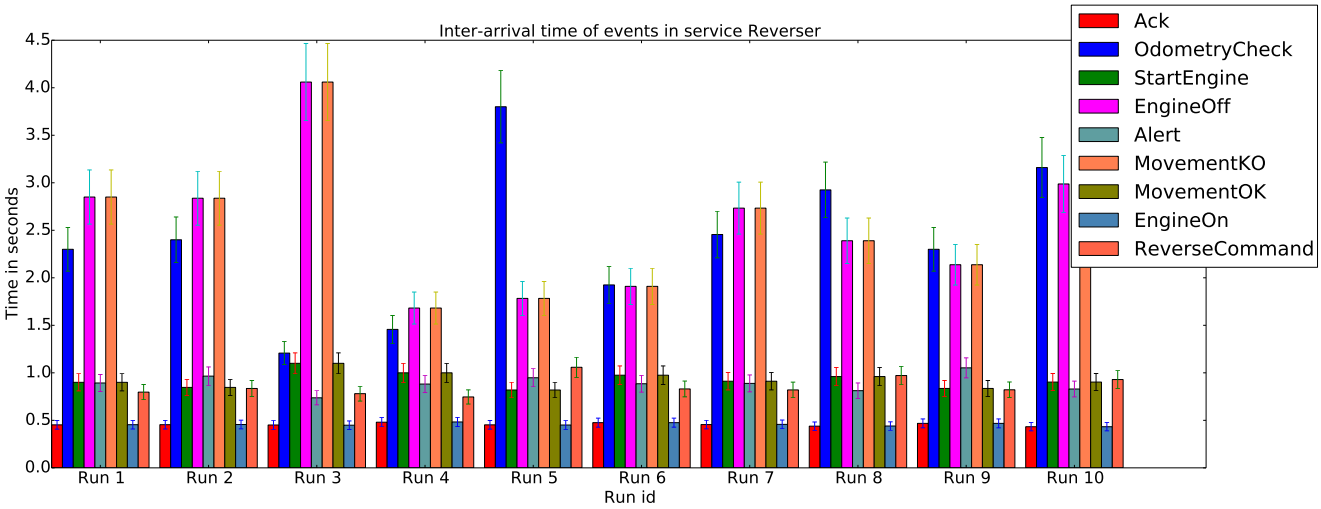


Figure 6.20 – Avoidance service messages inter-arrival time

As shown in fig. 6.20, the event with the higher inter-arrival time is *MovementKO* (average of 1.2076 seconds) followed by *EngineOff* (average of 1.1 seconds) and *OdometryCheck* (average of 0.9807 seconds). The *EngineOff* is triggered only after a waiting time in which the robot motor is allowed to completely change state from on to off, while the two other events are linked by the time it takes to compute the real executed path. The *ReverseCommand* event is correlated with the *EngineOff* because the same waiting time is propagated to the state where

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

the reverse command is computed. Having a high inter-arrival time for *MovementKO* means that the robot executed correctly the assigned path most of the time.

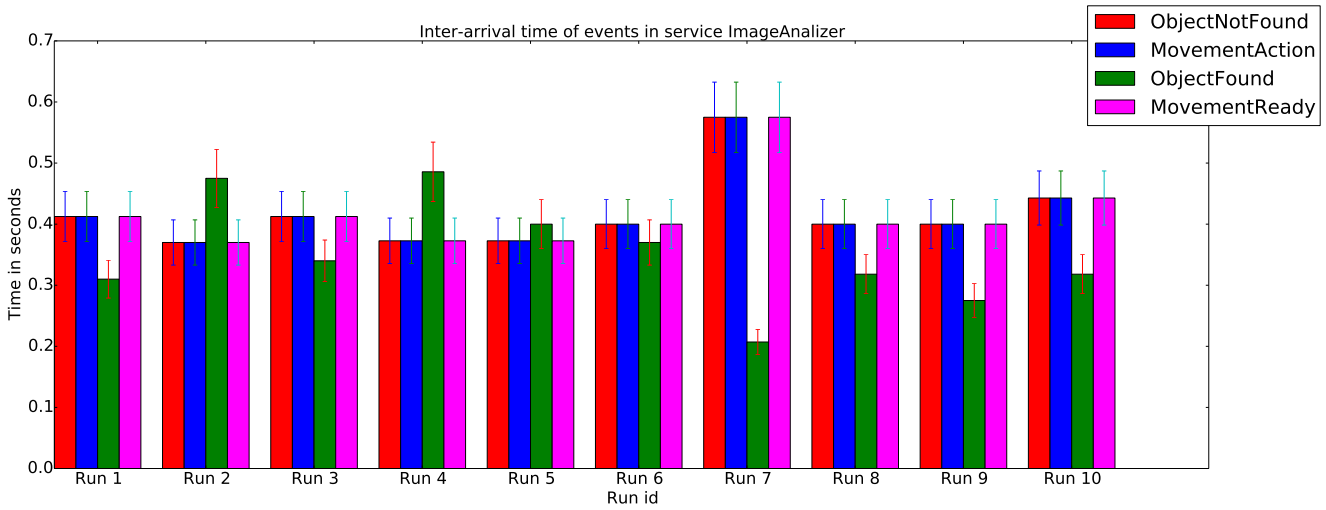


Figure 6.21 – Image analyzer messages inter-arrival time

Figure 6.21 presents the results for *Image analyzer service*. The events *ObjectNotFound*, *MovementAction* and *MovementReady* are correlated because when an object is not found, no other events of that time are triggered. Their average is 0.575 seconds.

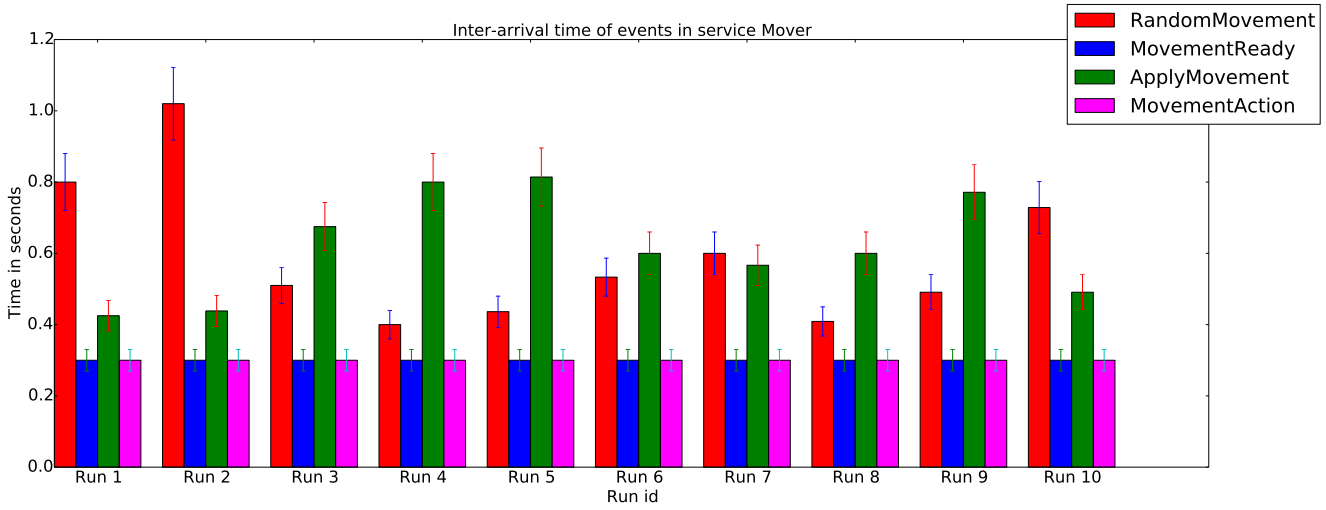


Figure 6.22 – Mover messages inter-arrival time

In fig. 6.22, a random movement is executed with an average of 0.54 seconds while the action of computed path that will approach the robot to the moving path has an average of inter-arrival time of 0.6375 seconds. The *MovementAction* and *MovementReady* are linked together because they are serialised events.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

This experimentation has allowed us to verify the use of *ROSMDB* tool-chain. First, it shows how mistakes that are not visible when applying model checking on theoretical models can happen in runtime. The tool was useful enough to detect this problem and we have corrected the behavior. Second, the feedback provided by the tool helped us to refine the source code in order to be compliant to the desired behavior of the robot.

6.3 Summary

We provided two scenarios and experimented with two sample applications in each scenario the use of *ROSMDB* tool-chain. It was proven to be helpful in the process of iterating multiple versions of the applications by allowing us to refine the model and detect violations that were not visible using classical off-line model checkers.

The main advantages of using *ROSMDB* that we have noticed in the previous experiments are the gain in time in performing multiple iterations of the development phase (the python skeleton was generated automatically, the deployment was done transparent from the development station, the feedback was displayed in a human readable way) as well as the possibility of noticing different behaviors between the theoretical model and runtime model. Combining development phase model validation with runtime alerts and transparent mapping of the theoretical model to the executed code allows for a faster and simpler iterations. On the other hand, the downside of *ROSMDB* is generated by the time needed by a developer to accommodate with the *MDD* approach in a web *GUI*.

ROSMDB uses timed automata as formalism to design multi-robot applications using a *MDD* over *SOA*. The framework behind uses *SDfR* and point to point network connections, being able to scale up with an extensible number of robots. The model checker behind *ROSMDB* is *UPPAAL* which accepts *TCTL* properties. The tool-chain can verify liveness and reachability properties and only analyses the events between the components (i.e. each component is seen as a black box).

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

7 Conclusion and perspectives

We conclude this thesis by summarizing the contributions. Then, we provide research and applications perspectives beyond this work.

7.1 Concluding remarks

First, we have reviewed and compared the existing middlewares for robotics that can be applied to a fleet context. It was noted that a robotic fleet can benefit from parallelization of tasks which reduces the time needed to accomplish them. The use of middlewares improves the information sharing and the robustness to failure. Based on this comparison, we believe that *Robot operating system (ROS)* [ROS, 2014] and *Microsoft Robotics Developer Studio (MRDS)* [MRDS, 2012] are the most suitable single robot middleware that can be applied out of the box to a fleet of robots. One assumption made in the beginning of this thesis, is to use *ROS* as low level robotic middleware because we think it is the emerging middleware with the most potential to become the most used framework for robotic fleets [Chitic et al., 2015]. The main reason for this assumption is based on the communication schemes provided by *ROS* (i.e. it supports both synchronous and asynchronous communications and it can easily be customized with new message types), on the large drivers ecosystem which allows a very good abstractions of hardware, on its plug-and-play modules system. It also allows the use of many programming languages including *Python* and *C++*. *ROS* was designed using a modular vision, thus, it allows us to compose multiple modules in order to create software services in a *Service Oriented Architecture (SOA)*

Secondly, it was reviewed a development approach to design and develop new software components called *Model driven development (MDD)*. Next, a series of classical formalism that can be used to design software behavior and their applications in robotics were reviewed. The focus was then assigned on a particular formalism called timed automata [Alur and Dill, 1994]. We have observed that models are used as a starting point into developing robotic software and architectures. *MDD* can also be applied in robotics, allowing an automated translation from models to software components. Robotic applications require often real-time

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

processes. Timed automaton was chosen as formalism applied to *MDD* because it allows time to be considered in the modelling phase of a robotic applications and it provides a powerful mathematical tool-set for model checking.

Then, we have combined the concepts reviewed in the state of the art in order to provide a service discovery protocol for robot fleet systems. After starting discussing the limited applicability of existing service discovery protocols in the context of robot fleets, we proposed afterwards a new protocol called *Service Discovery for Robots (SDfR)* that is suitable for service discovery inside an ad-hoc networked fleet. An extensive evaluation of different text and binary alternatives to implement *SDfR* was made using a fleet of Turtlebot robots, in order to measure and show that the overhead of *SDfR* is limited. *SDfR* is not only a contribution that combined a *MDD* approach using timed automata on a *ROS* based *SOA*, but was further used in tooling provided with the timed automata model based programming methodology we propose, *Robot operating system Model Driven Behavior (ROSMDB)*

Finally, the challenges to define a complete tool-chain to develop *MDD* multi-robot applications over a *SOA* using timed automata were presented. *ROSMDB* was proposed which includes a methodology based on timed automata to express robotic behavior, to specify temporal properties and to verify those properties against the model at both design and runtime phase. It represents a tool-chain that interacts with the entire lifecycle of a multi-robot application: design of the behavioral model, validation using timed automata formalism, code development, application deployment and runtime monitoring and feedback retrieval. In addition to the tool-chain, a *ROS* compliant framework has been proposed that allows the user to focus less on *ROS* and networking modules development. The *ROSMDB* framework allows for transparent states and transition manager and lets the user to only define the specific application source code logic for the multi-robot application. We provided a series of scenarios and experimented with samples applications in each scenario the use of *ROSMDB* tool-chain. These experiments, executed first on a fleet of Parrot Bebop Drones and secondly on a fleet of mobile Turtlebot allowed us to develop, validated, deploy real applications in order to evaluate our methodology and the proposed tool-chain. It was proved to be helpful in the process of iterating multiple versions of the applications by allowing us to refine the model and detect violations that were not visible using classical off-line model checkers.

We believe that modelling and analysis techniques with formal foundations such as the ones that we have presented will help at transforming the development and the maintenance of multi-robot fleet applications from a process that requires a substantial amount of manual interventions, to a model-driven process that is automated to a large extent.

7.2 Perspectives beyond *ROSMDB* and *SDfR*

The concepts and software presented in this thesis focused on providing a fully integrated timed automaton model based developing environment for robotic fleet application. While already being innovative by themselves, we believe that those contributions can play a signifi-

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

cant role when leveraged in the following contexts.

7.2.1 Short-term perspectives

Collaborative and Versioning extension for ROSMDB Even if *ROSMDB* tool-chain is represented as a web application that can be hosted on a centralized server and used by multiple developers, multiple users cannot work on the same model at the same time because this will end in overriding the others work, thus in version conflicts. A challenging extension could allow multiple users to work at the same time by providing a collaborative working environment.

These extensions should allow each user to trace the progress of his own work, thus to provide an historical view of the models and source code evolutions over time. Moreover, being able to work in a cooperative way, multiple user can share the same workspace and collaborate in order to design or refine the multi-robot application.

Both the Collaborative and the Versioning extensions provides interesting research and development problems:

- Versioning both the user code and the models needs to guarantee the mapping between those two in any given version. If the system tags a version with a refined model but with an un-updated source code, the mapping can be broken. This will end in properties being valid during the model checking phase but the application might have a different behavior at runtime. A departing point in building such system can be found in [Chacon and Straub, 2014] where the versioning systems reflects the existing model as a local copy on the user workspace. All the model modifications and updates will be saved only in this workspace. As mentioned in [Loeliger and McCullough, 2012], such system can facilitate the distributed development, scale to large number of developers, perform quickly and efficiently, provide atomic transactions and enforce accountability.
- Collaborative work on the same models and source code needs a consensus system that needs to take into account not only each user updates, but also the order of these updates [Weiss et al., 2009]. As mentioned in [Sun et al., 1998], one of the most challenging problems in real-time collaborative editing systems is consistency maintenance. This kind of system should provide not only a consistency model, with properties of convergence but also schemes and algorithms for generic operation transformation in order to support intention preservation. For the models, two users might refine the model together and the result is represented by the sum in time of both contributions. If a third person needs to work on the same model and gets only the updates for one of the two previous users, the consensus system should guarantee the final model is represented by the sum in time of all 3 contributors.

Other development perspectives for ROSMDB From development point of view, the follow-

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

ing features would be interesting to be included in *ROSMDB*:

a. Models and components repositories for existing services for *ROSMDB* : For example, if a robotic application needs a collision avoidance service, it could automatically obtain it from a repository including all the avoidance service models and source codes (if it was already published to the repository).

Existing software tool-chain like Apache Maven[1] or Gradle[2] provide such repositories. The user only specifies its dependencies in the manifest file of the service, and the tool-chain automatically downloads the corresponding packages. Such mechanism could be implemented in *ROSMDB* tool-chain allowing for a better reuse of existing components and services.

b. Security layer : The security issue was out of the scope of *ROSMDB* contribution. The main issue is how a system that relies on message passing communication scheme and which relies on source code deployment can be made secured?

ROSMDB framework offers an unsecured communication layer between fleet peers. This mechanism and the service description urls used by *SDfR* could provide service signatures and all the transported data should be encrypted [Tyagi et al., 2017].

Secondly, a security improvements can be also done in the tool-chain *GUI* (i.e. authentication and user management) by implementing a AAA[3] security protocol [Park et al., 2003]. Moreover, the package distributing system of the tool-chain could be render secured by implementing asymmetric encryption system using private-public security keys for data transmissions [Rodgers et al., 2017].

7.2.2 Long-term perspectives

Continuous integration system for robotic fleets In today's applications, providing automated testing at different layers (i.e unit-testing, integration and regressions tests) and continuous integration is critical. "Continuous integration is a software practice where developers integrate frequently, at least daily updates for their software" [Ståhl and Bosch, 2014]. This software concepts can be also applied in developing multi-robot applications.

This is becoming even more true in the case of *SOA* because no one has control on the services it uses. Even if a component model is valid today and the corresponding code acts in concordance with the model today, if an upgrade of the model or source code is performed tomorrow, it might generate changes in behaviors. If the model is

[1] Maven is a build automation tool used primarily for Java projects.

[2] Gradle is an open-source build automation system that builds upon the concepts of Apache Ant and Apache Maven and introduces a Groovy-based domain-specific language (DSL) instead of the XML form used by Apache Maven for declaring the project configuration

[3] AAA refers to Authentication, Authorization and Accounting. It is used to refer to a family of protocols which mediate network access.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

updated, this may cause the mapping corresponding code to become obsolete. In the other case, the update of the source code may introduce violations of the model. Indeed, performing continuous model validation based on the inputted rules can guarantee that a model is correct even after the update.

Such continuous integration system provides interesting research and development problems:

- Provide a test-case generator that, based on a model and properties can generate experimental test cases. The models validation test cases are represented by its properties. The output of injecting the model and its predefined properties into the model checker is represented as a boolean meaning if the properties are satisfied or not [Aichernig et al., 2017]. On the other hand, generating test-case for the executed behavior at runtime becomes more complex because these test cases should include a large number of possible failure points including communication problems, hardware problems, operating system errors, etc. [Panichella et al., 2017]. A test-case generator for such scenarios is needed in order to simulate at runtime various problems that may appear during the real execution of the mission.
- Provide a framework to analyze the trace-files based on the test-cases and the model. Currently, the analysis of the trace-files is executed manually. Such task needs to be automatized and included in an AI framework in order to decide if a generated test case has passed or failed [Lam et al., 2017].

Extend the usage of ROSMDB outside robotic environment ROSMDB is tool-chain that allows for the design, development, deployment and monitoring of multi-robot applications using models as starting point in the design of services inside a *SOA*. The same concept of modelling services could be used outside the robotic ecosystem that includes Internet of Things [Bermudez-Edo et al., 2017], smart home environments [Desolda et al., 2017] or vehicle to vehicle networks [Baldessari et al., 2007].

The main open question is if these environments are different from a robotic fleet or on which scale? It is true that the components in such environments are less mobile than robots (except vehicles), but all the components are network connected devices that offer services and capabilities in possible ad-hoc networks. Moreover, their behavior can be as well modelled using time automata *MDD* in a *SOA*.

If the robotic component is extended, *ROSMDB* could be used to model generic distributed services in a *SOA* as timed automata models. The tool-chain could be used to validate reachability and liveness properties of these models and their composition. Furthermore, the existing deployment system can be used to provision the newly developed application on networked nodes.

The main research issue in order to generalize the usage of *ROSMDB* outside the robotic world is generated by the specialty of *ROSMDB* framework. It is tightly linked to *ROS* and its data exchange schemes. Furthermore, generic distributed services may depend on a larger spectrum of communications types [Zeng et al., 2004]. The open question is how

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

could *ROSMDB* framework evolve in order to take into account all the communication and events handling in such nonspecific environments? Should the communication be delegated to the user code and rely on the user to hook the probes directly into the source code?

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

A Selected Middlewares descriptions

A.1 Player/Stage

The Player/Stage ([Kranz et al., 2006], [Collett et al., 2005]) project is designed to provide an infrastructure, drivers and a collection of dynamically loaded device-shared libraries for robotic applications. It is one of the first middleware that emerged for robotic systems and there are other middlewares that wrap Player. It doesn't consider a robot as a unity, but it instead treats each device separately, being a repository server for actuators and sensors.

The middleware is composed of 2 components: Player and Stage. Player/Stage is based on a three-layers architecture in which the top layer is represented by clients that are specialized software components. The middle layer is Player which provides common interfaces for different robot devices and services. The last layer is the robots, sensors, and actuators. Player refers to the device and server interface. The devices are made of a driver and an interface, and are independent of each other. They can subscribe to a Player server repository to become accessible to clients. Clients can connect to this repository to request data from the sensors, send commands to the actuators, or perform configuration changes to an existing device.

The connection between the clients and the devices are done in separate sockets, making the data transfer available for multiple concurrent clients. The communications between clients and devices are connection-less, leaving the control architecture for the client to deal with. The components of the device that allow the client to retrieve information and send control commands to the devices are the device interfaces. These interfaces communicate with the device drivers that process the information. The socket communication implies that the clients software can be written in any programming language that has socket support. *C*, *C++*, *Java*, Common *Lisp*, *TCL* and *Python* are supported client programming languages. Other programming languages can access the interface provided by Player using various client-side libraries.

Stage is a graphical 2D simulator that models devices in a user defined environment. It also has socket based communication that uses the same interface on the real robot as in the

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

simulator. The platform that can run the Player/Stage middleware include: MobileRobots, Segway, Acroname, K-Team robots, iRobot's RFLEX-based, Botrics and Evolution Robotics.

Players main futures are the device repository server, the variety of the programming languages, the socket based transport protocol, modularity and the implementation being open-source.

A.2 Robot operating system

Robot operating system (ROS) is a recent flexible middleware for robot applications [Cousins et al., 2010, ROS, 2014]. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. It provides hardware abstraction, device drivers, visualizers, message-passing, package management.

At a low-level, *ROS* is a *XML-RPC* communication framework for sending information across processes. The processes in *ROS* are call nodes. Since communications are being wrapped into *HTTP* requests, which represent a language-agnostic *Transmission Control Protocol over Internet Protocol (TCP/IP)* protocol, the applications that uses *ROS* can be written in a variety of languages and can be distributed across multiple *TCP/IP* enabled devices. The communication between nodes can be done in two ways. For asynchronous communications, a publish-subscriber mechanism is provided where multiple nodes can publish / receive broadcast information on a name channel (known as topic). Alternatively, synchronous communications can be done using *ROS* services, a *RPC* system that allows a node to call a service from another node.

ROS is composed of two key components: the *ROS* master and *ROS* nodes. The messages that are sent across the topics are written in a specific *IDL*. A message is a simple semi-structured data type. Standard primitive types (integer, floating point, boolean, etc.) and arrays of these types are supported. Messages can include arbitrarily nested structures and arrays (much like *C* structures). The *ROS* Core is composed of the master node (a name server that allows node to subscribe and keeps tracks of each created node and topic) and *ROS* parameter server (a shared, multi-variate dictionary that is accessible via network *APIs*). The *ROS* nodes are executables that use *ROS* to communicate with other nodes and represent the application layer of the architecture. *ROS* comes with a series of libraries containing often-need robotic services like *SLAM*, Autonomous navigation of a known map, object follower, etc. *ROS* is designed to be cross-platform.

The official supported programming languages for *ROS* are *C++* and *Python*, but there are compatibility libraries for *Java*, *C#*, *Lisp* and *Go*. Up to now, it can run native on Ubuntu Linux, but it is compatible with other Unix operating systems, MacOS and Windows. The platforms that support *ROS* include PR2, Turtlebot, Kobuki, Husky and Dr. Robot Jaguar V4 with Manipulator Arm, etc.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

A.3 Miro

Miro is a distributed, *object*-oriented middleware developed to improve the software development process by increasing the integrability of heterogeneous software, the modularity and the portability of robot applications [Kraetzschmar et al., 2002, Krüger et al., 2006]. It was developed in *C++* for Linux based on the *CORBA*. This allows cross-platform interoperability making the middleware applicable to a distribute multi-robot context. Due to the restrictive nature of *CORBA*, software application can be only written in languages that provide *CORBA* implementations.

The Miro architecture is organized in three layers: the device layer, the service layer, and the class framework layer. The device layer provides object-oriented interface abstractions for all hardware devices (sensor and actuator) and makes it platform-dependent. The service layer provides abstractions for the device layer via the *CORBA IDL*. The class framework provides a number of services usually needed by application such as mapping, self-localization, visualization facilities, and so on.

All the Miro's components data exchanges are event-triggered. The platform supporting Miro include iRobot B21 and MobilieRobots Pioneer. Miro is very flexible and can be easily extended to support new devices and robot applications.

A.4 MRDS

Microsoft Robotics Developer Studio (MRDS) is a Windows-based middleware for robot control and simulation from Microsoft [Johns and Taylor, 2008, MRDS, 2012]. It is composed of four major components: *CCR*, *DSSs*, *VPL* and *VSE*. The *CCR* is a .NET-based concurrent library implementation for managing asynchronous parallel tasks. *DSS*, which allows the orchestration of multiple services to achieve complex behaviors is lightweight services-oriented runtime using message-passing technique.

VPL is a graphical development environment that uses a service and activity catalog. A service or an activity is represented by a block that has inputs and outputs that just need to be dragged from the catalog to the diagram. These components can interact graphically. *VPL* also allows the generation of code of new "macro" services from diagrams created by users. Finally, *VSE* is a simulation environment.

MRDS is aimed at academic, hobbyist, and commercial developers. It handles a wide variety of robot hardware like Eddie Robot, ABB Group Robotic, CoroWare CoroBot, Lego Mindstorms NXT, iRobot Create, Parallax Boe-Bot and more.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

A.5 MARIE

Mobile and Autonomous Robotics Integration Environment (MARIE) is a middleware designed to allow the integration and distribution of software for robotic systems [Côté et al., 2007, Côté et al., 2006]. Its main objectives are to allow developers share, reuse and integrate software in order to accelerate the development of robotic applications. It was created in *C++* and uses the *ACE* communication framework. The centralized component provided by the middleware called *MDP* allows software components to connect to *MARIE*.

MARIE is organized in three layers: Core, Component and Application layer. The core layer consists of services for low-level communication, data handling, *Input/Output (I/O)* control, and distributed computing functions. The Component layer is used to add components for integrated services and support domain specific concepts. The Application layer contains useful services to build and manage integrated applications. MARIE can run on MobileRobots Pioneer 2. Its main features are the interoperability and re-usability of robotic software modules.

A.6 Orca

Orca is an open-source middleware for developing component-based systems [Makarenko et al., 2006, Makarenko et al., 2007]. It provides the mechanics to create building-blocks which can be pieced together to form arbitrarily complex robotic systems. Orca can be used in various applications, from single vehicles to distributed sensor networks. It was designed and developed to maximize the software reuse and modularity in robotic applications. Orca is highly dynamic, with a distributed component base system that allows the user to define custom interfaces and communication protocols.

To implement a distributed component-based system, *CORBA* was chosen in the first version of Orca, but it was rapidly changed with ICE [Michi Henning, 2010], a new approach to object-oriented middleware that offers a much smaller and more consistent *API*, lighter implementations, advanced services, and good performance. It supports essential *C*/*C++* programming languages on Linux. Since ICE supports *C++*, *Java*, *Python* and *C#* and since ICE clients and servers can work together regardless of the programming language in which they are implemented, the supported programming languages can be extended to these languages. The platform that can run the Orca middleware include: MobileRobots, Segway, K-Team robots, iRobot's RFLEX-based, Evolution Robotics.

A.7 Carmen

Carnegie Mellon Robot Navigation Toolkit (Carmen) is an open-source collection of middlewares that focuses on the robot control by providing various control interfaces [Montemerlo et al., 2003, CARMEN, 2008]. It is written in the *C* programming language and it is organized

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

in three layers: hardware interface, common services and application layer. The hardware interface provides low-level communication, control by creating a hardware abstraction for sensors and other components. The second layer offers off-needed robotic services like navigation, localization, object tracking, and motion planning. The last layer is represented by the user-defined applications that share information and relies on data revived from the lower layers.

A key feature of the middleware is the modularity. The communication between different modules is done using *IPC*. The platforms that supports *Carmen* include MobileRobots, Nomadic Technologies Scout, iRobot ATRV, etc. The middleware is accompanied by a simulator with graphical display and editors.

A.8 Pyro

The goal of *Python Robotics (Pyro)* is *"to provide a programming environment for easily exploring advanced topics in artificial intelligence and robotics without having to worry about the low-level details of the underlying hardware a robot programming environment"* [Blank et al., 2006, Blank et al., 2005, Pyro, 2012]. It has educational purposes, and it wraps the Player/Stage middleware so that any component written for this system is also available to *Pyro*.

There are many libraries for *Pyro* that provide specific robotic services. The only programming language supported is *Python*. The middleware is compatible with MobileRobots Pioneer, Sony Aibo and all robots supported by Player/Stage.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

B Model driven development in robotics

B.1 RobotML

RobotML [Dhouib et al., 2012] is a *MDD* approach based on *DSL* that allows the design, simulation and development of robotic software components. The model behind RobotML is composed of four sub-models. The first sub-model, the architectural, specifies the structure of the application. It also contains meta information about the data types used, the environment, the platform on which the application is being deployed and the context of the robotic mission. The communication is the second sub-model and is in charge of defining the means of communication like ports mapping and protocols used. The third sub-model is associated with the behavior of the robot. Using state machines, it specifies what is the behavior of the robot during its missions. The last sub-model refers to the rules to compose the robotic software during the deployment phase to a target robot or simulator. The framework is part of the PROTEUS [Dhouib et al., 2012] French research project.

B.2 *V3CMM*

V3CMM [Alonso et al., 2010] is a modelling language. It is composed of three views:

- Structural view - a static representation of the structure used to create the components from the model. It also includes the ports and interfaces used to communicate as well as a description of how these interfaces are bound together.

- Coordination view - describes the interconnections of components in an event-triggered context. The models are created by state machines described using *Unified Modeling Language (UML)*.

- Algorithmic view - describes the specific business logic of the application. It is created using *UML*.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

B.3 SmartSoft

In SmartSoft [Schlegel et al., 2009b], the model is used in the generation of component "hull" (skeleton) which is in charged with the external publication of its service and the internal interactions between parts of the component. This skeleton is composed of four layers:

- User code layer - This layer is filed with the specific application business logic.
- Communication layer - The communication is created over a top of communication patterns allowing external services to share information via internal interfaces to user code inside component.
- Platform independence layer - It includes all the concepts that are independent from the platform in use, such as threads management, processes management, etc
- Platform specific layer - It refers to the specific middleware, operating system and hardware in use.

B.4 BRICS model

BRICS [Bruyninckx et al., 2013] is based on model driven approach in which the model, named *Component Port Connector (CPC)* meta model, is oriented on:

- Configuration - The components are dynamically configured based on their properties defined in the meta-model.
- Composition - Components can be composed hierarchically in order to form a composite component.
- Coordination - A composite component includes a coordinator part who is scheduling the computational process.
- Computation - It contains the meta information needed for the algorithms resulted from the composition to perform the robotic mission.
- Communication - The information at this layer represent the means of communications like how connectors are linked to allow 2 components to execute event or service calls.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

C Product construction of examples

C.1 Obstacle detection and avoidance navigation service

One key module of a mobile robot fleet application is real time obstacle detection and avoidance. In nowadays context, most of the mobile robots are featured with some type of collision avoidance, starting from less complex algorithms which will stop the robot immediately when an obstacle is detected, towards more complex algorithms that will recompute the path in order for the robots to detour the obstacles. Those latter algorithms involve not only the means of detecting the obstacle, its size and dimensions, but also they include a more resourceful computational unit, since they need to drive the robot around the obstacle and resume the path to the initial target. These algorithms are being part of the autonomous navigation concept. In general, in autonomous navigation, the environment may have known and unknown obstacles. All these assumptions are taken into account in the global path planning algorithm that plots the robot initial path in order to avoid known obstacles as well as in local path planning involved in unknown obstacles avoidance.

C.1.1 Optical sensor component

Its dedicated task is to get a depth and RGBA image from the optical sensor in order to analyze if an object is present on the robot trajectory. The component *ERA* is presented in C.1.

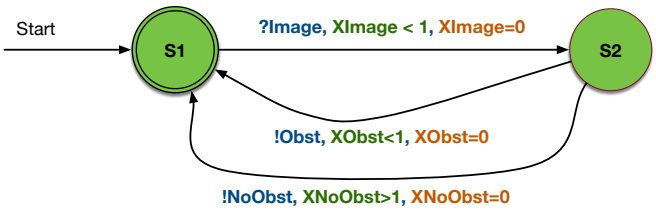


Figure C.1 – Optical sensor component Event Recording Automaton

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

C.1.2 Navigation component

This part is performing the actual movement of the robot and it avoids objects. The *ERA* is formed from several states, show in fig. C.2.

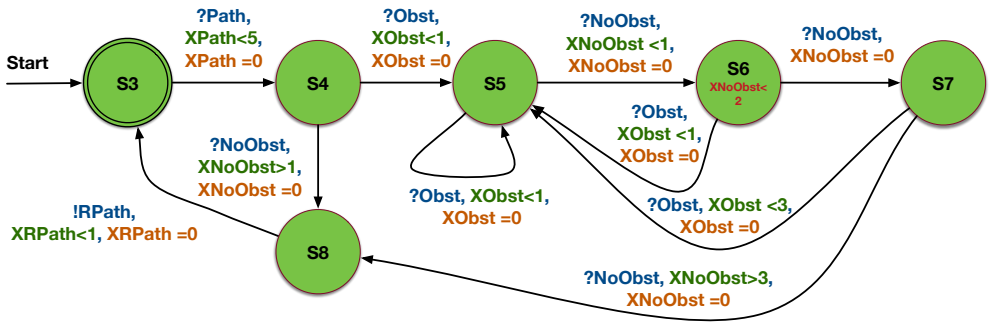


Figure C.2 – Navigation component Event Recording Automaton

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

C.1.3 Product construction of the service

$A_{Opticalsensor} \parallel A_{Navigation}$ represents the product of timed automata $A_{Opticalsensor}$ and $A_{Navigation}$ it is presented in fig C.3.

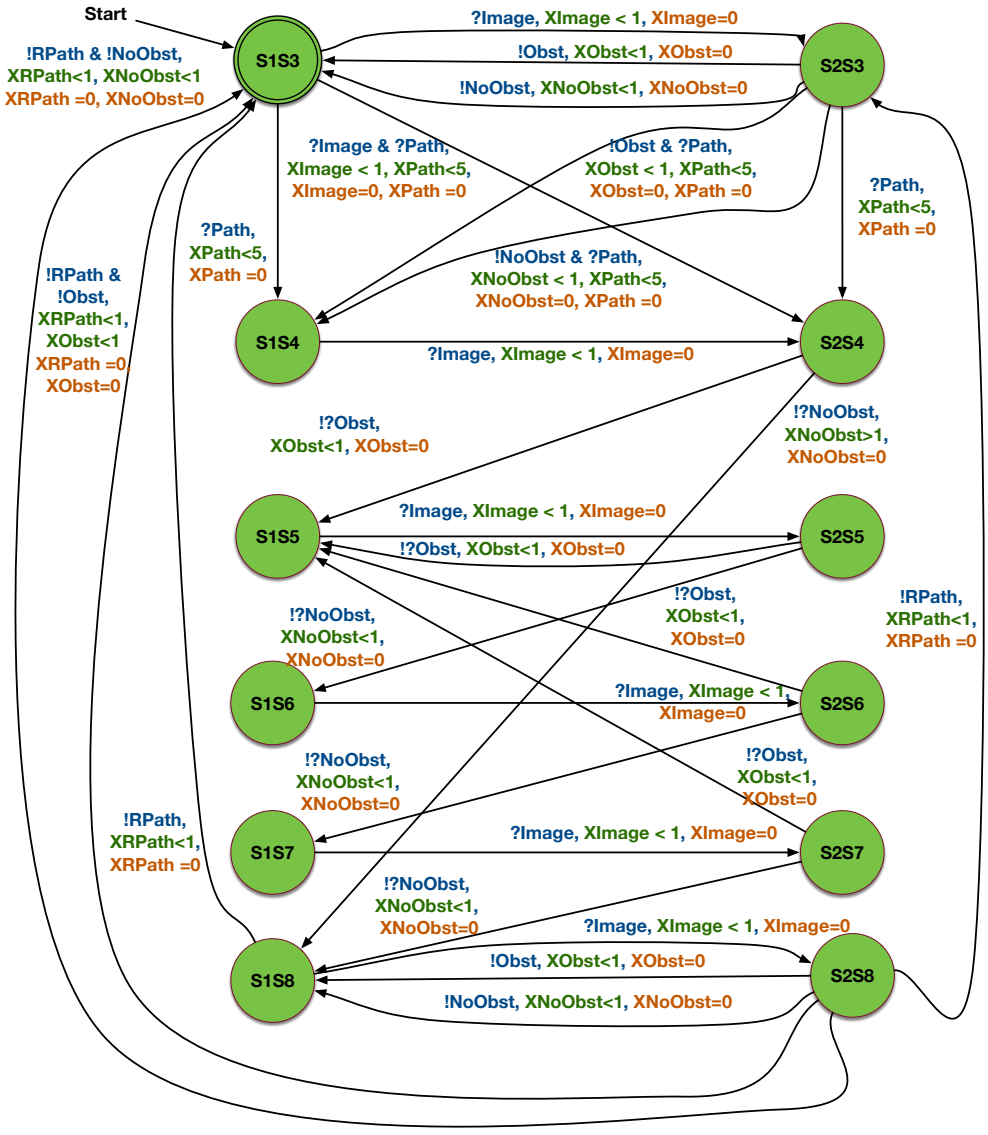


Figure C.3 – Product construction of components for service: Obstacle detection and avoidance navigation

The full representation of the product can be simplified just to two states showing how the service reacts with the external environment as shown in fig. C.4.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

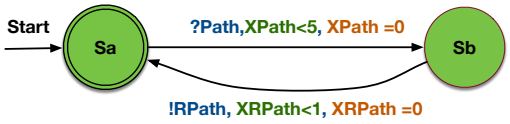


Figure C.4 – Reduced product construction of components for service: Obstacle detection and avoidance navigation

C.2 Fleet platooning service

A fleet platoon is a group of robots that react in a coordinated way. Typically, in a fleet, there is one robot that leads the platoon while all other robots are following it with the same speed and within certain boundaries for inter-robot distance. The leader can decide to accelerate, to brake or change direction and the following robots will mimic its actions. Coupled with autonomous navigation of an unknown map, the leader can avoid an obstacle leading to the entire fleet avoiding the same obstacle. Such systems that are found on the cooperation between peers (in the platooning case, cooperation between the leader and the other fleet members or between two adjacent robots) relay on wireless communication. In this case, the network should have standardized, efficient protocols with a minimum loss of packages.

C.2.1 Leader component

The main task of this model is to decide if a robot is a leader (first robot in the platooning row) or not based on a configuration file. The example can be future detailed with a leader election process, but it is out of the scope of this example. The model is visible in fig. C.5.

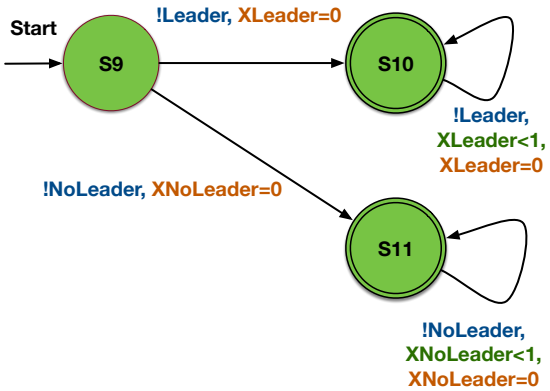


Figure C.5 – Leader component Event Recording Automaton

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

C.2.2 Networking component

This part is managing the *IP* messages that are exchanged between the robots in the fleet. The component *ERA* is presented in C.6.

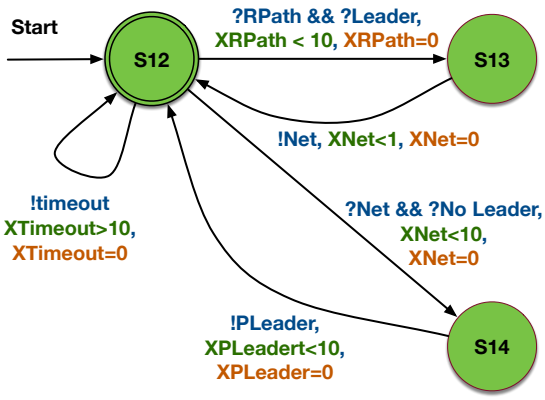


Figure C.6 – Networking component Event Recording Automaton

C.2.3 Platooning manager component

This component represents the main logic of the service. The model is visible in fig. C.7.

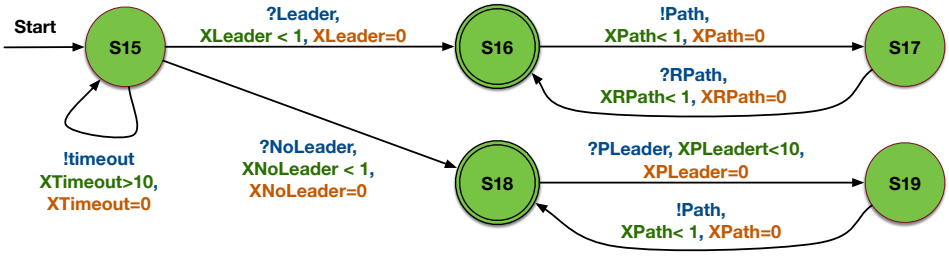


Figure C.7 – Platooning manager component Event Recording Automaton

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

C.2.4 Product construction of the service

Using the associativity property of the timed automata product, $A_{Leader} \parallel A_{Networking} \parallel A_{Platooningmanager} = (A_{Leader} \parallel A_{Networking}) \parallel A_{Platooningmanager}$.

$A_{Leader} \parallel A_{Networking}$ represents the product of timed automata A_{Leader} and $A_{Networking}$, used afterwards as a partial product, and it is presented in fig C.8. The non-connected states are unreachable combinations of states.

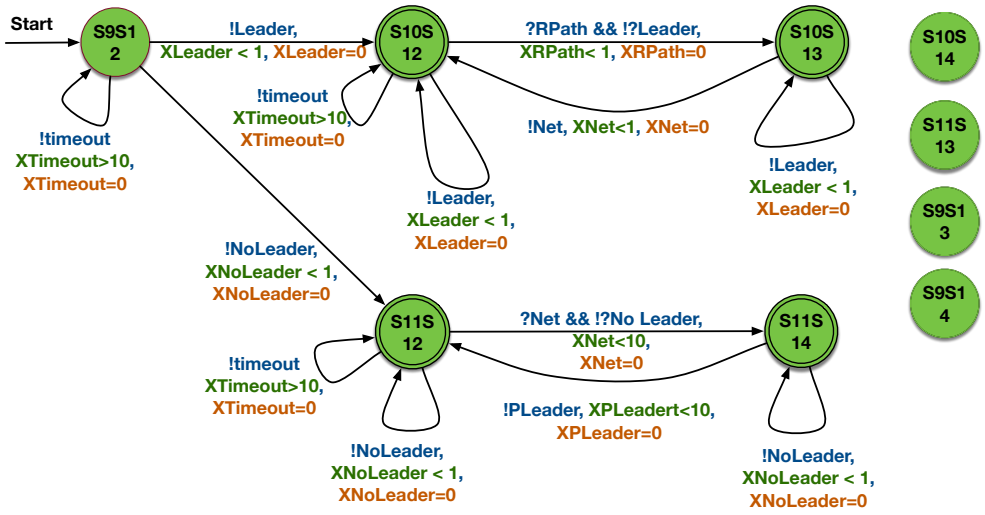


Figure C.8 – Partial Product construction of components for service: Fleet platooning

$A_{partial} \parallel A_{Platooningmanager}$ represents the full product of service timed automata and it is presented in fig C.9. The non-connected states are unreachable combinations of states.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

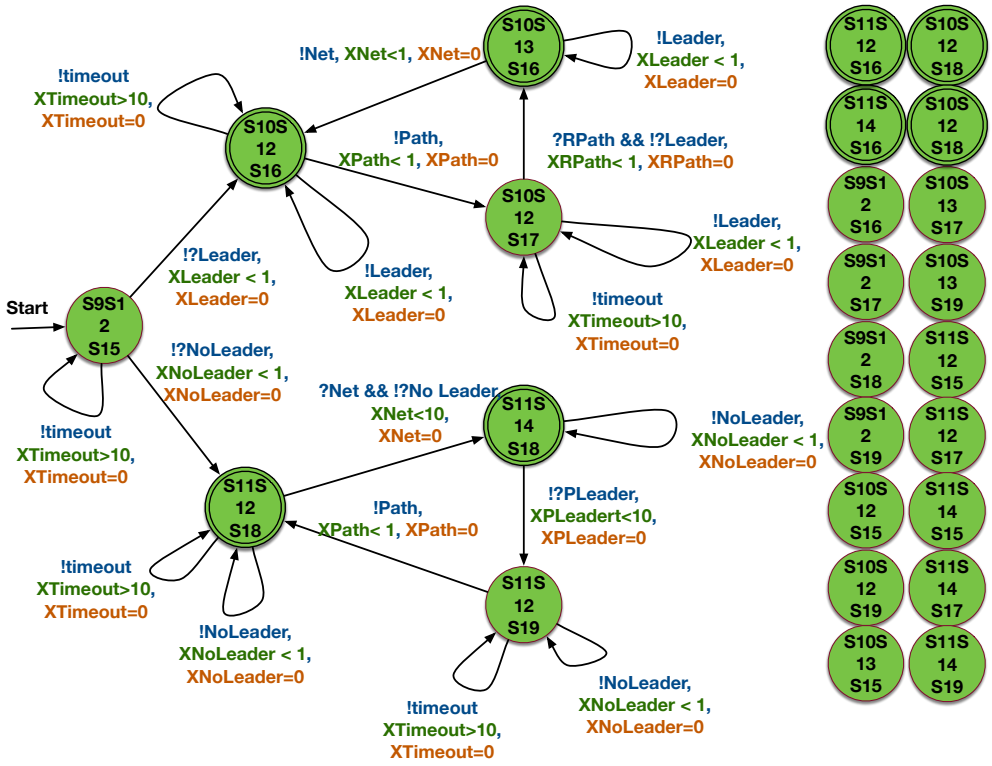


Figure C.9 – Product construction of components for service: Fleet platooning

The full representation of the product can be simplified just to four states showing how the service reacts with the external environment as shown in fig. C.10.

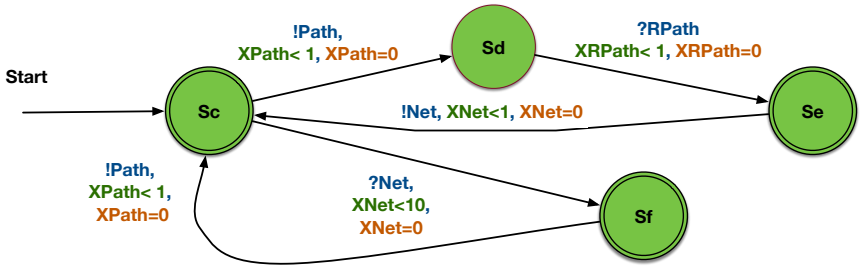


Figure C.10 – Reduced product construction of components for service: Fleet platooning

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

C.3 Fleet platooning robot with collision avoidance application

The examples presented above can be represented as services that can be combined inside of a same robotic application that is running inside a fleet. The robotic application, in this case, consists of a fleet platooning capable of avoiding collisions.

$A_{Obstacledetectionandavoidancenavigationservice} \parallel A_{Fleetplatooningservice}$ represents the full product of application timed automata and it is presented in fig C.11.

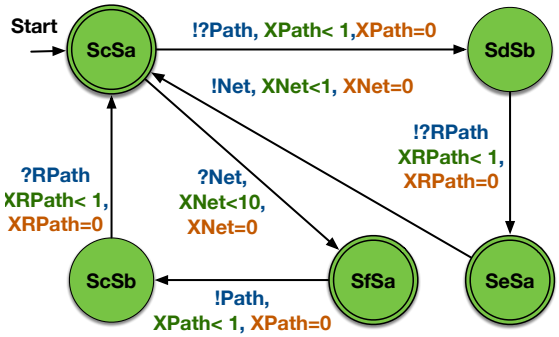


Figure C.11 – Product construction for entire robotic application

The full representation of the product can be simplified just to one state showing how the application reacts with the external environment as shown in fig. C.12. In this case, the application will exchange information with other instances of the same application running on different robots via *IP* messages.

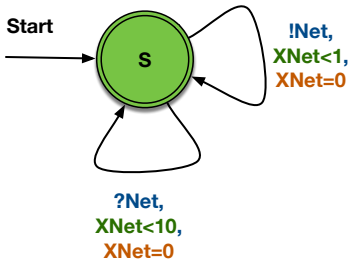


Figure C.12 – Reduced Product construction for entire robotic application

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

D Flight synchronization based on N pole

This appendix displays the screenshots of the project modelled, verified in *ROSMDB* tool-chain as well as the feedback given by the tool.

D.1 Commander application

D.1.1 Take off manager service

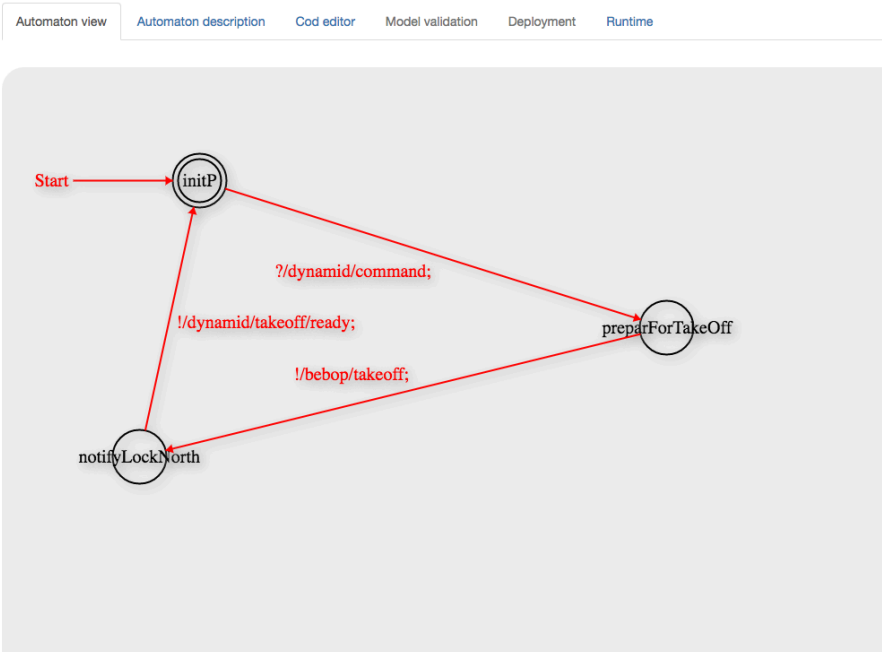


Figure D.1 – Take off manager: *ROSMDB* model screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

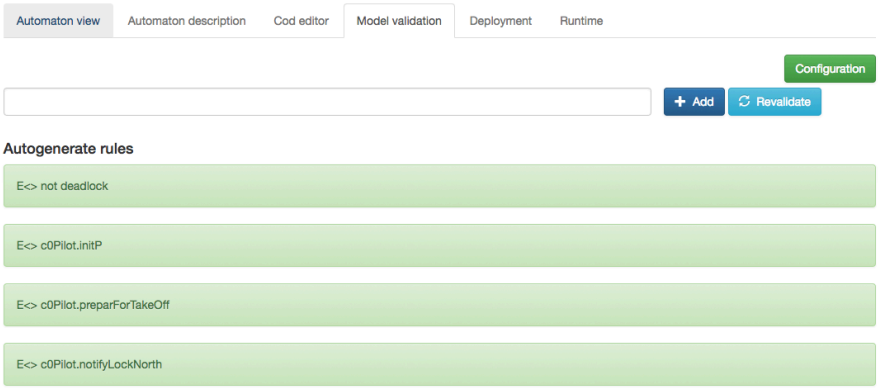


Figure D.2 – Take off manager: *ROSMDB* model-checking screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

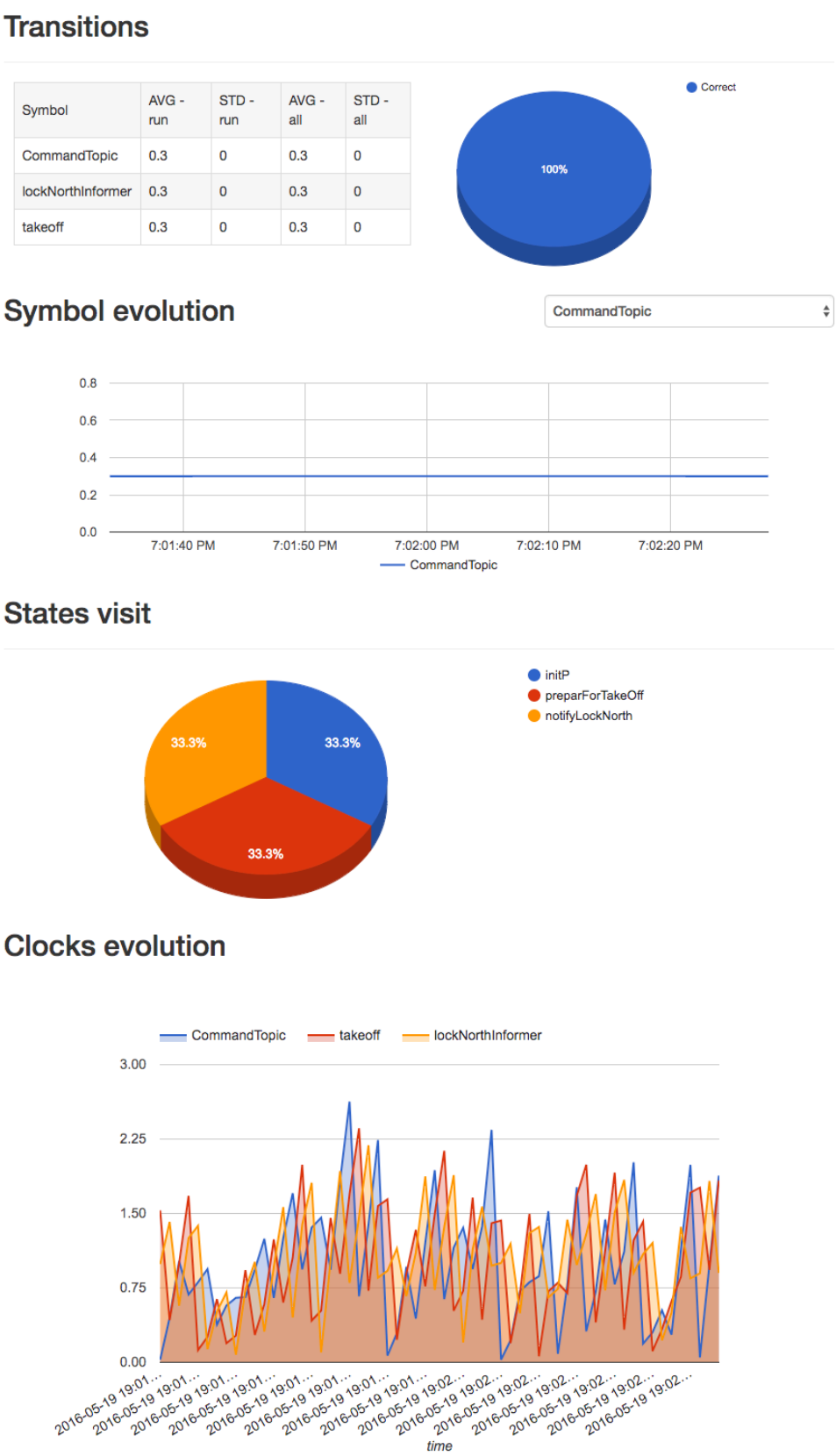


Figure D.3 – Take off manager: feedback results example screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

D.1.2 Lock North manager service

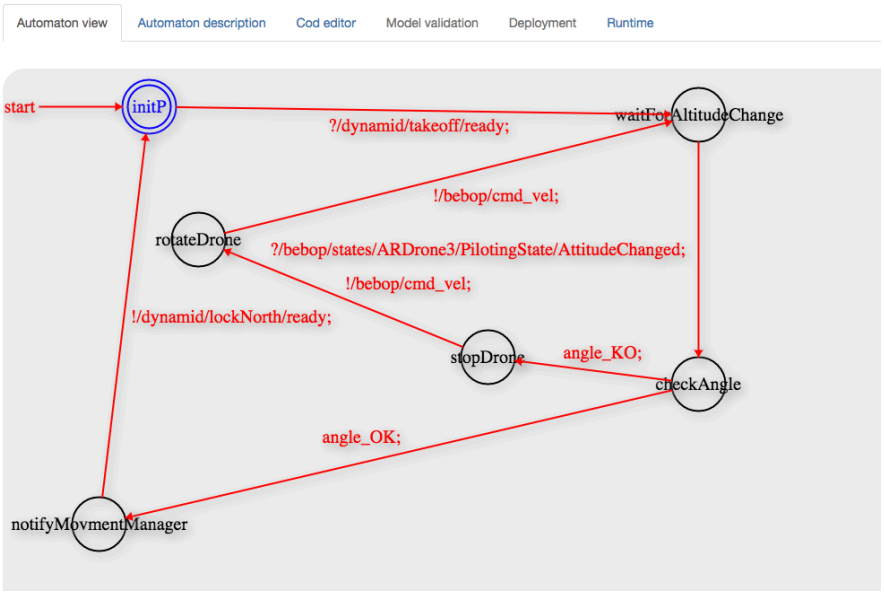


Figure D.4 – Lock North manager: *ROSMDB* model screenshot

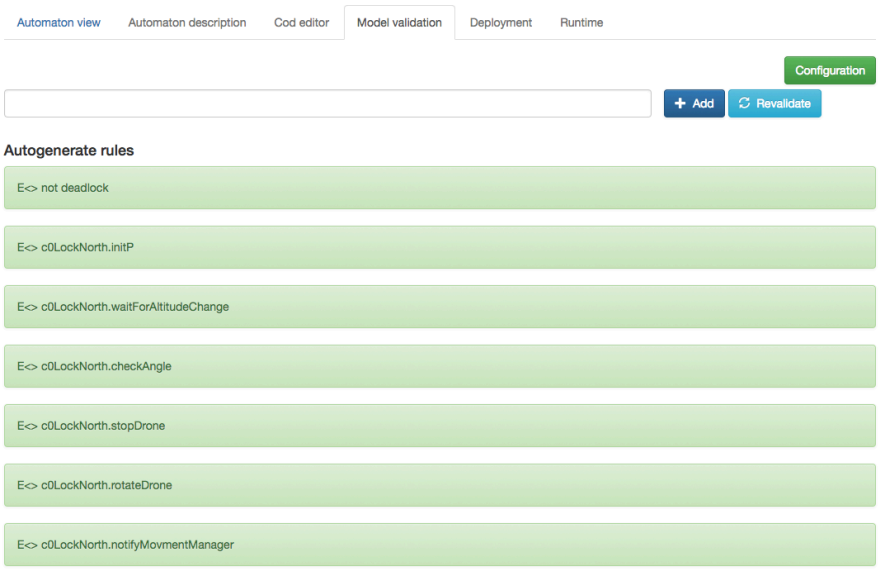


Figure D.5 – Lock North manager: *ROSMDB* model-checking screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

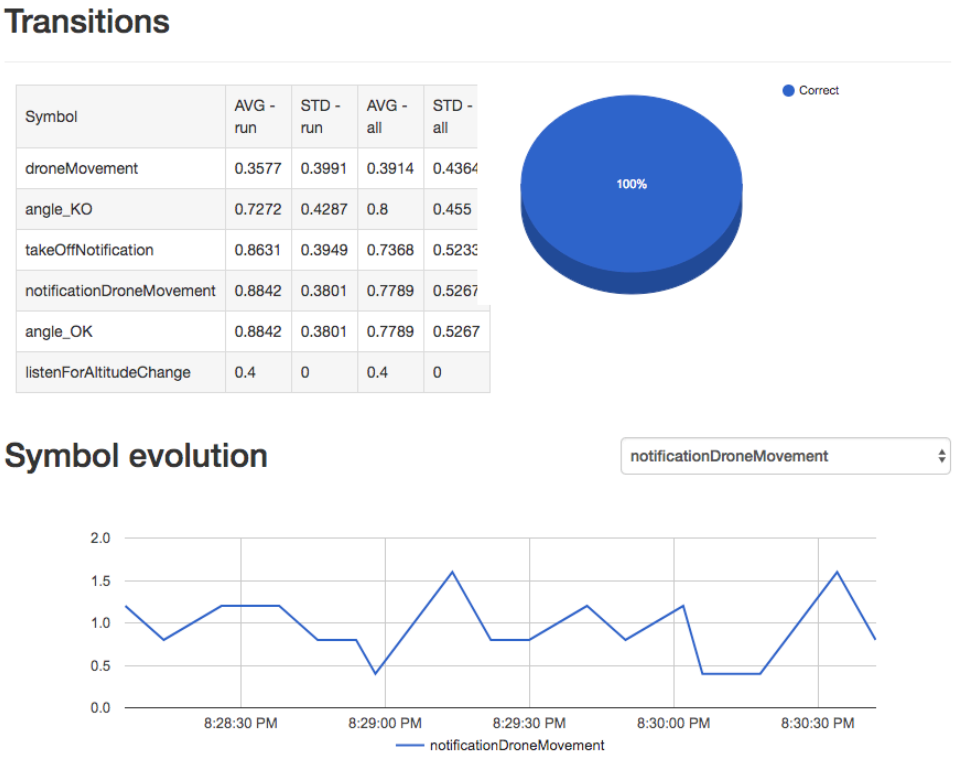


Figure D.6 – Lock North manager: feedback results example screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

D.1.3 Drone Movement Manager service

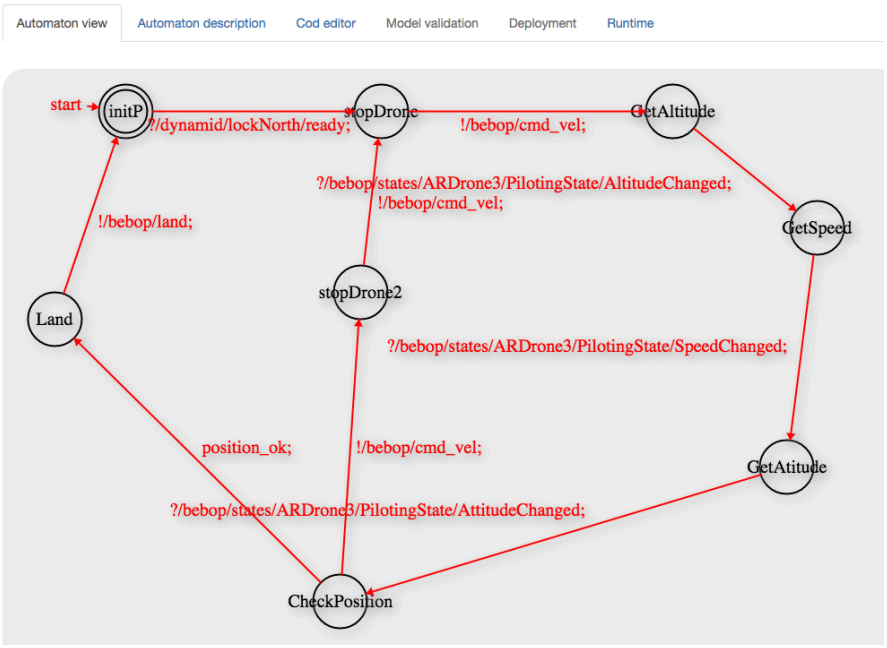


Figure D.7 – Drone Movement Manager: *ROSMDB* model screenshot

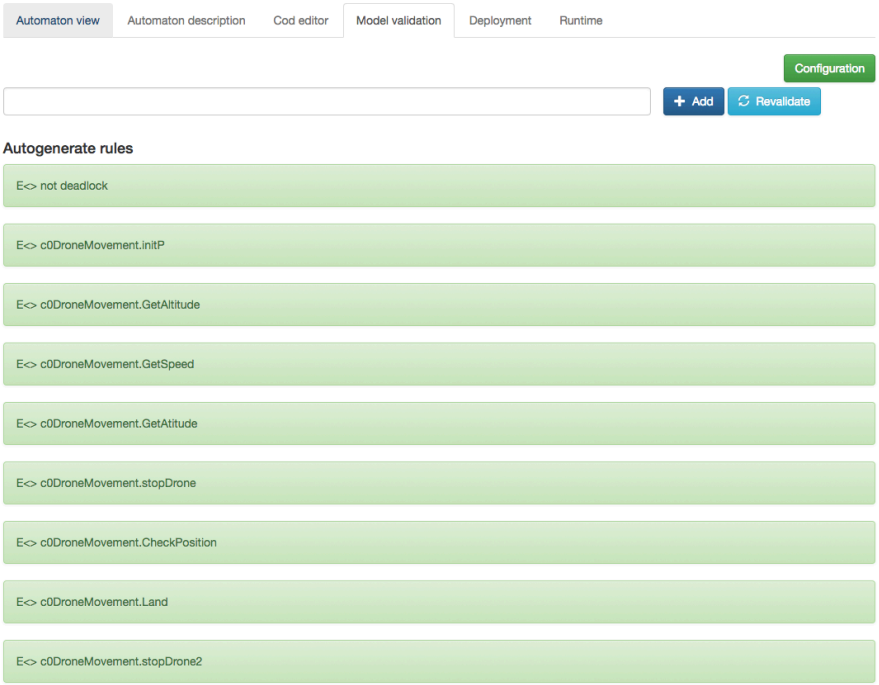
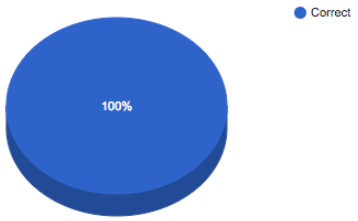


Figure D.8 – Drone Movement Manager: *ROSMDB* model-checking screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

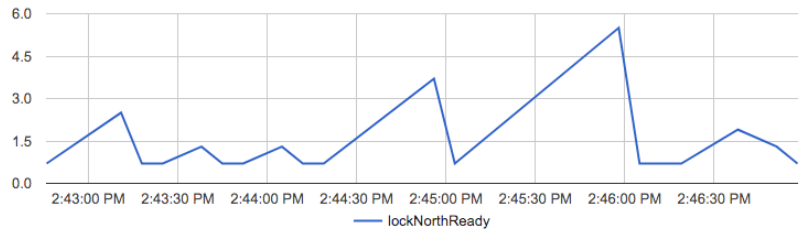
Transitions

Symbol	AVG - run	STD - run	AVG - all	STD - all
moveDrone	0.3158	0.2446	0.3158	0.2446
landDrone	1.3631	1.2448	1.3631	0.8234
listenForAltitudeChange	0.6475	0.0499	0.6475	0.0499
position_ok	1.3631	1.2448	1.3631	0.8234
listenForSpeedChange	0.6475	0.0499	0.6475	0.0499
AttitudeChange	0.6475	0.0499	0.6475	0.0499
lockNorthReady	1.3631	1.2448	1.3631	0.8234

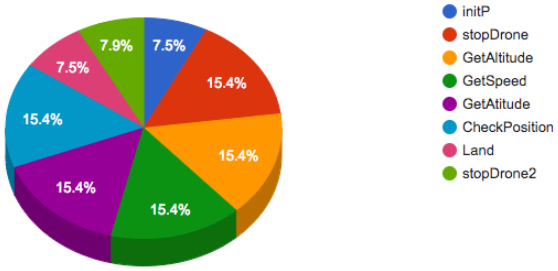


Symbol evolution





States visit



Clocks evolution

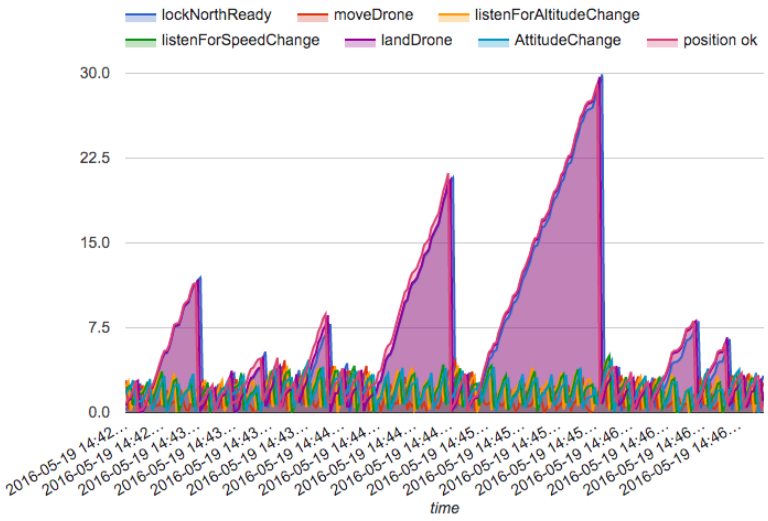


Figure D.9 – Drone Movement Manager: feedback results example screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

D.1.4 Networking service

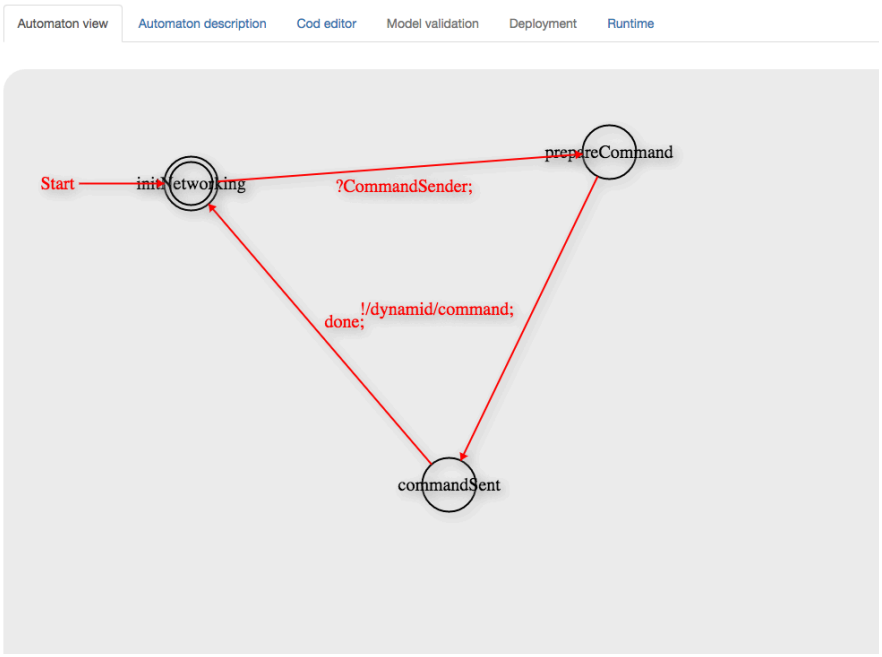


Figure D.10 – Networking: *ROSMDB* model screenshot

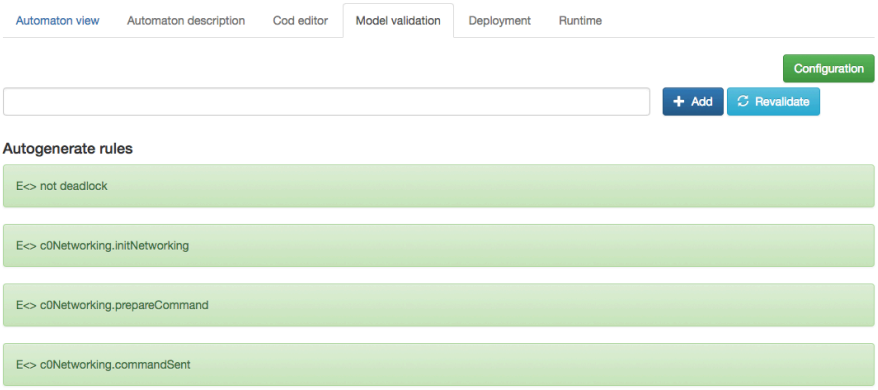


Figure D.11 – Networking: *ROSMDB* model-checking screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

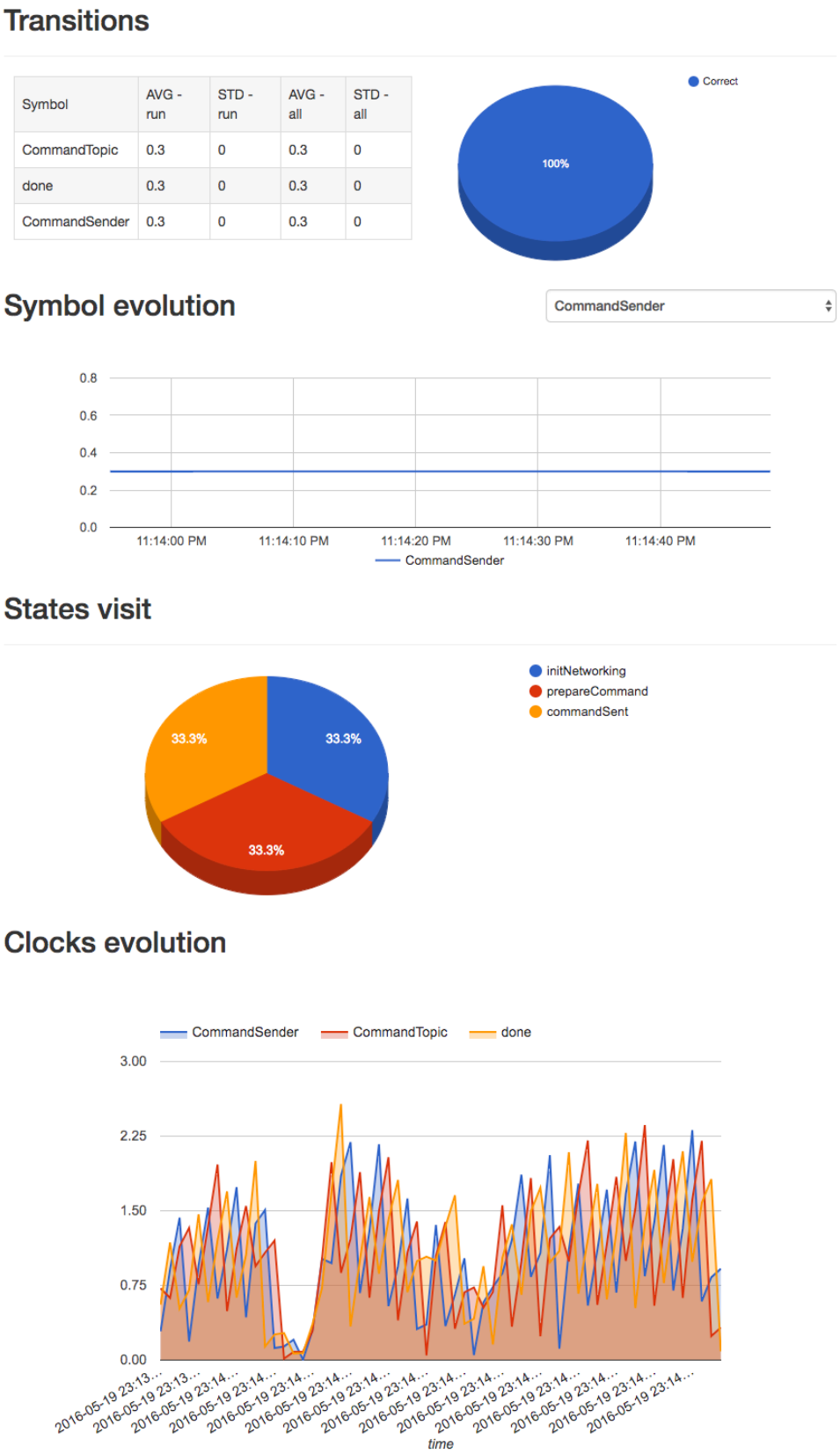


Figure D.12 – Networking: feedback results example screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

D.2 Controller application

D.2.1 Command sender service

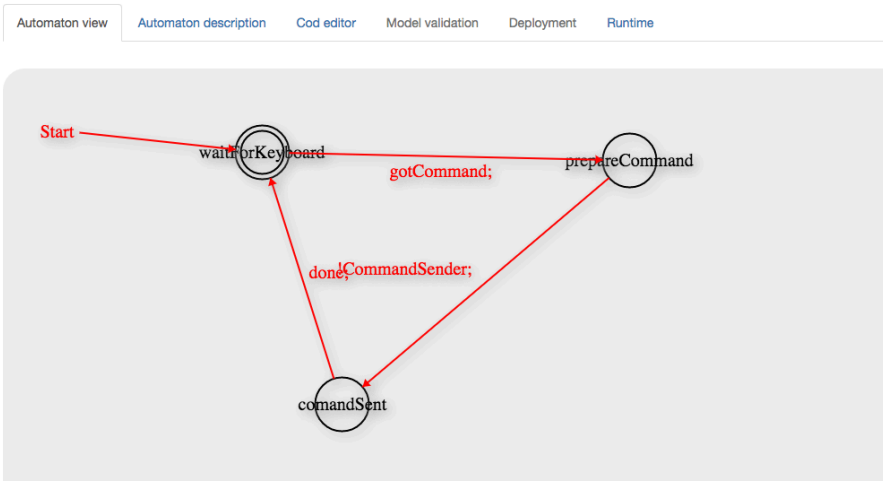


Figure D.13 – Command sender: *ROSMDB* model screenshot

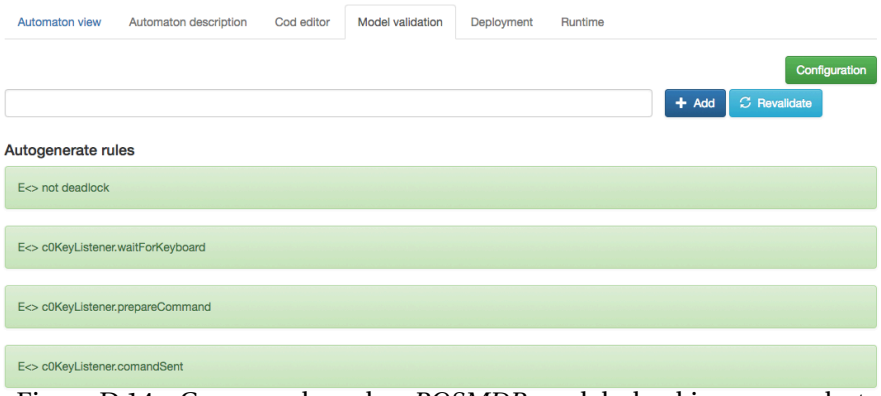


Figure D.14 – Command sender: *ROSMDB* model-checking screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

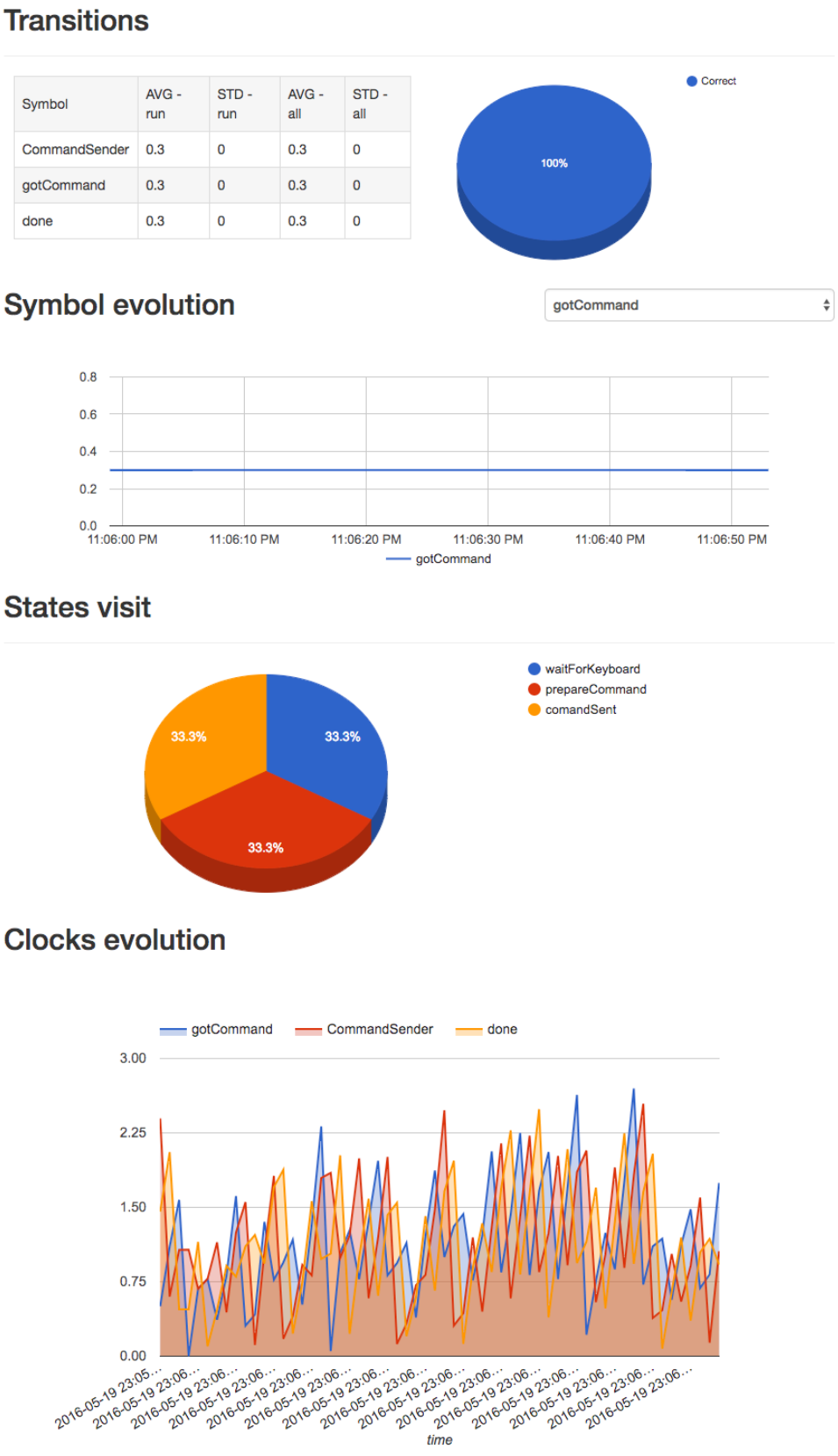


Figure D.15 – Command sender: feedback results example screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

E Random movement object search

This appendix displays the screenshots of the project modelled, verified in *ROSMDB* tool-chain as well as the feedback given by the tool.

E.1 Collision avoidance application

E.1.1 Engine stopper service

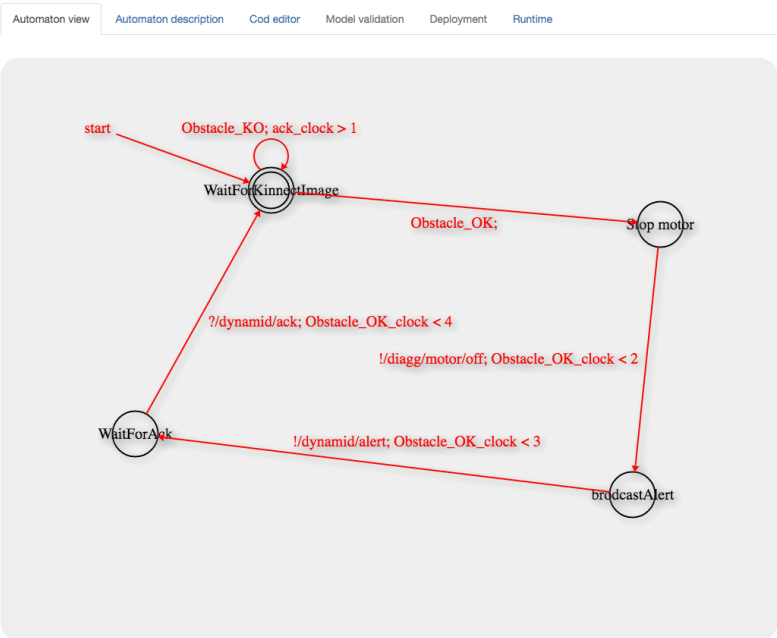


Figure E.1 – Engine stopper: *ROSMDB* model screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

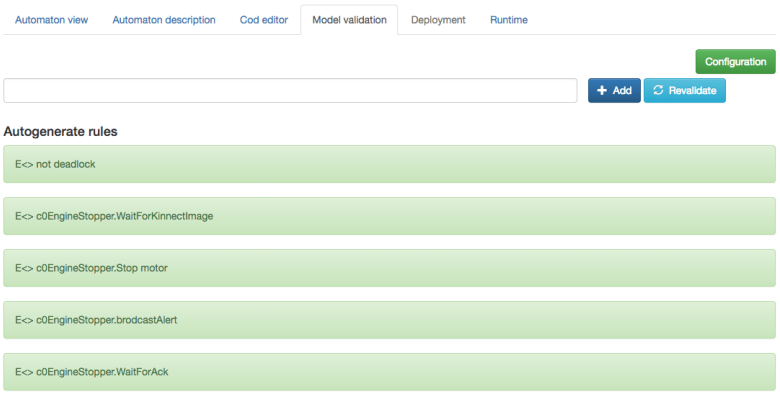


Figure E.2 – Engine stopper: *ROSMDB* model-checking screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

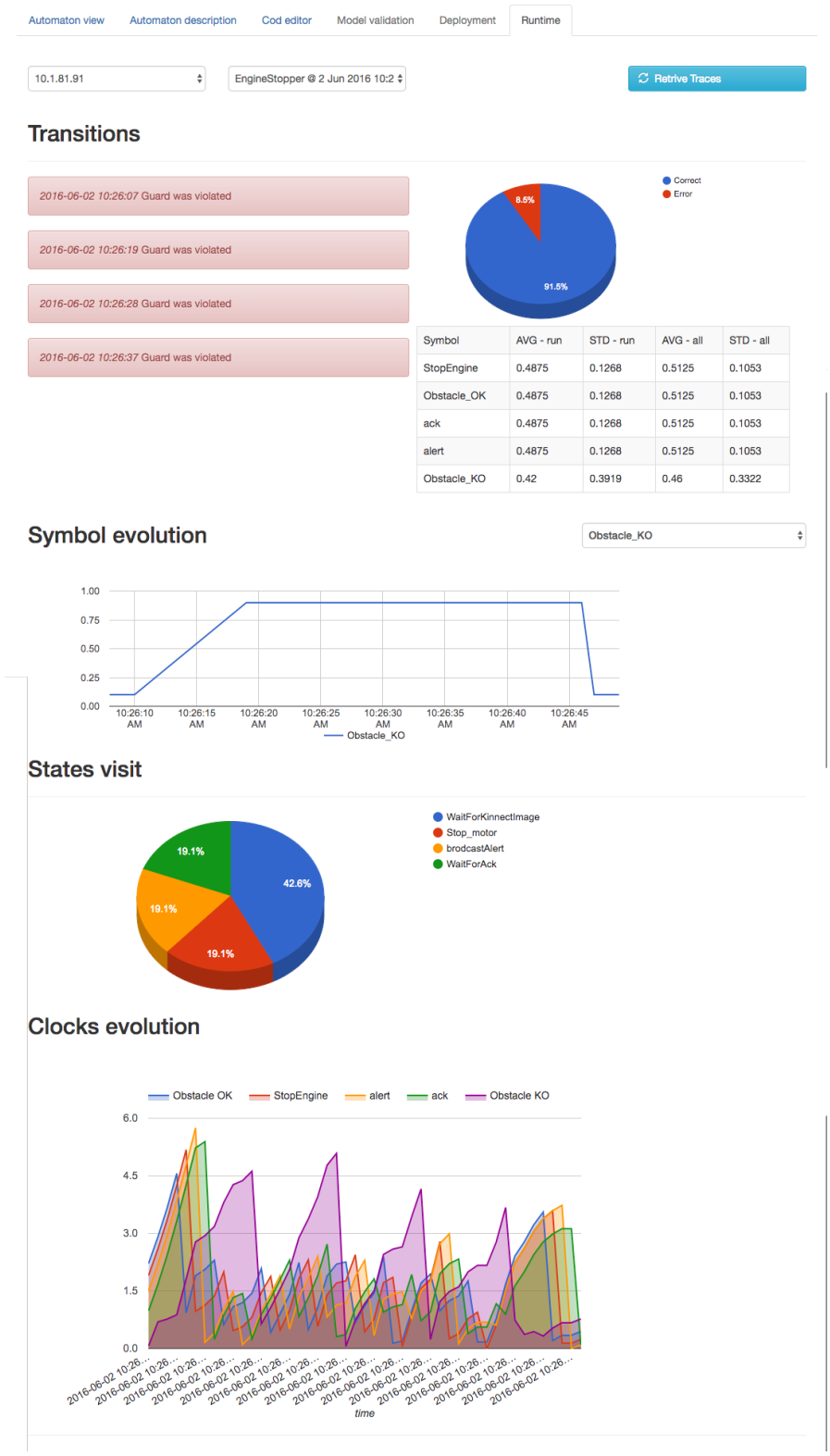


Figure E.3 – Engine stopper: feedback results example screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

E.1.2 Avoidance service

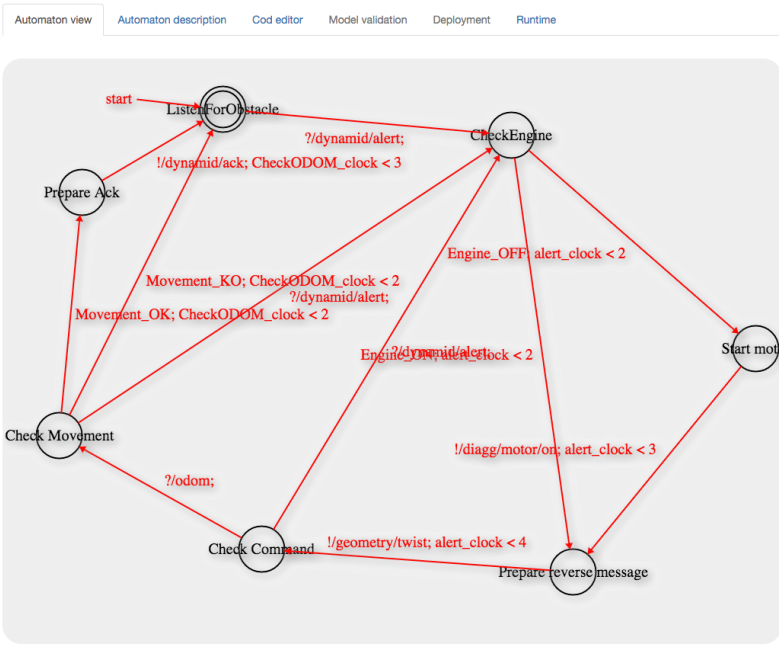


Figure E.4 – Avoidance: *ROSMDB* model screenshot

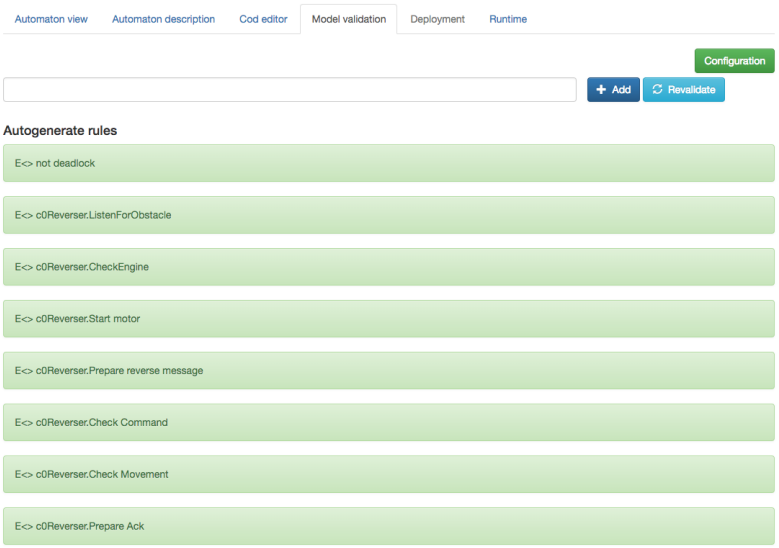


Figure E.5 – Avoidance: *ROSMDB* model-checking screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

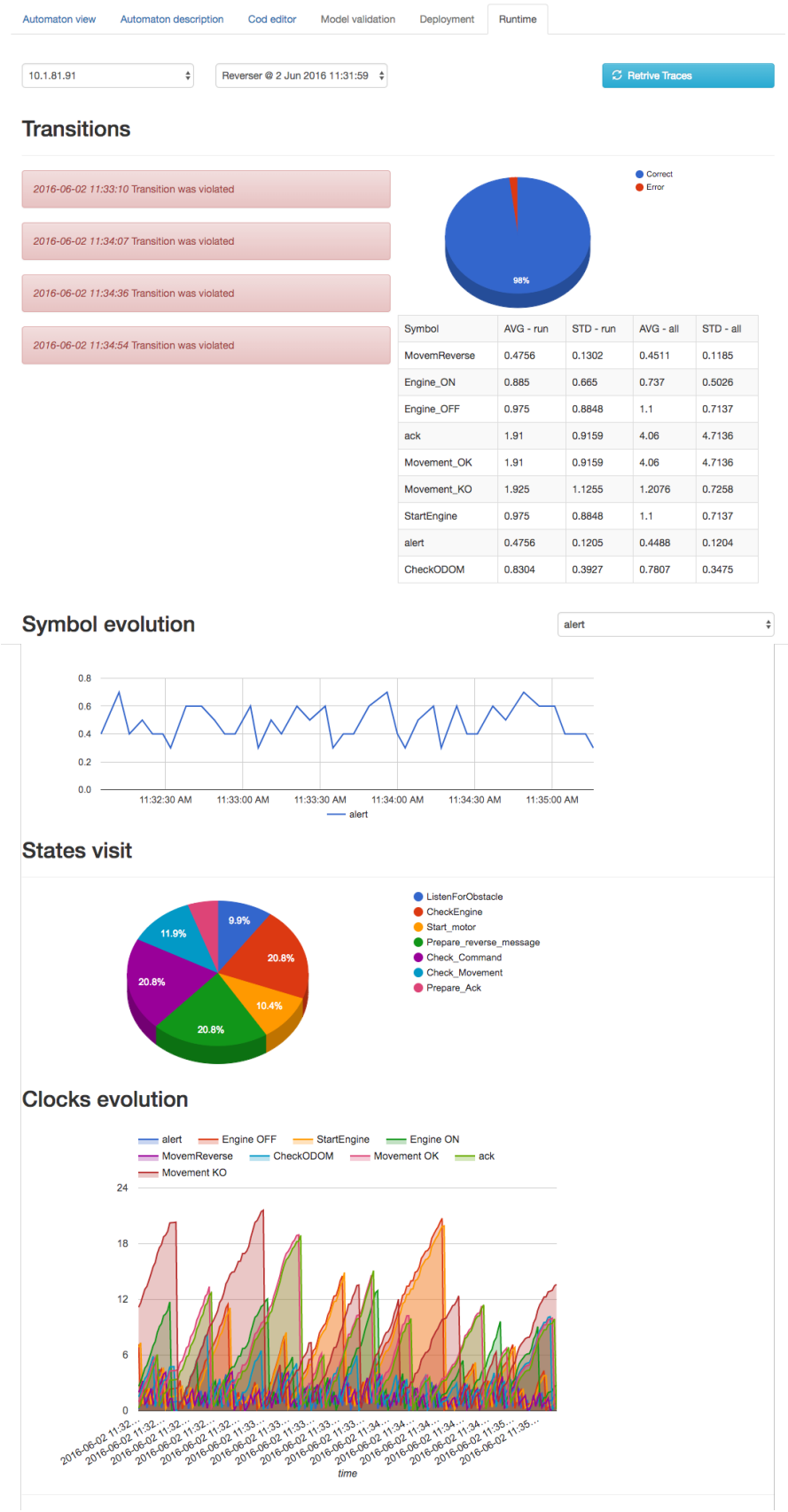


Figure E.6 – Avoidance: feedback results example screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

E.2 Object detection application

E.2.1 Image analyzer service

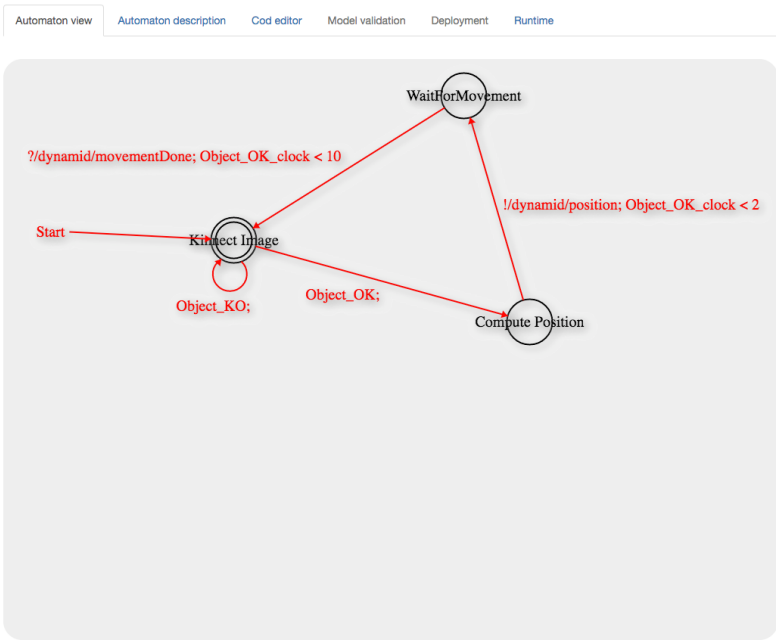


Figure E.7 – Image analyzer: *ROSMDB* model screenshot

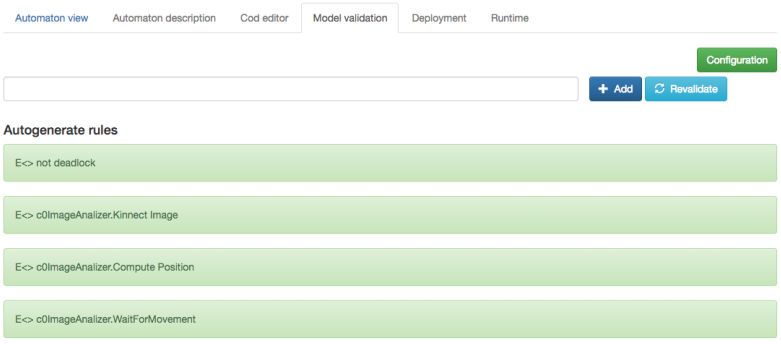


Figure E.8 – Image analyzer: *ROSMDB* model-checking screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés



Figure E.9 – Image analyzer: feedback results example screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

E.2.2 Mover service

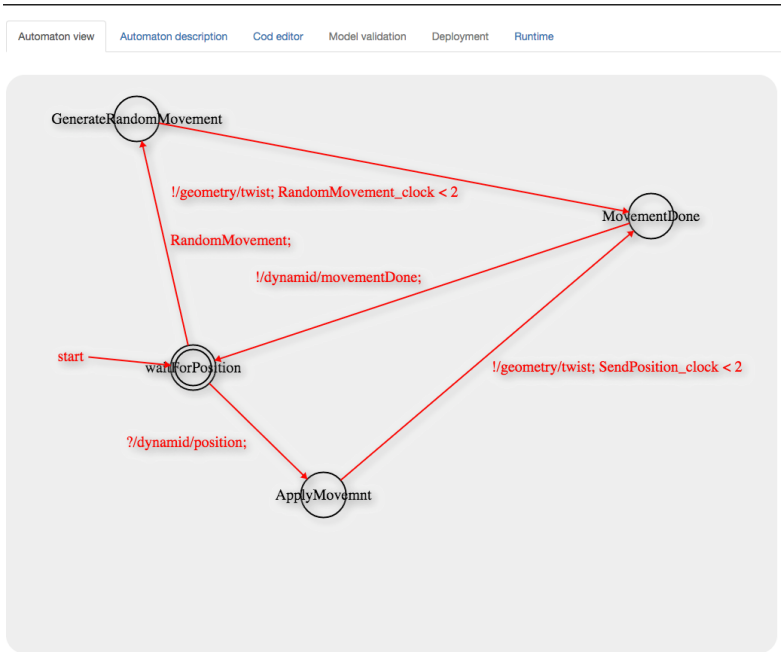


Figure E.10 – Mover: *ROSMDB* model screenshot

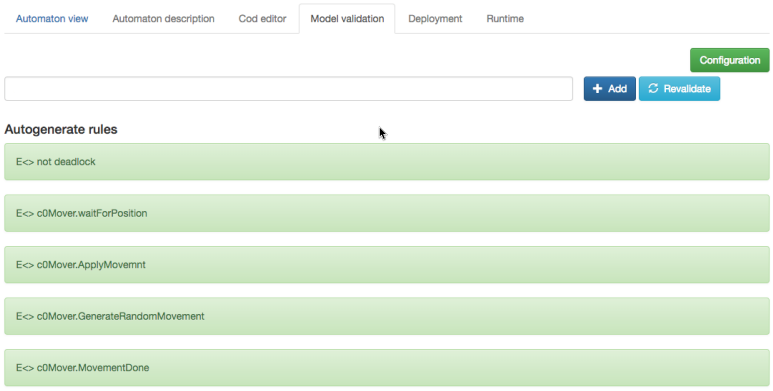


Figure E.11 – Mover: *ROSMDB* model-checking screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

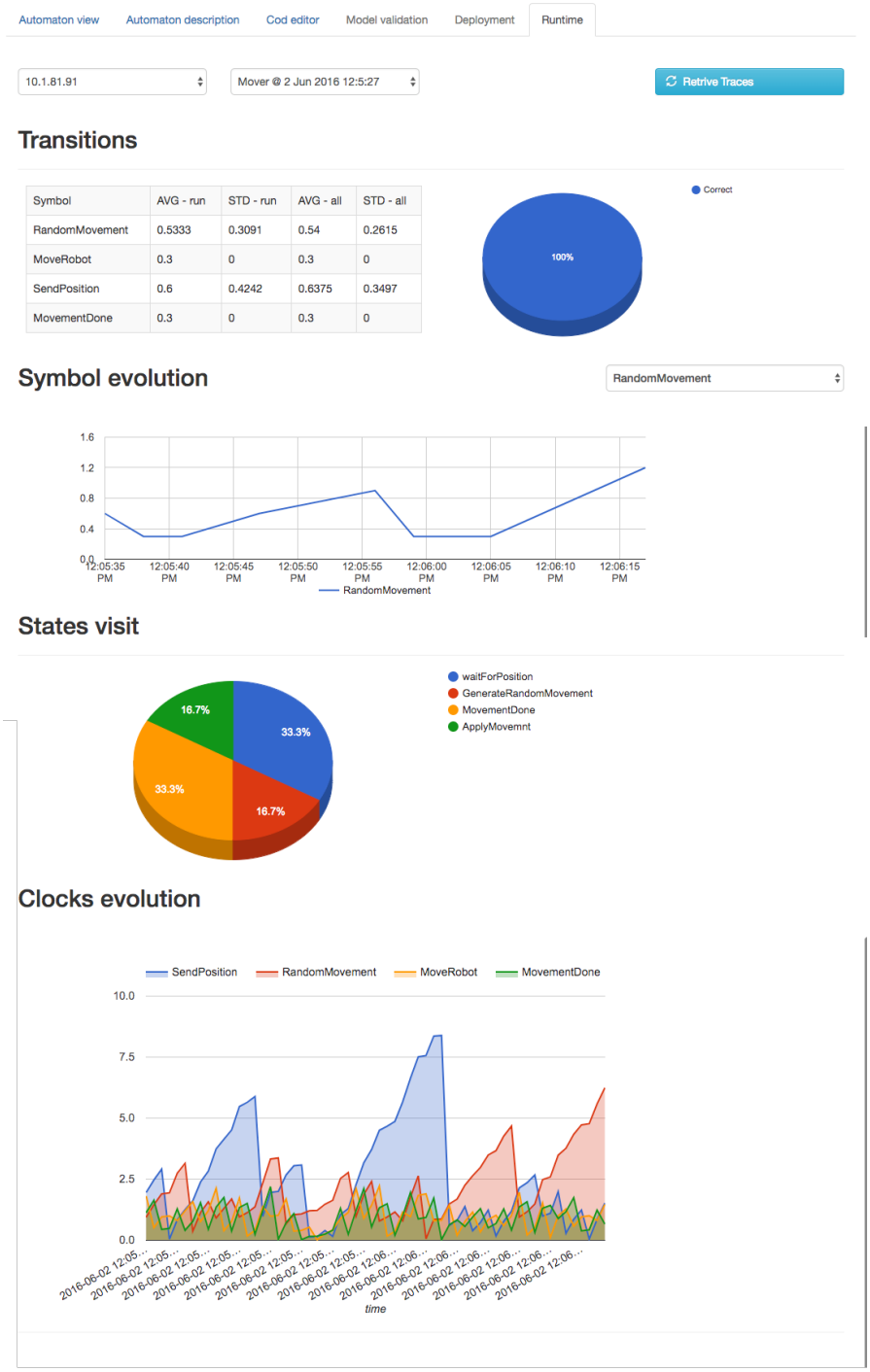


Figure E.12 – Mover: feedback results example screenshot

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Bibliography

[Ahn et al., 2005] Ahn, S. C., Kim, J. H., Lim, K., Ko, H., Kwon, Y.-M., and Kim, H.-G. (2005). Upnp approach for robot middleware. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 1959–1963. IEEE.

[Aichernig et al., 2017] Aichernig, B. K., Marcovic, S., and Schumi, R. (2017). Property-based testing with external test-case generators. In *Software Testing, Verification and Validation Workshops (ICSTW), 2017 IEEE International Conference on*, pages 337–346. IEEE.

[Al-Houmaily and Samaras, 2009] Al-Houmaily, Y. J. and Samaras, G. (2009). Three-phase commit. In *Encyclopedia of Database Systems*, pages 3091–3097. Springer.

[Alonso et al., 2010] Alonso, D., Vicente-Chicote, C., Ortiz, F., Pastor, J., and Alvarez, B. (2010). V3cmm: A 3-view component meta-model for model-driven robotic software development. *Journal of Software Engineering for Robotics*, 1(1):3–17.

[Altman, 1999] Altman, E. (1999). *Constrained Markov decision processes*, volume 7. CRC Press.

[Altman, 2000] Altman, E. (2000). *Applications of markov decision processes in communication networks: A survey*. PhD thesis, INRIA.

[Alur, 1999] Alur, R. (1999). Timed automata. In *International Conference on Computer Aided Verification*, pages 8–22. Springer.

[Alur, 2004] Alur, R. (2004). Timed automata. http://www.cis.upenn.edu/ alur/Talks/sfm-rt-04.ppt.

[Alur et al., 1993] Alur, R., Courcoubetis, C., and Dill, D. (1993). Model-checking in dense real-time. *Information and computation*, 104(1):2–34.

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

[Alur et al., 1996] Alur, R., Feder, T., and Henzinger, T. A. (1996). The benefits of relaxing punctuality. *Journal of the ACM (JACM)*, 43(1):116–146.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

[Alur et al., 1999] Alur, R., Fix, L., and Henzinger, T. A. (1999). Event-clock automata: a determinizable class of timed automata. *Theoretical Computer Science*, 211(1):253–273.

[Alur and Henzinger, 1994] Alur, R. and Henzinger, T. A. (1994). A really temporal logic. *Journal of the ACM (JACM)*, 41(1):181–203.

[Alur et al., 1997] Alur, R., Henzinger, T. A., and Wong-Toi, H. (1997). Symbolic analysis of hybrid systems. In *Decision and Control, 1997., Proceedings of the 36th IEEE Conference on*, volume 1, pages 702–707. IEEE.

[Alur and Madhusudan, 2004] Alur, R. and Madhusudan, P. (2004). Decision problems for timed automata: A survey. In *Formal Methods for the Design of Real-Time Systems*, pages 1–24. Springer.

[Arias et al., 2010] Arias, S., Boudin, F., Pissard-Gibollet, R., and Simon, D. (2010). Orccad, robot controller model and its support using eclipse modeling tools. In *5th National Conference on Control Architecture of Robots*.

[Arkin, 1998] Arkin, R. C. (1998). *Behavior-based robotics*. MIT press.

[Atkinson and Kühne, 2003] Atkinson, C. and Kühne, T. (2003). Model-driven development: a metamodeling foundation. *Software, IEEE*, 20(5):36–41.

[Baer et al., 2008] Baer, P. A., Reichle, R., and Geihs, K. (2008). The Spica Development Framework – Model-Driven Software Development for Autonomous Mobile Robots. In Burgard, W., Dillmann, R., Plagemann, C., and Vahrenkamp, N., editors, *IAS-10*, pages 211–220. IAS Society, Proceedings of the 10th International Conference on Intelligent Autonomous Systems.

[Baeten, 2005] Baeten, J. C. (2005). A brief history of process algebra. *Theoretical Computer Science*, 335(2):131–146.

[Bagnell and Schneider, 2001] Bagnell, J. A. and Schneider, J. G. (2001). Autonomous helicopter control using reinforcement learning policy search methods. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 2, pages 1615–1620. IEEE.

[Baldessari et al., 2007] Baldessari, R., Bödekker, B., Deegener, M., Festag, A., Franz, W., Kellum, C. C., Kosch, T., Kovacs, A., Lenardi, M., Menig, C., et al. (2007). Car-2-car communication consortium-manifesto.

[Barrett, 1995] Barrett, G. (1995). Model checking in practice: The t9000 virtual channel processor. *IEEE transactions on software engineering*, 21(2):69–78.

[Barto and Anandan, 1985] Barto, A. G. and Anandan, P. (1985). Pattern-recognizing stochastic learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, (3):360–375.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

[Bautin et al., 2012] Bautin, A., Simonin, O., and Charpillet, F. (2012). Stratégie d'exploration multirobot fondée sur les champs de potentiels artificiels. *Revue des Sciences et Technologies de l'Information-Série RIA: Revue d'Intelligence Artificielle*, 26(5):523–542.

[Beck and Krogh, 1986] Beck, C. and Krogh, B. (1986). Models for simulation and discrete control of manufacturing systems. In *Robotics and Automation. Proceedings. 1986 IEEE International Conference on*, volume 3, pages 305–310. IEEE.

[Bell, 2010] Bell, M. (2010). *SOA Modeling patterns for service-oriented discovery and analysis*. Wiley Online Library.

[Bellman, 1957] Bellman, R. (1957). A markovian decision process. Technical report, DTIC Document.

[Bergstra and Klop, 1986] Bergstra, J. and Klop, J. W. (1986). Process algebra: specification and verification in bisimulation semantics. *Math. & Comp. Sci. II*, 4.

[Bermudez-Edo et al., 2017] Bermudez-Edo, M., Elsaleh, T., Barnaghi, P., and Taylor, K. (2017). Iot-lite: a lightweight semantic model for the internet of things and its use with dynamic semantics. *Personal and Ubiquitous Computing*, pages 1–13.

[Berry and Gonthier, 1992] Berry, G. and Gonthier, G. (1992). The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152.

[Bert et al., 2005] Bert, D., Potet, M., and Stouls, N. (2005). Genesyst: A tool to reason about behavioral aspects of B event specifications. application to security properties. In *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005, Proceedings*, pages 299–318.

[Bezemer et al., 2011] Bezemer, M. M., Wilterdink, R. J., and Broenink, J. F. (2011). Luna: Hard real-time, multi-threaded, csp-capable execution framework.

[Bézivin et al., 2003] Bézivin, J., Farcet, N., Jézéquel, J.-M., Langlois, B., and Pollet, D. (2003). Reflective model driven engineering. In *International Conference on the Unified Modeling Language*, pages 175–189. Springer.

[Birkhoff and MacLane, 1948] Birkhoff, G. and MacLane, S. (1948). A survey of modern algebra. *New York*.

[Birrell and Nelson, 1981] Birrell, A. D. and Nelson, B. J. (1981). Remote procedure call.

[Blank et al., 2005] Blank, D., Kumar, D., Meeden, L., and Yanco, H. (2005). Pyro: an integrated environment for robotics education. In *AAAI'05: Proceedings of the 20th national conference on Artificial intelligence 2005*, pages 1718–1719. AAAI Press.

[Blank et al., 2006] Blank, D. S., Kumar, D., Meeden, L., and Yanco, H. A. (2006). The pyro toolkit for ai and robotics. *AI Magazine*, 27(1):39–50.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

[Borja et al., 2013] Borja, R., de la Pinta, J., Álvarez, A., and Maestre, J. (2013). Integration of service robots in the smart home by means of UPnP: A surveillance robot case study. *Robotics and Autonomous Systems*, 61(2):153 – 160.

[Bouyer, 2009] Bouyer, P. (2009). Model-checking timed temporal logics. *Electronic Notes in Theoretical Computer Science*, 231:323–341.

[Bouyer and Laroussinie, 2010] Bouyer, P. and Laroussinie, F. (2010). Model checking timed automata. *Modeling and Verification of Real-Time Systems: Formalisms and Software Tools*, pages 111–140.

[Bozga et al., 1998] Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., and Yovine, S. (1998). Kronos: A model-checking tool for real-time systems. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 298–302. Springer.

[Brookes, 1983] Brookes, S. D. (1983). A model for communicating sequential processes.

[Brookes et al., 1984] Brookes, S. D., Hoare, C. A., and Roscoe, A. W. (1984). A theory of communicating sequential processes. *Journal of the ACM (JACM)*, 31(3):560–599.

[Brookes and Roscoe, 1984] Brookes, S. D. and Roscoe, A. (1984). An improved failures model for communicating processes. In *International Conference on Concurrency*, pages 281–305. Springer.

[Bruemmer et al., 2006] Bruemmer, D. J., Few, D. A., Walton, M. C., and Nielsen, C. W. (2006). The robot intelligence kernel. In *AAAI*.

[Brugali and Scandurra, 2009] Brugali, D. and Scandurra, P. (2009). Component-based robotic engineering (part i)[tutorial]. *IEEE Robotics & Automation Magazine*, 16(4):84–96.

[Bruyninckx et al., 2013] Bruyninckx, H., Klotzbücher, M., Hochgeschwender, N., Kraetzschmar, G., Gherardi, L., and Brugali, D. (2013). The brics component model: a model-based development paradigm for complex robotics software systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1758–1764. ACM.

[Calisi and Censi, 2009] Calisi, D. and Censi, A. (2009). Robotic software development and interoperability using the openrdk framework. In *ICAR'09 Workshop on "Rapid Application Development in Robotics: On the role of re-use and adaptation of system components, middleware, and control architectures"*, Munich, Germany.

[Capra et al., 2001] Capra, L., Emmerich, W., and Mascolo, C. (2001). Middleware for mobile computing: Awareness vs. transparency (position summary).

[CARMEN, 2008] CARMEN (2008). the carnegie mellon robot navigation toolkit. http://carmen.sourceforge.net/.

[Chacon and Straub, 2014] Chacon, S. and Straub, B. (2014). *Pro git*. Apress.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

[Channabasavaiah et al., 2003] Channabasavaiah, K., Holley, K., and Tuggle, E. (2003). Migrating to a service-oriented architecture. *IBM DeveloperWorks*, 16.

[Chishiro et al., 2009] Chishiro, H., Fujita, Y., Takeda, A., Kojima, Y., Funaoka, K., Kato, S., and Yamasaki, N. (2009). Extended rt-component framework for rt-middleware. pages 161 –168.

[Chitic et al., 2015] Chitic, S.-G., Ponge, J., and Simonin, O. (2015). Are middlewares ready for multi-robots systems? In *Simulation, Modeling, and Programming for Autonomous Robots, Volume 8810 of the series Lecture Notes in Computer Science*, pages 279–290.

[Chitic et al., 2016] Chitic, S.-G., Ponge, J., and Simonin, O. (2016). Sdfr-service discovery for multi-robot systems. In *ICAART 2016 The 8th International Conference on Agents and Artificial Intelligence*.

[Choi et al., 2006] Choi, D.-H., Kim, S.-H., Lee, K.-K., Beak, B.-H., and Park, H.-S. (2006). Middleware architecture for module-based robot. *SICE-ICASE International Joint Conference*, 0:4202–4205.

[Chollet et al., 2015] Chollet, S., Lalanda, P., and Escoffier, C. (2015). Extension of service-oriented component models for dynamic environment. In *Services Computing (SCC), 2015 IEEE International Conference on*, pages 648–655. IEEE.

[Choset, 2001] Choset, H. (2001). Coverage for robotics–a survey of recent results. *Annals of mathematics and artificial intelligence*, 31(1-4):113–126.

[Chow, 1978] Chow, T. S. (1978). Testing software design modeled by finite-state machines. *IEEE transactions on software engineering*, 4(3):178.

[Christensen and Pirjanian, 1997] Christensen, H. I. and Pirjanian, P. (1997). Theoretical methods for planning and control in mobile robotics. In *Knowledge-Based Intelligent Electronic Systems, 1997. KES'97. Proceedings., 1997 First International Conference on*, volume 1, pages 81–86. IEEE.

[Clarke et al., 1986] Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263.

[Coelingh and Solyom, 2012] Coelingh, E. and Solyom, S. (2012). All aboard the robotic road train. *Ieee Spectrum*, 49(11).

[Collett et al., 2005] Collett, T. H., MacDonald, B. A., and Gerkey, B. P. (2005). Player 2.0: Toward a practical robot programming framework.

[Corbett et al., 2000] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., Zheng, H., et al. (2000). Bandera: Extracting finite-state models from java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 439–448. IEEE.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

[Costelha and Lima, 2007] Costelha, H. and Lima, P. (2007). Modelling, analysis and execution of robotic tasks using petri nets. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1449–1454. IEEE.

[Côté et al., 2006] Côté, C., Brosseau, Y., Létourneau, D., Raïevsky, C., and Michaud, F. (2006). Robotic software integration using marie. *International Journal of Advanced Robotic Systems*, 3(1):55–60.

[Côté et al., 2007] Côté, C., Brosseau, Y., Létourneau., D., Raïevsky, C., Brosseau, Y., and Michaud, F. (2007). Using marie for mobile robot component development and integration. *Software Engineering for Experimental Robotics Book Series Springer Tracts in Advanced Robotics Publisher Springer Berlin / Heidelberg*, 30/2007:211–230.

[Cotner et al., 1999] Cotner, C. L., Crus, R. A., Howell, B. K., Pickel, J. W., and Wisneski, D. J. (1999). System, method and program for performing two-phase commit with a coordinator that performs no logging. US Patent 5,884,327.

[Courcoubetis and Yannakakis, 1992] Courcoubetis, C. and Yannakakis, M. (1992). Minimum and maximum delay problems in real-time systems. *Formal Methods in System Design*, 1(4):385–415.

[Cousins et al., 2010] Cousins, S., Gerkey, B., Conley, K., and Garage, W. (2010). Sharing software with ros [ros topics]. *Robotics & Automation Magazine, IEEE*, 17(2):12–14.

[Cranen et al., 2013] Cranen, S., Groote, J. F., Keiren, J. J., Stappers, F. P., de Vink, E. P., Wesselink, W., and Willemse, T. A. (2013). An overview of the mcrl2 toolset and its recent advances. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 199–213. Springer.

[Creese, 2001] Creese, S. (2001). Data independent induction: Csp model checking of arbitrary sized networks.

[Crockett et al., 1987] Crockett, D., Desrochers, A., DiCesare, F., and Ward, T. (1987). Implementation of a petri net controller for a machining workstation. In *Robotics and Automation. Proceedings. 1987 IEEE International Conference on*, volume 4, pages 1861–1867. IEEE.

[Curry, 2004] Curry, E. (2004). Message-oriented middleware. *Middleware for communications*, pages 1–28.

[Dave, 2009] Dave (2009). Dave's robotic operating system. Online: http://dros.org/.

[David et al., 2003] David, A., Moller, M. O., and Yi, W. (2003). Verification of uml statechart with real-time extensions. *FASE 2002*, pages 218–232.

[David and Alla, 2010] David, R. and Alla, H. (2010). *Discrete, continuous, and hybrid Petri nets*. Springer Science & Business Media.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

[del Val et al., 2014] del Val, E., Rebollo, M., and Botti, V. (2014). Enhancing decentralized service discovery in open service-oriented multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 28(1):1–30.

[Desolda et al., 2017] Desolda, G., Ardito, C., and Matera, M. (2017). Empowering end users to customize their smart environments: Model, composition paradigms, and domain-specific tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 24(2):12.

[Dhouib et al., 2012] Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., and Ziane, M. (2012). Robotml, a domain-specific language to design, simulate and deploy robotic applications. In *Simulation, Modeling, and Programming for Autonomous Robots*, pages 149–160. Springer.

[Diaz et al., 2011] Diaz, D., R-Moreno, M. D., Cesta, A., Oddi, A., and Rasconi, R. (2011). Toward a csp-based approach for energy management in rovers. In *Space Mission Challenges for Information Technology (SMC-IT), 2011 IEEE Fourth International Conference on*, pages 121–128. IEEE.

[Djonin and Krishnamurthy, 2007] Djonin, D. V. and Krishnamurthy, V. (2007). Mimo transmission control in fading channels-a constrained markov decision process formulation with monotone randomized policies. *IEEE Transactions on Signal Processing*, 55(10):5069–5083.

[Egerstedt, 2000] Egerstedt, M. (2000). Behavior based robotics using hybrid automata. In *HSCC*, pages 103–116. Springer.

[Elkady and Sobh, 2012] Elkady, A. and Sobh, T. (2012). Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of Robotics*, 2012.

[Endo et al., 2004] Endo, Y., MacKenzie, D., and Arkin, R. (2004). Usability evaluation of high-level user assistance for robot mission specification. *IEEE Transactions on Systems, Man, and Cybernetics*, 34:168–180.

[ERSP, 2010] ERSP (2010). Ersp 3.1 software development kit. Online: http://www.evolution.com/products/ersp/.

[Ferber, 1999] Ferber, J. (1999). *Multi-Agent Systems. An Introduction to Distributed Artificial Intelligence*. Addison Wesley, London.

[Fernandez-Madrigal et al., 2006] Fernandez-Madrigal, J., Galindo, C., and Gonzalez, J. (2006). Integrating heterogeneous robotic software. In *Electrotechnical Conference, 2006. MELECON 2006. IEEE Mediterranean*, pages 433 –436.

[Ferrara, 2004] Ferrara, A. (2004). Web services: a process algebra approach. In *Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251. ACM.

[Fielding, 2000] Fielding, R. (2000). Representational state transfer. *Architectural Styles and the Design of Netowork-based Software Architecture*, pages 76–85.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

[Fitzpatrick et al., 2008] Fitzpatrick, P., Metta, G., and Natale, L. (2008). Towards long-lived robot genes. *Robot. Auton. Syst.*, 56(1):29–45.

[Freedman, 1991] Freedman, P. (1991). Time, petri nets, and robotics. *IEEE transactions on robotics and automation*, 7(4):417–433.

[Frénot et al., 2010] Frénot, S., Le Mouël, F., Ponge, J., and Salagnac, G. (2010). Various Extensions for the Ambient OSGi framework. In *Adamus Workshop in ICPS*, Berlin, Allemagne.

[Gajski et al., 1994] Gajski, D. D., Vahid, F., Narayan, S., and Gong, J. (1994). *Specification and design of embedded systems*. Prentice Hall Englewood Cliffs.

[Garavel et al., 2013] Garavel, H., Lang, F., Mateescu, R., and Serwe, W. (2013). Cadp 2011: a toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107.

[Gerbert-Gaillard and Lalanda, 2016] Gerbert-Gaillard, E. and Lalanda, P. (2016). Self-aware model-driven pervasive systems. In *Autonomic Computing (ICAC), 2016 IEEE International Conference on*, pages 221–222. IEEE.

[Gill et al., 1962] Gill, A. et al. (1962). Introduction to the theory of finite-state machines.

[Google,] Google. Protocol buffers. https://developers.google.com/protocol-buffers/.

[Grigorik, 2013] Grigorik, I. (2013). Making the web faster with http 2.0. *Commun. ACM*, 56(12):42–49.

[Gummadi et al., 2002] Gummadi, P. K., Saroiu, S., and Gribble, S. D. (2002). A measurement study of napster and gnutella as examples of peer-to-peer file sharing systems. *ACM SIGCOMM Computer Communication Review*, 32(1):82–82.

[Guo and Hernández-Lerma, 2009] Guo, X. and Hernández-Lerma, O. (2009). Continuous-time markov decision processes. In *Continuous-Time Markov Decision Processes*, pages 9–18. Springer.

[Hall and Chapman, 2002] Hall, A. and Chapman, R. (2002). Correctness by construction: Developing a commercial secure system. *IEEE software*, 19(1):18–25.

[Hardin et al., 1996] Hardin, R. H., Har'El, Z., and Kurshan, R. P. (1996). Cospan. In *International Conference on Computer Aided Verification*, pages 423–427. Springer.

[Hazelhurst, 2008] Hazelhurst, S. (2008). Scientific computing using virtual high-performance computing: a case study using the amazon elastic computing cloud. In *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology*, pages 94–103. ACM.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

[Heckel et al., 2006] Heckel, F., Blakely, T., Dixon, M., Wilson, C., and Smart, W. D. (2006). The wurde robotics middleware and ride multi-robot tele-operation interface. *AAAI Mobile Robotics Workshop, 2006*.

[Henzinger et al., 1997] Henzinger, T. A., Ho, P.-H., and Wong-Toi, H. (1997). Hytech: A model checker for hybrid systems. In *International Conference on Computer Aided Verification*, pages 460–463. Springer.

[Henzinger et al., 1994] Henzinger, T. A., Nicollin, X., Sifakis, J., and Yovine, S. (1994). Symbolic model checking for real-time systems. *Information and computation*, 111(2):193–244.

[Hermanns et al., 2002] Hermanns, H., Herzog, U., and Katoen, J.-P. (2002). Process algebra for performance evaluation. *Theoretical computer science*, 274(1):43–87.

[Hershey et al., 1995] Hershey, P. C., Johnson, D. B., Le, A. V., Matyas, S. M., Waclawsky, J. G., and Wilkins, J. D. (1995). Network security system and method using a parallel finite state machine adaptive active monitor and responder. US Patent 5,414,833.

[Hessel and Pettersson, 2004] Hessel, A. and Pettersson, P. (2004). A test case generation algorithm for real-time systems. In *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, pages 268–273. IEEE.

[Hilaire et al., 2008] Hilaire, V., Gruer, P., Koukam, A., and Simonin, O. (2008). Formal driven prototyping approach for multiagent systems. *International Journal of Agent-Oriented Software Engineering*, 2(2):246–266.

[Hilderink et al., 2003] Hilderink, G. H., Jovanovic, D. S., and Broenink, J. F. (2003). A mulit-model robotic control law modelled and implemented with the csp/ct framework.

[Hoare, 1983] Hoare, C. A. R. (1983). Communicating sequential processes. *Communications of the ACM*, 26(1):100–106.

[Hopcroft, 1979] Hopcroft, J. E. (1979). *Introduction to Automata Theory, Languages and Computation: For VTU, 3/e*. Pearson Education India.

[iRobotware, 2010] iRobotware (2010). Aware 2 robot intelligient software. Online: http://www.irobot.com/gi/developers/Aware/.

[Issarny et al., 2007] Issarny, V., Caporuscio, M., and Georgantas, N. (2007). A perspective on the future of middleware-based software engineering. In *Future of Software Engineering, 2007. FOSE '07*, pages 244–258.

[Issarny et al., 2011] Issarny, V., Georgantas, N., Hachem, S., Zarras, A., Vassiliadist, P., Autili, M., Gerosa, M. A., and Hamida, A. B. (2011). Service-oriented middleware for the future internet: state of the art and research directions. *Journal of Internet Services and Applications*, 2(1):23–45.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

[Iversen et al., 2000] Iversen, T. K., Kristoffersen, K. J., Larsen, K. G., Laursen, M., Madsen, R. G., Mortensen, S. K., Pettersson, P., and Thomasen, C. B. (2000). Model-checking real-time control programs: verifying lego mindstorms tm systems using uppaal. In *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, pages 147–155. IEEE.

[Jafari, 1992] Jafari, M. A. (1992). An architecture for a shop-floor controller using colored petri nets. *International journal of flexible manufacturing systems*, 4(2):159–181.

[Jang et al., 2010] Jang, C., Lee, S.-I., Jung, S.-W., Song, B., Kim, R., Kim, S., and Lee, C.-H. (2010). Opros: A new component-based robot software platform. *ETRI Journal*.

[Jaulin et al., 2012] Jaulin, L., Le Bars, F., Clement, B., Gallou, Y., Menage, O., Reynet, O., Sliwka, J., and Zerr, B. (2012). Suivi de route pour un robot voilier. In *Conférence Internationale Francophone d'Automatique (CIFA2012)*, pages 695–702.

[Jensen, 1989] Jensen, K. (1989). Coloured petri nets: A high level language for system design and analysis. In *International Conference on Application and Theory of Petri Nets*, pages 342–416. Springer.

[Jeronimo and Weast, 2003] Jeronimo, M. and Weast, J. (2003). *UPnP design by example: a software developer's guide to universal plug and play*. Intel Press.

[Johns and Taylor, 2008] Johns, K. and Taylor, T. (2008). *Professional Microsoft Robotics Developer Studio*. Wrox Press Ltd., Birmingham, UK, UK.

[Jones and Jones, 1987] Jones, G. and Jones, G. (1987). *Programming in OCCAM*. Prentice-Hall International New York, NY.

[Jouault et al., 2008] Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). Atl: A model transformation tool. *Science of computer programming*, 72(1):31–39.

[Kaelbling et al., 1998] Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1):99–134.

[Karaman et al., 2009] Karaman, S., Rasmussen, S., Kingston, D., and Frazzoli, E. (2009). Specification and planning of uav missions: a process algebra approach. In *2009 American Control Conference*, pages 1442–1447. IEEE.

[King et al., 2003] King, J., Pretty, R. K., and Gosine, R. G. (2003). Coordinated execution of tasks in a multiagent environment. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 33(5):615–619.

[Kleppe et al., 2003] Kleppe, A. G., Warmer, J. B., and Bast, W. (2003). *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional.

[Klusch et al., 2006] Klusch, M., Fries, B., and Sycara, K. (2006). Automated semantic web service discovery with owls-mx. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 915–922. ACM.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

[KnowRob, 2014] KnowRob (2014). Knowledge processing for robots. http://www.knowrob.org/.

[Kodate et al., 1987] Kodate, H., Fujii, K., and Yamanoi, K. (1987). Representation of fms with petri net graph and its application to simulation of system operation. *Robotics and Computer-Integrated Manufacturing*, 3(3):275–283.

[Koenig and Simmons, 1998] Koenig, S. and Simmons, R. (1998). Xavier: A robot navigation architecture based on partially observable markov decision process models. *Artificial Intelligence Based Mobile Robotics: Case Studies of Successful Robot Systems*, pages 91–122.

[König et al., 2009] König, L., Mostaghim, S., and Schmeck, H. (2009). Decentralized evolution of robotic behavior using finite state machines. *International Journal of Intelligent Computing and Cybernetics*, 2(4):695–723.

[Košecká et al., 1997] Košecká, J., Christensen, H. I., and Bajcsy, R. (1997). Experiments in behavior composition. *Robotics and Autonomous systems*, 19(3):287–298.

[Kotb et al., 2007] Kotb, Y. T., Beauchemin, S. S., and Barron, J. L. (2007). Petri net-based cooperation in multi-agent systems. In *Computer and Robot Vision, 2007. CRV'07. Fourth Canadian Conference on*, pages 123–130. IEEE.

[Koymans, 1990] Koymans, R. (1990). Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299.

[Kraetzschmar et al., 2002] Kraetzschmar, G. K., Utz, H., Sablatnög, S., Enderle, S., and Palm, G. (2002). Miro - middleware for cooperative robotics. In *RoboCup 2001: Robot Soccer World Cup V*, pages 411–416, London, UK. Springer-Verlag.

[Kranz et al., 2006] Kranz, M., Rusu, R. B., Maldonado, A., Beetz, M., and Schmidt, A. (2006). A player/stage system for context-aware intelligent environments.

[Krüger et al., 2006] Krüger, D., Lil, I., Sünderhauf, N., Baumgartl, R., and Protzel, P. (2006). Using and extending the miro middleware for autonomous robots. In *Towards Autonomous Robotic Systems (TAROS), Guildford, September 2006*.

[Kwak et al., 2006] Kwak, J.-Y., Yoon, J. Y., and Shinn, R. (2006). An intelligent robot architecture based on robot mark-up languages. In *Engineering of Intelligent Systems, 2006 IEEE International Conference*, pages 1 –6.

[Lam et al., 2017] Lam, A. N., Nguyen, A. T., Nguyen, H. A., and Nguyen, T. N. (2017). Bug localization with combination of deep learning and information retrieval. In *Proceedings of the 25th International Conference on Program Comprehension*, pages 218–229. IEEE Press.

[Lamond and Boukhtouta, 2002] Lamond, B. F. and Boukhtouta, A. (2002). Water reservoir applications of markov decision processes. In *Handbook of Markov Decision Processes*, pages 537–558. Springer.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

[Lankenau and Meyer, 1999] Lankenau, A. and Meyer, O. (1999). Formal methods in robotics: Fault tree based verification. In *In: Proc. of Quality Week*. Citeseer.

[Larsen et al., 1995] Larsen, K. G., Pettersson, P., and Yi, W. (1995). Model-checking for real-time systems. In *International Symposium on Fundamentals of Computation Theory*, pages 62–88. Springer.

[Liggett and Lippman, 1969] Liggett, T. M. and Lippman, S. A. (1969). Stochastic games with perfect information and time average payoff. *Siam Review*, 11(4):604–607.

[Lindahl et al., 2001] Lindahl, M., Pettersson, P., and Yi, W. (2001). Formal design and analysis of a gear controller. *International Journal on Software Tools for Technology Transfer*, 3(3):353–368.

[Loeliger and McCullough, 2012] Loeliger, J. and McCullough, M. (2012). *Version Control with Git: Powerful tools and techniques for collaborative software development.* " O'Reilly Media, Inc.".

[Lovejoy, 1991] Lovejoy, W. S. (1991). A survey of algorithmic methods for partially observed markov decision processes. *Annals of Operations Research*, 28(1):47–65.

[Lyons and Arbib, 1989] Lyons, D. M. and Arbib, M. A. (1989). A formal model of computation for sensory-based robotics. *IEEE Transactions on Robotics and Automation*, 5(3):280–293.

[Magnenat et al., 2010] Magnenat, S., Retornaz, P., Bonani, M., Longchamp, V., and Mondada, F. (2010). Aseba: A modular architecture for event-based control of complex robots. *Mechatronics, IEEE/ASME Transactions on*, PP(99):1 –9.

[Makarenko et al., 2007] Makarenko, A., Brooks, A., , and Kaupp, T. (2007). On the benefits of making robotic software frameworks thin. In *POn the Benefits of Making Robotic Software Frameworks Thin IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'07), San Diego CA, USA, 29 Oct. - 02 Nov. 2007*.

[Makarenko et al., 2006] Makarenko, A., Brooks, A., and Kaupp, T. (2006). Orca: Components for robotics. In *Proceedings of 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'06) Workshop on Robotic Standardization, Beijing, China, 11 Oct. - 13 Oct. 2006*.

[Marciano, 2013] Marciano, L. (2013). *CPNP: Colored Petri Net Represention of Single-Robot and Multi-Robot Plans.* PhD thesis, Citeseer.

[Martinoli et al., 2004] Martinoli, A., Easton, K., and Agassounon, W. (2004). Modeling swarm robotic systems: A case study in collaborative distributed manipulation. *The International Journal of Robotics Research*, 23(4-5):415–436.

[Maymounkov and Mazieres, 2002] Maymounkov, P. and Mazieres, D. (2002). Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*, pages 53–65. Springer.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

[McCreesh and Daniel, 2014] McCreesh, J. and Daniel, E. (2014). Interoperability definition. http://interoperability-definition.info/en/.

[McLendon Jr and Vidale, 1992] McLendon Jr, W. W. and Vidale, R. F. (1992). Analysis of an ada system using coloured petri nets and occurrence graphs. In *International Conference on Application and Theory of Petri Nets*, pages 384–388. Springer.

[Mell and Grance, 2011] Mell, P. and Grance, T. (2011). The nist definition of cloud computing.

[Michel, 2004] Michel, O. (2004). Cyberbotics ltd. webots tm : Professional mobile robot simulation. *Int. Journal of Advanced Robotic Systems*, 1:39–42.

[Michi Henning, 2010] Michi Henning, M. S. (2010). Distributed programming with ice. http://www.zeroc.com/doc/Ice-3.4.0/manual/.

[Mihaylova et al., 2002] Mihaylova, L., Lefebvre, T., Bruyninckx, H., Gadeyne, K., and De Schutter, J. (2002). A comparison of decision making criteria and optimization methods for active robotic sensing. In *International Conference on Numerical Methods and Applications*, pages 316–324. Springer.

[Mohamed et al., 2008] Mohamed, N., Al-Jaroodi, J., and Jawhar, I. (2008). Middleware for robotics: A survey. In *Robotics, Automation and Mechatronics, 2008 IEEE Conference on*, pages 736–742. IEEE.

[Montemerlo et al., 2003] Montemerlo, M., Roy, N., and Thrun, S. (2003). Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (carmen) toolkit. In *In Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)2003*, pages 2436–2441.

[MRDS, 2012] MRDS (2012). Microsoft robotics developer studio. http://msdn.microsoft.com/en-us/library/bb648760.aspx.

[MRPT, 2010] MRPT (2010). The mobile robot programming toolkit. Online: http://www.mrpt.org/.

[Murata, 1989] Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580.

[Murata et al., 1986] Murata, T., Komoda, N., Matsumoto, K., and Haruna, K. (1986). A petri net-based controller for flexible and maintainable sequence control and its applications in factory automation. *IEEE Transactions on Industrial Electronics*, (1):1–8.

[Murata et al., 1989] Murata, T., Shenker, B., and Shatz, S. M. (1989). Detection of ada static deadlocks using petri net invariants. *IEEE Transactions on Software engineering*, 15(3):314–326.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

[Nesnas et al., 2006] Nesnas, I. A., Simmons, R., Gaines, D., Kunz, C., Diaz-Calderon, A., Estlin, T., Madison, R., Guineau, J., McHenry, M., Shu, I.-H., and Apfelbaum, D. (2006). Claraty: Challenges and steps toward reusable robotic software. *International Journal of Advanced Robotic Systems*.

[OMG, 2016] OMG (2016). Object Management Group.

[openJaus, 2010] openJaus (2010). Openjaus. http://www.openjaus.com/.

[Ouaknine, 2007] Ouaknine, J. (2007). Patricia bouyer, nicolas markey, joël ouaknine, and james worrell.

[Ouaknine and Worrell, 2004] Ouaknine, J. and Worrell, J. (2004). On the language inclusion problem for timed automata: Closing a decidability gap. In *Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on*, pages 54–63. IEEE.

[Ouaknine and Worrell, 2005] Ouaknine, J. and Worrell, J. (2005). On the decidability of metric temporal logic. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS'05)*, pages 188–197. IEEE.

[Panichella et al., 2017] Panichella, A., Kifetew, F., and Tonella, P. (2017). Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*.

[Papazoglou et al., 2007] Papazoglou, M. P., Traverso, P., Dustdar, S., and Leymann, F. (2007). Service-oriented computing: State of the art and research challenges. *Computer*, 40(11).

[Park et al., 2003] Park, J.-M., Bae, E.-H., Pyeon, H.-J., and Chae, K. (2003). A ticket-based aaa security mechanism in mobile ip network. In *International Conference on Computational Science and Its Applications*, pages 210–219. Springer.

[Parker, 2008] Parker, L. E. (2008). Distributed intelligence: Overview of the field and its application in multi-robot systems. *Journal of Physical Agents*, 2(1):5–14.

[Pereira et al., 2011] Pereira, A., Costa, N., and Serôdio, C. (2011). Peer-to-peer Jini for truly service-oriented WSNs. *International Journal of Distributed Sensor Networks*, 2011.

[Petersson et al., 2001] Petersson, L., Austin, D., and Christenseni, H. (2001). Dca: a distributed control architecture for robotics. In *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, volume 4, pages 2361–2368. IEEE.

[Petri, 1962] Petri, C. A. (1962). Kommunikation mit automaten.

[Petters et al., 2007] Petters, S., Thomas, D., and von Stryk, O. (2007). Roboframe - a modular software framework for lightweight autonomous robots. In *Proc. Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware of the 2007 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, San Diego, CA, USA.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

[Pike, 2012] Pike, R. (2012). Go at google. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 5–6, New York, NY, USA. ACM.

[Pinci and Shapiro, 1990] Pinci, V. O. and Shapiro, R. M. (1990). An integrated software development methodology based on hierarchical colored petri nets. In *International Conference on Application and Theory of Petri Nets*, pages 227–252. Springer.

[Pirjanian et al., 2000] Pirjanian, P., Huntsberger, T. L., Trebi-Ollennu, A., Aghazarian, H., Das, H., Joshi, S. S., and Schenker, P. S. (2000). Campout: a control architecture for multirobot planetary outposts. In *Intelligent Systems and Smart Manufacturing*, pages 221–230. International Society for Optics and Photonics.

[Pnueli, 1977] Pnueli, A. (1977). The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE.

[Ponge, 2008] Ponge, J. (2008). *Model based analysis of Time-aware Web service interactions*. PhD thesis, Université Blaise Pascal-Clermont-Ferrand II.

[Puterman, 2014] Puterman, M. L. (2014). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

[Pyro, 2012] Pyro (2012). Website. http://pyrorobotics.com/?page=PyroModuleIntroduction/.

[Qilin and Mintian, 2010] Qilin, L. and Mintian, Z. (2010). The state of the art in middleware. In *Information Technology and Applications (IFITA), 2010 International Forum on*, volume 1, pages 83–85. IEEE.

[Raicu et al., 2008] Raicu, I., Foster, I. T., and Zhao, Y. (2008). Many-task computing for grids and supercomputers. In *Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on*, pages 1–11. IEEE.

[Ramaswamy et al., 2014] Ramaswamy, A., Monsuez, B., and Tapus, A. (2014). Model-driven software development approaches in robotics research. In *Proceedings of the 6th International Workshop on Modeling in Software Engineering*, pages 43–48. ACM.

[Ravn et al., 2011] Ravn, A. P., Srba, J., and Vighio, S. (2011). Modelling and verification of web services business activity protocol. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 357–371. Springer.

[Reed and Roscoe, 1988] Reed, G. M. and Roscoe, A. W. (1988). A timed model for communicating sequential processes. *Theoretical Computer Science*, 58(1):249–261.

[Reisig, 1986] Reisig, W. (1986). Petri nets in software engineering. In *Advanced Course on Petri Nets*, pages 62–96. Springer.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

[Ringert et al., 2015] Ringert, J. O., Rumpe, B., and Wortmann, A. (2015). Transforming platform-independent to platform-specific component and connector software architecture models. *arXiv preprint arXiv:1511.05365*.

[Ritchie, 1997] Ritchie, D. M. (1997). The limbo programming language. *Inferno Programmer's Manual*, 2.

[Robelin and Madanat, 2007] Robelin, C.-A. and Madanat, S. M. (2007). History-dependent bridge deck maintenance and replacement optimization with markov decision processes. *Journal of Infrastructure Systems*, 13(3):195–201.

[Rodgers et al., 2017] Rodgers, R. S., Eatherton, W. N., Beesley, M. J., Dyckerhoff, S. A., Lacroute, P. G., Swierk, E. R., Geraghty, N. V., Holleman, K. E., Giuli, T. J., Rajagopal, S., et al. (2017). Method and system for key management. US Patent 9680805B1.

[Romero et al., 2010] Romero, D., Rouvoy, R., Seinturier, L., and Carton, P. (2010). Service discovery in ubiquitous feedback control loops. In *Distributed Applications and Interoperable Systems*, pages 112–125. Springer.

[Rompothong and Senivongse, 2003] Rompothong, P. and Senivongse, T. (2003). A query federation of uddi registries. In *Proceedings of the 1st international symposium on Information and communication technologies*, pages 561–566. Trinity College Dublin.

[ROS, 2014] ROS (2014). Robot operating system. http://www.ros.org/.

[Rust and Rammig, 2004] Rust, C. and Rammig, F. J. (2004). A petri net approach for the design of dynamically modifiable embedded systems. In *Design Methods and Applications for Distributed Embedded Systems*, pages 257–266. Springer.

[Sales et al., 2010] Sales, D. O., Shinzato, P., Pessin, G., Wolf, D. F., and Osorio, F. S. (2010). Vision-based autonomous navigation system using ann and fsm control. In *Robotics Symposium and Intelligent Robotic Meeting (LARS), 2010 Latin American*, pages 85–90. IEEE.

[Schäl, 2002] Schäl, M. (2002). Markov decision processes in finance and dynamic options. In *Handbook of Markov decision processes*, pages 461–487. Springer.

[Schaller, 1997] Schaller, R. R. (1997). Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59.

[Schantz and Schmidt, 2002] Schantz, R. E. and Schmidt, D. C. (2002). Research advances in middleware for distributed systems: State of the art. In *Communication Systems*, pages 1–36. Springer.

[Schlegel et al., 2009a] Schlegel, C., Hassler, T., Lotz, A., and Steck, A. (2009a). Robotic software systems: From code-driven to model-driven designs. In *Advanced Robotics, 2009. ICAR 2009. International Conference on*, pages 1 –8.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

[Schlegel et al., 2009b] Schlegel, C., Haßler, T., Lotz, A., and Steck, A. (2009b). Robotic software systems: From code-driven to model-driven designs. In *Advanced Robotics, 2009. ICAR 2009. International Conference on*, pages 1–8. IEEE.

[Schmidt et al., 2005] Schmidt, M.-T., Hutchison, B., Lambros, P., and Phippen, R. (2005). The enterprise service bus: making service-oriented architecture real. *IBM Systems Journal*, 44(4):781–797.

[Selic, 2003] Selic, B. (2003). The pragmatics of model-driven development. *IEEE software*, 20(5):19.

[Shannon, 1957] Shannon, C. E. (1957). A universal turing machine with two internal states. *Automata studies*, 34:157–165.

[Shelby et al., 2014] Shelby, Z., Hartke, K., and Bormann, C. (2014). The constrained application protocol (coap).

[Shepard et al., 2012] Shepard, D. P., Bhatti, J. A., Humphreys, T. E., and Fansler, A. A. (2012). Evaluation of smart grid and civilian uav vulnerability to gps spoofing attacks. In *Proceedings of the ION GNSS Meeting*, volume 3, pages 3591–3605.

[Skilligent, 2010] Skilligent (2010). Skilligent. Online: http://www.skilligent.com/index.shtml.

[Smart, 2007] Smart, W. D. (2007). Is a common middleware for robotics possible? In *Proceedings of the IROS 2007 workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware*. Citeseer.

[Soetens, 2010] Soetens, P. (2010). RTT: Real-Time Toolkit. http://www.orocos.org/rtt.

[Sorea, 2001] Sorea, M. (2001). Tempo: A model checker for event-recording automata. In *In Proceedings of RT-Tools' 01*. Citeseer.

[Spaan, 2012] Spaan, M. T. (2012). Partially observable markov decision processes. In *Reinforcement Learning*, pages 387–414. Springer.

[Ståhl and Bosch, 2014] Ståhl, D. and Bosch, J. (2014). Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48–59.

[Sun et al., 1998] Sun, C., Jia, X., Zhang, Y., Yang, Y., and Chen, D. (1998). Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1):63–108.

[Talal and Rachid, 2013] Talal, B. K. and Rachid, M. (2013). Service discovery–a survey and comparison. *arXiv preprint arXiv:1308.2912*.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

[Tenorth et al., 2012] Tenorth, M., Perzylo, A. C., Lafrenz, R., and Beetz, M. (2012). The roboearth language: Representing and exchanging knowledge about actions, objects, and environments. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 1284–1289. IEEE.

[Thangavel et al., 2014] Thangavel, D., Ma, X., Valera, A., Tan, H.-X., and Tan, C. K.-Y. (2014). Performance evaluation of mqtt and coap via a common middleware. In *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on*, pages 1–6. IEEE.

[Theocharous et al., 2001] Theocharous, G., Rohanimanesh, K., and Maharlevan, S. (2001). Learning hierarchical observable markov decision process models for robot navigation. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 1, pages 511–516. IEEE.

[Thomson et al., 2007] Thomson, S., Narten, T., and Jinmei, T. (2007). Ipv6 stateless address autoconfiguration. IETF RFC 4862.

[Torkestani, 2013] Torkestani, J. A. (2013). A highly reliable and parallelizable data distribution scheme for data grids. *Future Generation Computer Systems*, 29(2):509 – 519. Special section: Recent advances in e-Science.

[Tripakis, 2003] Tripakis, S. (2003). Folk theorems on the determinization and minimization of timed automata. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 182–188. Springer.

[Tyagi et al., 2017] Tyagi, N., Gilad, Y., Leung, D., Zaharia, M., and Zeldovich, N. (2017). Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 423–440. ACM.

[Valle et al., 2013] Valle, D., Nuno, E., Basañez, L., and Arana-Daniel, N. (2013). Consensus of networks of nonidentical robots with flexible joints, variable time-delays and immeasurable velocities. In *IROS*, pages 5878–5883.

[Varró et al., 2002] Varró, D., Varró, G., and Pataricza, A. (2002). Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227.

[Ververidis and Polyzos, 2008] Ververidis, C. N. and Polyzos, G. C. (2008). Service discovery for mobile ad hoc networks: a survey of issues and techniques. *IEEE Communications Surveys & Tutorials*, 10(3).

[Waldo, 1998] Waldo, J. (1998). Remote procedure calls and java remote method invocation. *Concurrency, IEEE*, 6(3):5–7.

[Weiss et al., 2009] Weiss, S., Urso, P., and Molli, P. (2009). Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 404–412. IEEE.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

[Whitcomb and Koditschek, 1990] Whitcomb, L. L. and Koditschek, D. E. (1990). Robot control in a message passing environment: Theoretical questions and preliminary experiments. In *Robotics and Automation, 1990. Proceedings., 1990 IEEE International Conference on*, pages 1198–1123. IEEE.

[Zeng et al., 2004] Zeng, J.-z., Xu, J., Wu, Y., Li, Y., and Xu, N. (2004). Service unit based network architecture and its micro-communication element system. *ACTA ELECTRONICA SINICA.*, 32(5):745–749.

[Zhang and Gao, 2012] Zhang, X. and Gao, H. (2012). Road maintenance optimization through a discrete-time semi-markov decision process. *Reliability Engineering & System Safety*, 103:110–119.

[Zhang, 2012] Zhang, Z. (2012). The internet remote robot with skype webcam. In *System Science and Engineering (ICSSE), 2012 International Conference on*, pages 117–119. IEEE.

[Ziparo et al., 2011] Ziparo, V. A., Iocchi, L., Lima, P. U., Nardi, D., and Palamara, P. F. (2011). Petri net plans. *Autonomous Agents and Multi-Agent Systems*, 23(3):344–383.

[Zuberek, 2001] Zuberek, W. M. (2001). Timed petri nets in modeling and analysis of cluster tools. *IEEE Transactions on Robotics and Automation*, 17(5):562–575.

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Glossary

C C is a general-purpose, imperative computer programming language, supporting structured programming, lexical variable scope and recursion, while a static type system prevents many unintended operations. 23, 30, 151, 152, 154

C++ C++ is a general-purpose programming language. It has imperative, object-oriented and generic programming features, while also providing facilities for low-level memory manipulation.. 22, 29, 30, 151–154

C# C# is a multi-paradigm programming language including strong typing, imperative, declarative, functional, generic, *object*-oriented and component-oriented programming paradigms from Microsoft. 29, 30, 152, 154

cloud "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction " [Mell and Grance, 2011]. 16, 17, 20, 30, 31

CoAP Constrained Application Protocol (CoAP) is a software protocol intended to be used in very simple electronics devices, allowing them to communicate interactively over the Internet. . 69, 70, 73–75, 83

Go Go (often referred to as golang) is an open source programming language from Google.. 152

Java Java is a general-purpose computer programming language that is concurrent, class-based, *object*-oriented and specifically designed to have as few implementation dependencies as possible.. 29, 30, 151, 152, 154

Lisp Lisp is a family of computer programming languages with a long history and a distinctive, fully parenthesized Polish prefix notation. 29, 151, 152

object In the class-based *OOP* paradigm, "object" refers to a particular instance of a class where the object can be a combination of variables, functions, and data structures.. 10, 14, 22, 35, 153, 209

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Python Python is a widely used high-level, general-purpose, interpreted, dynamic programming language. 29, 30, 151, 152, 154, 155

TCL TCL is a scripting language.. 151

UNIX Unix is a family of multitasking, multiuser computer operating systems that derive from the original AT&T Unix, development starting in the 1970s at the Bell Labs research center by Ken Thompson, Dennis Ritchie, and others. 86

Windows Microsoft Windows, or simply Windows, is a meta-family of graphical operating systems developed, marketed, and sold by Microsoft. It consists of several families of operating systems, each of which cater to a certain sector of the computing industry with the OS typically associated with IBM PC compatible architecture. 86

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Acronyms

ACE Adaptive Communication Environment. 22, 154

ACID Atomicity, Consistency, Isolation, and Durability. 12

AI Artificial intelligence. 93

API Application programming interface. 11, 17, 18, 22, 24, 25, 28–30, 50, 67–70, 77, 79, 83, 152, 154

Carmen Carnegie Mellon Robot Navigation Toolkit. 20, 23, 25–30, 154, 155

CCR Concurrency and Coordination Runtime. 22, 26, 153

CDIS Code and data integration services. 25, 28–30

CMDP Constrained Markov decision processes. 39

COM Communication. 24, 26–28

CORBA Common Object Request Broker Architecture. 22, 25, 26, 28–30, 153, 154

CPC Component Port Connector. 158

CPU Central Processing Unit. 16, 23, 25, 70, 77, 79, 87, 88, 94, 123, 130, 133

CSP Communicating Sequential Processes. 40, 41, 93

CSS3 Cascading Style Sheets 3. 109

CTL Computational tree logic. 49

DDSS Durable data storage services. 24, 26–28

DHCP Dynamic Host Configuration Protocol. 57

DHT Distributed Hash Tables. 56

DLC Deployment and life-cycle. 24, 28, 29

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

DOM Distributed Object Middleware. 9, 10, 14

DOS Deny of service. 89

DSL Domain Specific Language. 37, 157

DSS Decentralized Software Service. 22, 26, 153

DSSP Decentralized Software Services Protocol. 28

EPI Extension points and interfaces. 25, 28–30

ERA Event recording automata. 95–102, 104–107, 110, 112–115, 117–125, 159, 160, 163

ESB Enterprise Service Bus. 13, 14

FIFO First in First out. 11

Flat-MTL Flat metric temporal logic. 49

FSM Finite-state machine. 38, 93, 94

GENA General Event Notification Architecture. 57

GPS Global Positioning System. 128

GUI Graphical User Interface. 108, 110–113, 118–122, 144, 148

HTML5 Hypertext Markup Language 5. 109

HTTP Hypertext Transfer Protocol. 28, 57, 67, 129, 130, 133, 152

I/O Input/Output. 154

IDL Interface description language. 28–30, 152, 153

IP Internet Protocol. 26, 56, 57, 98, 101–103, 117, 163, 166

IPC Inter-Process Communication System. 26, 28, 89, 155

JS JavaScript. 109

JSON JavaScript Object Notation. 14, 67, 68, 113

LTL Linear temporal logic. 49

LTS Labeled transition system. 44, 45

MARIE Mobile and Autonomous Robotics Integration Environment. 20, 22, 25–30, 154

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

MAS Multi-Agent Systems. 56

MCS Multi-robot coordination services. 24, 26, 27

MDA Model Driven Architecture. 36

MDD Model driven development. 3, 5, 33–38, 41, 50, 51, 85, 86, 90, 91, 97, 99, 101, 107, 115, 117, 120, 125, 129, 144–146, 149, 157

MDE Model driven engineering. 36

MDP Markov decision process. 39, 40

MDP Mediator Design Pattern. 22, 39, 154

MITL Metric interval temporal logic. 49

MM Management and monitoring. 24, 26, 27

MOM Message-Oriented Middleware. 10, 11, 17

MRDS Microsoft Robotics Developer Studio. 20, 22, 25–31, 86, 87, 145, 153

MTL Metric temporal logic. 49

OMG Object Management Group. 36

OOP Object-oriented programming. 10, 35, 209

OV Overhead. 23, 25, 26

P2P Peer to peer. 56

PM Programming model. 24, 28, 29

POMDP Partially observable Markov decision process. 39

Pyro Python Robotics. 20, 23, 25–30, 155

QoS Quality of service. 11, 12

REST Representational State Transfer. 67–70

RF Robustness to failures. 23, 25, 26

ROS Robot operating system. 3–5, 20–22, 25–31, 50, 51, 69, 86, 87, 90, 96, 97, 102, 103, 108, 110, 111, 115, 117, 118, 125, 129, 138, 139, 142, 145, 146, 149, 152

ROSMDB Robot operating system Model Driven Behavior. vii, 5, 85, 107–109, 112–125, 127–129, 132, 133, 138, 140, 141, 144, 146–150

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

RPC Remote Procedure Call. 9, 10, 152

Safety-MTL Safety metric temporal logic. 49

SDfR Service Discovery for Robots. vii, 5, 56, 58–60, 62, 64–75, 77, 79, 81, 83, 119–122, 130, 144, 146–149

SLAM Simultaneous localization and mapping. 18, 88, 152

SLP Service location protocol. 58

SOA Service Oriented Architecture. 4, 5, 13, 14, 31, 33, 34, 51, 85–87, 89, 90, 97, 107, 108, 125, 129, 144–146, 148, 149

SOAP Simple Object Access Protocol. 15, 28, 57, 86

SOE Service-Oriented Environments. 56

SOM Service Oriented Middleware. 13, 14

SOTS Scheduled operations and tasks services. 24, 26, 27

SSDP Simple Service Discovery Protocol. 56–60, 65, 69

TCP Transmission Control Protocol. 28

TCP/IP Transmission Control Protocol over Internet Protocol. 152

TCTL Time computational tree logic. 49

TOM Transaction-Oriented Middleware. 12

TPTL Timed propositional temporal logic. 49

UDP User Datagram Protocol. 57, 59, 60, 73, 75

UML Unified Modeling Language. 157

UPnP Universal Plug and Play. 57–59, 64, 69, 70, 83

V3CMM 3-View Component Meta-Model for Model-Driven Robotic Software Development. viii, 37, 157

VL Vendor locking. 23, 25, 26

VPL Visual Programming Language. 22, 29, 30, 153

VSE Visual Simulation Environment. 22, 153

WSDL Web Services Description Language. 86

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

XML Extensible Markup Language. 28, 57, 86, 113, 152

YAML YAML Ain't Markup Language. 110, 111

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

FOLIO ADMINISTRATIF

THESE DE L'UNIVERSITE DE LYON OPEREE AU SEIN DE L'INSA LYON

NOM : CHITIC DATE de SOUTENANCE : 15/03/2018

Prénoms : Ștefan-Gabriel

TITRE : **Middleware and programming models for multi-robot systems**

NATURE : Doctorat Numéro d'ordre : 2018LYSEI018

Ecole doctorale : Ecole Doctorale 512 - INFORMATIQUE ET MATHÉMATIQUES

Spécialité : Informatique

RESUME :

Malgré de nombreuses années de travail en robotique, il existe toujours un manque d'architecture logicielle et de middleware stables pour les systèmes multi-robot. Un intergiciel robotique devrait être conçu pour faire abstraction de l'architecture matérielle de bas niveau, faciliter la communication et l'intégration de nouveaux logiciels. Cette thèse se concentre sur le middleware pour systèmes multi-robot et sur la façon dont nous pouvons améliorer les frameworks existantes dans un contexte multi-robot en ajoutant des services de coordination multi-robot, des outils de développement et de déploiement massif. Nous nous attendons à ce que les robots soient de plus en plus utiles car ils peuvent tirer profit des données provenant d'autres périphériques externes dans leur prise de décision au lieu de simplement réagir à leur environnement local (capteurs, robots coopérant dans une flotte, etc.).
Cette thèse évalue d'abord l'un des intergiciels les plus récents pour robot(s) mobile(s), Robot operating system (ROS), suivi par la suite d'un état de l'art sur les middlewares couramment utilisés en robotique. Basé sur les conclusions, nous proposons une contribution originale dans le contexte multi-robots, appelé SDfR (Service discovery for Robots), un mécanisme de découverte des services pour les robots. L'objectif principal est de proposer un mécanisme permettant aux robots de garder une trace des pairs accessibles à l'intérieur d'une flotte tout en utilisant une infrastructure ad-hoc. Un autre objectif est de proposer un protocole de négociation de configuration réseau. A cause de la mobilité des robots, les techniques classiques de configuration de réseau pair à pair ne conviennent pas. SDfR est un protocole hautement dynamique, adaptatif et évolutif adapté du protocole SSDP (Simple Service Discovery Proto- col). Nous conduisons un ensemble d'expériences, en utilisant une flotte de robots Turtlebot, pour mesurer et montrer que le surdébit de SDfR est limité.
La dernière partie de la thèse se concentre sur un modèle de programmation basé sur un automate temporisé. Ce type de programmation a l'avantage d'avoir un modèle qui peut être vérifié et simulé avant de déployer l'application sur de vrais robots. Afin d'enrichir et de faciliter le développement d'applications robotiques, un nouveau modèle de programmation basé sur des automates à états temporisés est proposé, appelé ROSMDB. Il fournit une vérification de modèle lors de la phase de développement et lors de l'exécution. Cette contribution est composée de plusieurs composants : une interface graphique pour créer des modèles basés sur un automate temporisé, un vérificateur de modèle intègré basé sur UPPAAL et un générateur de squelette de code. De plus, un framework spécifique à ROS est proposé pour vérifier l'exactitude de l'exécution du modèle et déclencher des alertes. Enfin, nous avons effectué deux expériences : une avec une flotte de drones Parrot et l'autre avec des Turtlebots afin d'illustre le modèle proposé et sa capacité à vérifier les propriétés.

MOTS-CLÉS :

Middleware dynamique, robots connectés, reconfiguration, résilience, mobilité, langages de programmation, systèmes multi-robots, cloud robotisé, flotte robotique, service de découverte, architecture orientée service.

Laboratoire (s) de recherche :

Equipes Dynamid et Inria Chroma, Laboratoire CITI, INSA Lyon

Directeur de thèse: Prof. Olivier SIMONIN, INSA Lyon
Co-directeur de thèse: Dr. Julien PONGE, INSA Lyon

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés

Président de jury :

Prof. Noury BOURAQADI, IMT Lille Douai

Composition du jury :

Prof. Abderrafiaa KOUKAM, Université de Technologie de Belfort-Montbéliard	Rapporteur
Prof. Philippe LALANDA, Université Joseph Fourier, Saint-Martin-d'Hères	Rapporteur
Prof. Noury BOURAQADI, IMT Lille Douai	Examinateur
Dr. Stéphanie CHOLLET, Grenoble INP - Esisar, Valence	Examinatrice
Prof. Olivier SIMONIN, INSA Lyon	Directeur de thèse
Dr. Julien PONGE, INSA Lyon	Co-directeur de thèse

Cette thèse est accessible à l'adresse : http://theses.insa-lyon.fr/publication/2018LYSEI018/these.pdf
© [S-G. Chitic], [2018], INSA Lyon, tous droits réservés