Universidade do Minho
Escola de Engenharia
Departamento de Informática

André Filipe Faria dos Santos

Safety Verification for ROS Applications

Programa de Doutoramento em Informática
das Universidades do Minho, de Aveiro e do Porto

Universidade do Minho

March 2021

André Filipe Faria dos Santos **Safety Verification for ROS Applications**

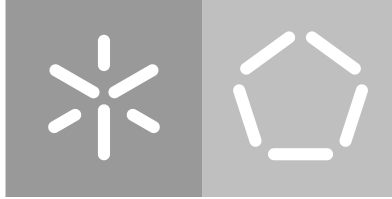
UMinho | 2021

This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project PTDC/CCI-INF/29583/2017 (POCI-01-0145-FEDER-029583) and the PhD grant SFRH/BD/124836/2016.

COMPETE 2020 PORTUGAL 2020

UNIÃO EUROPEIA
Fundos Europeus Estruturais e de Investimento

FCT Fundação para a Ciência e a Tecnologia



Universidade do Minho
Escola de Engenharia
Departamento de Informática

André Filipe Faria dos Santos

Safety Verification for ROS Applications

Tese de Doutoramento
Programa de Doutoramento em Informática
das Universidades do Minho, de Aveiro e do Porto

Trabalho realizado sob a orientação do
Professor Doutor Manuel Alcino Cunha
e do
Professor Doutor Nuno Moreira Macedo

March 2021

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho

Atribuição

CC BY

https://creativecommons.org/licenses/by/4.0/

ACKNOWLEDGEMENTS

First, and foremost, I would like to express my deepest gratitude to my advisors, Professors Alcino Cunha and Nuno Macedo. They maintained a constant presence and guidance throughout the thesis, by reviewing my work, helping me stay on the right track, and pointing out routes and resources of relevance. Their expert knowledge was invaluable to formalise the approach this thesis proposes. Oftentimes our discussions were long and exhausting, but never fruitless. I have faced a fair share of obstacles and dead ends along the way. Without their support, however, I would have faced several more.

I am also grateful to our colleagues from the Centre for Robotics in Industry and Intelligent Systems (CRIIS), from INESC TEC, namely Filipe Neves dos Santos, Luís Santos and Rafael Arrais. They are among the first adopters of our work and have provided us with one of our robot case studies. Seeing as it is still a system under development, without full documentation, I would have been lost in trying to understand it without their assistance. Besides, they have also contributed to the preliminary empirical study that we conducted at the very beginning of this thesis.

I could not go by without extending my sincerest thanks to Mirko Bordignon, Nadia Hammoudeh García and a few other colleagues from Fraunhofer IPA. Our work has been widely promoted among the ROS-Industrial community, and I have participated in various relevant events, such as ROS-Industrial conferences and training sessions, all thanks to them. Furthermore, they are also among the early adopters of our work, and have been applying it to some of the robots built at Fraunhofer IPA, including the well-known Care-o-Bot.

I also had the great pleasure of working with Andrzej Wasowski (IT University of Copenhagen), Gijs van der Hoorn (Delft University of Technology), Harsh Deshpande (Fraunhofer IPA) and Chris Timperley (Carnegie Mellon University), over the major part of this thesis, conducting empirical studies on common bugs that ROS developers face. We have had several enlightening discussions, and their feedback on my progress has been deeply appreciated.

On a personal note, I must thank my parents and my partner Sofia for all the unconditional support they provided over these years. I thank them for a multitude of things, but especially for the small things of everyday life that one tends to take for granted, such as their patience. Many times they have listened to my rambling, even though they likely did not understand any of it.

Lastly, I would like to acknowledge the financial support this work has received. This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project PTDC/CCI-INF/29583/2017 (POCI-01-0145-FEDER-029583) and the PhD grant SFRH/BD/124836/2016.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

RESUMO

Verificação de Propriedades de Correção em ROS

Os robôs são agora parte do nosso quotidiano e a sua utilidade parece não ter limites. Fabricam os nossos bens, colhem alimentos de plantações e conduzem-nos de um lugar para o outro. A inovação na área da robótica é uma constante, as expectativas são altas, e as responsabilidades que depositamos nos robôs são cada vez maiores; os robôs são a nova definição de sistema crítico. Em parte, este sucesso deve-se à abundância de frameworks livres para o desenvolvimento de sistemas robóticos, como é o caso do Robot Operating System (ROS).

A comunidade ROS é numerosa e bastante ativa. O ROS tem sido usado principalmente ao nível da investigação, mas, ultimamente, tem sido a base para vários projetos comerciais ou governamentais. Sendo um ecossistema de componentes de software reutilizáveis, suportado por uma comunidade muito diversa, garantir que sistemas baseados em ROS são confiáveis é da maior importância. No entanto, mostrar que um software robótico é confiável não é, de todo, uma tarefa fácil.

Várias técnicas, como a Verificação Formal, Model Checking, Verificação Runtime, entre outras, têm dado inúmeras provas da sua aptidão e eficácia para verificar uma variedade de propriedades críticas, noutros domínios de software. Contudo, se há algo que todos estes métodos têm em comum, é a sua complexidade. É necessária uma formação especializada para uma aplicação eficaz, e a maioria da comunidade ROS não tem tais conhecimentos.

Nesta tese, respondemos à questão de como adaptar o estado da arte em técnicas de garantia de qualidade para aplicações ROS, e como torná-las acessíveis a não especialistas. Temos em conta o facto de que a maioria dos sistemas são desenvolvidos em torno do código-fonte, em vez de seguirem práticas de Engenharia à base de modelos. Propomos um fluxo de trabalho unificado que, dado o código-fonte de uma aplicação ROS, extrai modelos automaticamente e, de seguida, aplica uma série de técnicas de verificação, estáticas e dinâmicas, relativamente a propriedades do sistema especificadas pelo utilizador. No caso de as análises detetarem uma violação de propriedades, os resultados podem ser utilizados como guias para a resolução de problemas. Caso contrário, os resultados constituem evidência da confiabilidade do sistema, respetivamente às propriedades, que pode ser usada para construção de um argumento de confiabilidade. Este fluxo de trabalho é implementado na plataforma HAROS, e avaliado com dois estudos de caso em robôs reais. O resultado revelou-se eficaz, quer na construção automática de modelos, quer na deteção de falhas, relativamente a propriedades especificadas pelo utilizador. De um modo geral, esta abordagem foi bem recebida pela comunidade ROS. Vários membros já a usam de forma independente, e estão até a propôr as suas próprias extensões.

Palavras-chave: engenharia de software, métodos formais, robótica

ABSTRACT

Safety Verification for ROS Applications

Robots are now part of our daily lives and their usefulness is, seemingly, never-ending. They manufacture our goods, harvest our crops and drive us from place to place. Innovation in the field of robotics is constant, expectations are high, and the responsibilities we place on robots are ever increasing; robots are the new definition of safety-critical devices. In part, this success is due to the abundance of open source frameworks to help develop robotic systems, such as the Robot Operating System (ROS).

ROS has a large and active community. It was mostly used in research, but is recently finding its way into commercial and government projects as well. With such a diverse community, and being based on an ecosystem of reusable software components, ensuring that ROS-based systems are dependable is paramount. However, showing that robotic software is dependable is not an easy task.

Many techniques, such as Formal Verification, Model Checking, Runtime Verification, and more, have proved, over and over, to be capable of verifying a diverse range of properties in other software domains. However, if there is one characteristic that often defines these techniques, it is their steep learning curve. They have a hard requirement on expert knowledge, and the vast majority of the ROS community is composed of non-experts.

In this thesis, we answer the research question of how to adapt state-of-the-art quality assurance techniques to ROS applications, and how to make them usable by non-experts. We take into consideration the fact that most systems are developed with a code-first approach, rather than following Model-driven Engineering practices. We propose a unified workflow that takes in ROS application code, automatically extracts system models from it, and then applies a variety of static and dynamic analysis techniques with respect to user-specified properties. If a property violation is detected, the results of the analyses can be used as a debugging aid. Otherwise, they provide compelling evidence to structure a dependability case for the given properties. This workflow is implemented in the HAROS framework, and later evaluated with two robot case studies. We have found this approach to be effective, both at building models and at finding faults for user-specified properties. Overall, it has been well-received in the ROS community. Several members are now independently using it, as well as proposing their own extensions.

Keywords: lightweight formal methods, software engineering, robotics

CONTENTS

1 Introduction 1
1.1 Context 2
1.1.1 On Building Robot Systems 2
1.1.2 The Robot Operating System 4
1.1.3 A Note on ROS2 5
1.2 Thesis Statement 5
1.3 Running Example: Fictibot 7
1.4 Contributions and Document Structure 8
2 The Robot Operating System 11
2.1 The Basics of ROS 11
2.1.1 Packaging and Distributing Software 11
2.1.2 Deployment and Runtime 14
2.2 Formal Methods and Quality Assurance for ROS 24
2.2.1 Model-based Techniques 24
2.2.2 Static Analysis Techniques 26
2.2.3 Dynamic Analysis Techniques 27
2.3 ROS in Practice 28
2.3.1 Corpus of the Study 29
2.3.2 RQ1 – Which ROS communication primitives are actually used, and how commonly? 31
2.3.3 RQ2 – In which context are these primitives used and how are their arguments defined? 33
2.3.4 RQ3 – What kind of features are typically used in ROS launch files to deploy applications? 35
2.3.5 RQ4 – How and how commonly is the ROS parameter server used? 37
2.3.6 RQ5 – How commonly are custom message and service types used in ROS communications? 37
2.3.7 Impact on Static Analysis 38
2.4 Summary 39
3 Modelling and Reverse-Engineering ROS Systems 40
3.1 State of the Art 40
3.1.1 Architectural Models 41
3.1.2 Model Extraction 42
3.2 A Metamodel for ROS Applications 46
3.2.1 Source Entities 48

3.2.2 Runtime Entities 53
3.3 Model Extraction 55
3.3.1 Overview of the Algorithm 57
3.3.2 File System Artefacts 58
3.3.3 ROS Primitive Calls 60
3.3.4 Configurations and Resources 61
3.3.5 User-provided Hints 63
3.4 Summary 65
4 Behavioural Property Specification 66
4.1 State of the Art 66
4.2 Language Overview 70
4.2.1 Context 70
4.2.2 Concept 72
4.3 Language Syntax 73
4.3.1 Scopes 73
4.3.2 Patterns 74
4.3.3 Events and Predicates 75
4.3.4 Examples 76
4.4 Language Semantics 77
4.4.1 Examples 82
4.5 Summary 86
5 A Framework for ROS-specific Analysis 88
5.1 High-Assurance ROS 88
5.1.1 Architecture and Workflow 89
5.1.2 HAROS Plug-ins 93
5.2 ROS Metamodel and Model Extraction 96
5.2.1 Changes to the HAROS Analyser 96
5.2.2 Changes to the HAROS Visualiser 99
5.3 A Simple Architectural Analysis Plug-in 101
5.3.1 The PyFlwor Query Engine 102
5.3.2 Plug-in Implementation 103
5.3.3 Querying the Computation Graph 105
5.4 Summary 110
6 Online Runtime Verification of ROS Applications 111
6.1 State of the Art 111
6.1.1 Taxonomies for Runtime Verification 112
6.1.2 Runtime Verification Tools and Algorithms 116
6.2 The Need for a New Runtime Algorithm 121
6.2.1 Online vs. Offline Monitoring 121

6.2.2 General MFOTL Monitoring vs. Custom Implementation 122
6.3 Approach Overview 124
6.3.1 Generation and Execution 125
6.3.2 Deployment 127
6.3.3 Architecture and Orchestration 129
6.3.4 Semantics 130
6.4 Monitor Implementation 133
6.4.1 Absence Properties 134
6.4.2 Existence Properties 137
6.4.3 Precedence Properties (Without Variable References) 139
6.4.4 Precedence Properties (General Case) 142
6.4.5 Response Properties (Without Variable References) 145
6.4.6 Response Properties (General Case) 147
6.4.7 Prevention Properties 148
6.5 Summary 151
7 Property-based Testing for ROS Applications 152
7.1 State of the Art 153
7.1.1 Specification-based Testing 153
7.1.2 Property-based Testing 156
7.2 Preliminaries 160
7.2.1 Open Topics and Open Subscriptions 160
7.2.2 Input Traces and Schemas 161
7.2.3 Testable Properties 165
7.2.4 Axioms 166
7.3 Approach Overview 166
7.3.1 Workflow 167
7.3.2 Semantics 169
7.3.3 A Short Introduction to Hypothesis 170
7.4 Pattern-based Trace Schemas 172
7.4.1 Absence Properties 173
7.4.2 Existence Properties 175
7.4.3 Precedence Properties 177
7.4.4 Response Properties 179
7.4.5 Prevention Properties 181
7.4.6 Schema Refinement 182
7.5 Implementation 185
7.5.1 Test Generation Plug-in 185
7.5.2 Input Trace Data Generators 187
7.5.3 Hypothesis Test Driver 191

7.6 Summary 192

8 Case Studies and Evaluation 193

8.1 Case Studies 193

8.1.1 TurtleBot2 193

8.1.2 AgRob V16 195

8.2 Evaluation of the Model Extractor 198

8.2.1 Core Performance Metrics 198

8.2.2 Performance Metrics for Complex Entities 200

8.2.3 Methodology 201

8.2.4 Results for the TurtleBot2 Case Study 204

8.2.5 Results for the AgRob V16 Case Study 207

8.3 Evaluation of the Online Runtime Monitors 209

8.3.1 Prototype MonPoly Implementation 209

8.3.2 Methodology 210

8.3.3 Comparison of Implementations 213

8.4 Evaluation of the Property-based Tests 215

8.4.1 Methodology 215

8.4.2 Results for the TurtleBot2 Case Study 221

8.4.3 Results for the AgRob V16 Case Study 225

8.4.4 Overview and Discussion 231

8.5 Summary 235

9 Conclusion 237

9.1 Summary 237

9.2 Impact and Collaborations 239

9.3 Prospect for Future Work 242

A Fictibot's Source Code 262

a.1 Fictibot Driver Package 262

a.2 Fictibot Controller Package 269

a.3 Fictibot Multiplexer Package 274

a.4 Fictibot Messages Package 279

B ROS Metamodel Schema 281

C Pyflwor Plug-in's Source Code 302

D Catalogue of Properties for TurtleBot2 and AgRob V16 308

d.1 TurtleBot2 Safety Controller Node 308

d.1.1 Axioms 308

d.1.2 Properties 308

d.2 TurtleBot2 Random Walker Controller Node 309

d.2.1 Axioms 309

d.2.2 Properties 309
d.3 TurtleBot2 Multiplexer Node 310
d.3.1 Axioms 310
d.3.2 Properties 310
d.4 TurtleBot2 Random Walker Configuration 310
d.4.1 Axioms 310
d.4.2 Properties 310
d.5 AgRob V16 Safety Controller Node 311
d.5.1 Axioms 311
d.5.2 Properties 312
d.6 AgRob V16 Supervisor Node 312
d.6.1 Axioms 312
d.6.2 Properties 312
d.7 AgRob V16 Joystick Controller Node 313
d.7.1 Axioms 313
d.7.2 Properties 313
d.8 AgRob V16 Basic Configuration 314
d.8.1 Axioms 314
d.8.2 Properties 315
E Property-based Testing Evaluation Results 316
e.1 TurtleBot2 Random Walker Controller Node 316
e.2 TurtleBot2 Multiplexer Node 317
e.3 AgRob V16 Safety Controller Node 319
e.4 AgRob V16 Joystick Controller Node 320

LIST OF FIGURES

Figure 1	The Sense-Plan-Act robot architecture.	2
Figure 2	The Subsumption robot architecture.	3
Figure 3	The Three-Layered robot architecture.	3
Figure 4	The Kobuki robot.	8
Figure 5	The Fictibot robot's architecture.	8
Figure 6	Package organisation within a ROS distribution.	12
Figure 7	Example of a minimal ROS package.	13
Figure 8	Number of Publishers, Subscribers, Service Clients and Service Servers in the corpus.	31
Figure 9	Usage of alternative primitive overloads.	32
Figure 10	Distribution of primitives under control flow.	34
Figure 11	Types of arguments for various ROS primitives.	34
Figure 12	Uses of the `<node>` tag.	36
Figure 13	Overall use of launch file features.	36
Figure 14	Use of custom message and service types.	38
Figure 15	Diagram notation.	47
Figure 16	Class diagram for Source Files.	48
Figure 17	Class diagram for ROS Packages.	49
Figure 18	Class diagram for Repositories.	50
Figure 19	Class diagram for a ROS Project.	50
Figure 20	Class diagram for a ROS Node.	51
Figure 21	Class diagram for ROS Primitive Calls.	52
Figure 22	Class diagram for Source Condition and Source Location.	53
Figure 23	Class diagram for a Node Instance of the ROS Computation Graph.	54
Figure 24	Class diagram for a Topic of the ROS Computation Graph.	55
Figure 25	Class diagram for a Service of the ROS Computation Graph.	55
Figure 26	Class diagram for a Parameter of the ROS Computation Graph.	56
Figure 27	Class diagram for a ROS Link between Resources of the Computation Graph.	56
Figure 28	Class diagram for a ROS Configuration.	57
Figure 29	The `fictibot_controller` package directory.	59
Figure 30	Semantics of LTL operators.	68
Figure 31	Semantics of MFOTL.	70
Figure 32	Common operators of MFOTL, defined in terms of the core set of operators.	71

Figure 33 Structure of a property in the proposed language. 73
Figure 34 Example trace of the Fictibot system. 83
Figure 35 Example trace of the Fictibot system violating an Absence property. 83
Figure 36 Example trace of the Fictibot system violating a Response property. 84
Figure 37 Workflow of the HAROS static analysis framework. 90
Figure 38 Summary page of the HAROS Visualiser. 93
Figure 39 Issue listing with the HAROS Visualiser. 93
Figure 40 Display of a Computation Graph in the HAROS Visualiser. 99
Figure 41 Computation Graph with parameters shown. 100
Figure 42 Resource details and traceability information. 100
Figure 43 Pattern matching for unresolved Resources. 101
Figure 44 Computation Graph with query reports in the HAROS Visualiser. Highlighted is a Link with a queue of size 1. 108
Figure 45 Query reports in the HAROS Visualiser. 109
Figure 46 Workflow of the proposed Runtime Verification HAROS plug-in. 127
Figure 47 Workflow of the proposed Runtime Verification approach. 133
Figure 48 Notation for state machine diagrams. 134
Figure 49 State machine monitoring for the Absence pattern. 135
Figure 50 State machine for the Absence pattern with the `globally` scope. 136
Figure 51 Monitoring example for the Absence pattern. 136
Figure 52 State machine monitoring for the Existence pattern. 137
Figure 53 State machine for the Existence pattern with the `globally` scope. 138
Figure 54 Monitoring examples for the Existence pattern. 139
Figure 55 State machine monitoring for the Precedence pattern (without variable references). 140
Figure 56 State machine for the Precedence pattern with the `globally` scope. 141
Figure 57 Monitoring example for the Precedence pattern (without variable references). 142
Figure 58 State machine monitoring for the Precedence pattern. 143
Figure 59 State machine for the Precedence pattern (general case) with the `until` scope. 144
Figure 60 Monitoring example for the Precedence pattern (with variable references). 145
Figure 61 State machine monitoring for the Response pattern (without variable references). 146
Figure 62 Monitoring example for the Response pattern (without variable references). 147
Figure 63 State machine monitoring for the Response pattern (with variable references). 148
Figure 64 Monitoring example for the Response pattern (with variable references). 149
Figure 65 State machine monitoring for the Prevention pattern (without variable references). 150
Figure 66 State machine monitoring for the Prevention pattern (with variable references). 150
Figure 67 Example ROS Configuration with open subscribers. 161
Figure 68 Runtime perspective of the proposed testing approach. 168
Figure 69 Workflow of the proposed Property-based Testing HAROS plug-in. 168

Figure 70 State machine monitoring for the Absence pattern. 174
Figure 71 State machine monitoring for the Existence pattern. 175
Figure 72 State machine monitoring for the Precedence pattern. 177
Figure 73 State machine monitoring for the Response pattern. 180
Figure 74 State machine monitoring for the Prevention pattern. 181
Figure 75 The TurtleBot2 robot. 194
Figure 76 The Random Walker Controller configuration. 195
Figure 77 The AMCL Navigation configuration. 195
Figure 78 The AgRob V16 agriculture robot monitoring a slope vineyard. 196
Figure 79 The AgRob V16 Basic and Path Planning configurations, the latter in dashed. 197
Figure 80 The evaluation system used in SemEval-2013. 201
Figure 81 The evaluation algorithm we use to assess the model extractor's performance. 203
Figure 82 Example of a minimum weight matching. 204
Figure 83 Architecture of the MonPoly integration with a ROS system. 210
Figure 84 Hypothetical counterexample trace for the observed false negative. 222
Figure 85 Error report for the error found in the AgRob V16 Supervisor. 226
Figure 86 Counterexample trace for the first observed false negative. 228
Figure 87 Counterexample trace structure for the observed false negatives. 230
Figure 88 The `fictibot_drivers` package tree. 262
Figure 89 The `fictibot_controller` package tree. 269
Figure 90 The `fictibot_multiplex` package tree. 274
Figure 91 The `fictibot_msgs` package tree. 279

LIST OF TABLES

Table 1 Corpus of packages used for the empirical study. 31

Table 2 Built-in parsing database for HAROS. 98

Table 3 Comparison of our custom monitor implementation versus MonPoly. 213

LIST OF LISTINGS

2.1 An example package manifest XML file. 13
2.2 An example CMake build file. 14
2.3 Basic lifecycle of a C++ ROS node. 16
2.4 Example of a custom ROS msg with a field and a constant. 17
2.5 A minimal node with a publisher and a subscriber. 18
2.6 A minimal node using a service server. 19
2.7 A minimal node using a service client. 19
2.8 A minimal node reading and writing parameters. 20
2.9 The ROS time primitives. 22
2.10 A minimal launch file for Fictibot. 23
2.11 A minimal launch file for Kobuki. 23
3.1 Specifying subscriber information as launch file comments [170]. 46
3.2 YAML specification of a Source File. 48
3.3 YAML specification of a Package. 49
3.4 YAML specification of a Repository. 50
3.5 YAML specification of a Project. 51
3.6 YAML specification of a Node (excerpt). 52
3.7 YAML specification of a Node Instance (excerpt). 54
3.8 YAML specification of a Configuration (excerpt). 57
3.9 Overview of the model extraction algorithm. 58
3.10 The `fictibot_controller` node, identified in the *CMakeLists.txt* file. 59
3.11 The `main` function of the `src/controller_node.cpp` file. 60
3.12 Overview of the configuration construction algorithm. 61
3.13 The `fictibot_controller` `launch/multiplexer.launch` file. 62
3.14 Some of the topics used in the `src/random_controller.cpp` file of `fictibot_controller`. 62
3.15 Example extraction hints in YAML syntax. 64
5.1 Minimal project file for Fictibot. 90
5.2 Interface through which plug-ins communicate with HAROS. 91
5.3 Excerpt of a JSON data file exported by HAROS. 92
5.4 Plug-in manifest file in YAML syntax. 94
5.5 Excerpt of the HAROS plug-in for the Radon Python tool. 95
5.6 Minimal project file for Fictibot with configurations. 97
5.7 Project file for Fictibot with a configuration and extraction hints. 97

5.8 Project file for Fictibot with plug-in-specific input data. 98
5.9 Basic query to identify topics with multiple publishers. 103
5.10 Basic query to identify topics with multiple publishers. 103
5.11 Plug-in manifest file for the Pyflwor plug-in. 104
5.12 Project file for Fictibot with a configuration and Pyflwor queries. 104
5.13 Entry point function of the Pyflwor plug-in. 105
5.14 Query catalogue to run over Fictibot's Computation Graph. 107
7.1 Trivial example of using Hypothesis. 170
7.2 Using Hypothesis strategies to build ROS messages. 171
7.3 Top-level function to generate test scripts from annotated Configurations. 185
7.4 Function to identify open subscribed topics in a Configuration. 186
7.5 Function to build test scripts using schemas and code templates. 186
7.6 Hypothesis strategy for 8-bit signed integers. 187
7.7 Strategy for `geometry_msgs/Twist` messages. 188
7.8 Strategy for `std_msgs/Int8` messages such that `data` $\leq$ 50. 189
7.9 Example strategy to generate message traces. 190
7.10 Main Hypothesis test function. 191
8.1 Excerpt of a ground truth model, illustrating a node with a publisher. 202
A.1 The *package.xml* file. 262
A.2 The *CMakeLists.txt* file. 263
A.3 The *include/fictibot_drivers/motor_manager.h* file. 264
A.4 The *include/fictibot_drivers/sensor_manager.h* file. 265
A.5 The *src/motor_manager.cpp* file (part 1 of 2). 266
A.6 The *src/motor_manager.cpp* file (part 2 of 2). 267
A.7 The *src/sensor_manager.cpp* file. 268
A.8 The *src/driver_node.cpp* file. 269
A.9 The *package.xml* file. 270
A.10 The *CMakeLists.txt* file. 270
A.11 The *include/fictibot_controller/random_controller.h* file. 271
A.12 The *src/random_controller.cpp* file (part 1 of 2). 272
A.13 The *src/random_controller.cpp* file (part 2 of 2). 273
A.14 The *src/controller_node.cpp* file. 274
A.15 The *launch/minimal.launch* file. 274
A.16 The *launch/multiplexer.launch* file. 274
A.17 The *package.xml* file. 275
A.18 The *CMakeLists.txt* file. 275
A.19 The *include/fictibot_multiplex/trichannel_multiplex.hpp* file. 276
A.20 The *src/trichannel_multiplex.cpp* file (part 1 of 3). 277
A.21 The *src/trichannel_multiplex.cpp* file (part 2 of 3). 278

A.22 The *src/trichannel_multiplex.cpp* file (part 3 of 3). 279
A.23 The *src/multiplex_node.cpp* file. 279
A.24 The *package.xml* file. 280
A.25 The *CMakeLists.txt* file. 280
A.26 The *msg/Custom.msg* file. 280
C.1 The *plugin.yaml* file. 302
C.2 The *plugin.py* file (part 1). 303
C.3 The *plugin.py* file (part 2). 304
C.4 The *pyflwor_monkey_patch.py* file. 305
C.5 Project file for Fictibot with Pyflwor queries (part 1). 306
C.6 Project file for Fictibot with Pyflwor queries (part 2). 307

1

INTRODUCTION

Mankind has always dreamt of machines capable of remarkable feats of intelligence and human-like behaviour. Robots are only a few decades old, but they are the culmination of more than two thousand years of technical evolution [76].

With the advent of the Third Industrial Revolution, in the 1960s, we transitioned from analog, mechanical and electronic technology to the digital technology we know and use. Coincidentally, this transition, that marks the dawn of the Information Age [137, 144], also characterises the recent advances in robotics. The common factor that enables this rapid progress is none other than software. The benefits of software over the traditional technologies is evident, and it did not take long until software became ubiquitous. It proliferated and took control of the emerging technologies, such as digital computers, cell phones, the Internet and, recently, robots.

Robots are now part of our daily lives and their usefulness is, seemingly, never-ending. They manufacture our goods, harvest our crops and drive us from place to place. They are the cornerstone of the Fourth Industrial Revolution, also known as Industry 4.0, and their numbers are ever-increasing. It is estimated that the worldwide operational stock of industrial robots has reached more than 3 million units in the year of 2020 [82], and will reach almost 4 million by 2022.

Robots are now performing feats that could only take place in science fiction, a few decades ago. For instance, the city of Seoul, in South Korea, has planned the construction of the Robot Science Museum starting in the year of 2020, expected to finish in the year of 2022. What is unique about this museum is that its first exhibition will start even before it opens – the façade of this museum will be entirely built by construction robots and drones [30]. Besides being a unique feat in the history of robotics, the intent behind this project is to support public education in robotics and raise awareness of artificial intelligence initiatives.

Robots have also proved their usefulness under exceptional circumstances. In 2020, the COVID-19 global pandemic has completely changed everyday life – remote work and social distancing became new realities, and hygiene is more imperative than it ever was. During these uncertain times, in which public health is a top priority and the global economy has taken a major hit, robots come to the rescue. Hospitals have employed mobile robots to deliver sterile goods and ensure that they do not run out of stock [87]; certain public spaces are sanitised using mobile robots mounted with ultraviolet light attachments [88]; cafes replace human waiters with robotic waiters; and the list goes on. As many jobs and workplaces become extremely conditioned, the economy relies more on robotics to recover.

There is no doubt that robots are capable of incredible things. Innovation is constant, expectations are high, and the responsibilities we place on robots are ever increasing. Robots are the new definition of safety-critical devices. But, what can we say about their overall software quality? In this thesis, we will explore how can some aspects of software quality be improved in robotic systems.

1.1 Context

1.1.1 *On Building Robot Systems*

Primitive robot systems, much like any relatively new technology, were built mostly in an ad hoc fashion. Over time, some software architectures thrived and became standards among practitioners.

One of the first major archetypes, that gained popularity in the first half of the decade of 1980, was the Sense-Plan-Act architecture (SPA) [130]. It was believed in the Artificial Intelligence community that a robotic system should be decomposed into three functional elements, as shown in Figure 1: a sensing system, a planning system and an execution system. The sensing system was responsible for translating raw sensor data into a world model. The planning system took the world model and a mission goal, such as moving towards a given location, and generated a plan to achieve the given goal. The execution system translated the action plan into concrete actions and commands that could be issued to the robot's actuators. This type of architecture is generally known as *deliberative*.

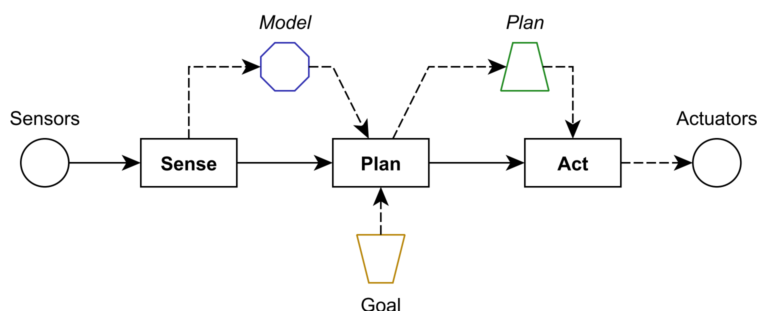


Figure 1: The Sense-Plan-Act robot architecture.

The SPA architecture is an open loop architecture; control flows in one direction, and feedback is never taken into consideration. As a consequence, this architecture does not handle environmental uncertainty and unpredictability very well. The intelligence of this approach, such as it is, resides in the plan generator, which is generally slow and dependent on an accurate model of the world. It turned out that world modelling and plan generation were later classified as Very Hard Problems.

It did not take long for alternative architectures to be proposed. In the mid 1980s, Rodney Brooks proposed the Subsumption architecture [33], the most common departure from the previous paradigm. The Subsumption architecture, as seen in Figure 2, is a *reactive*, behaviour-based architecture; it lives on the opposite end of the spectrum, relative to the SPA architecture.

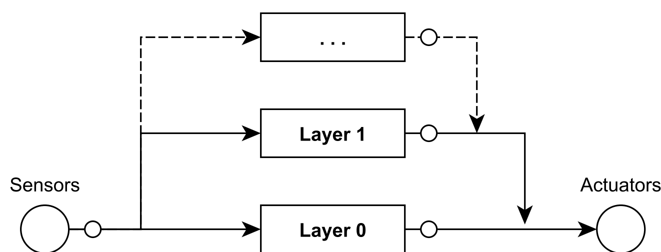


Figure 2: The Subsumption robot architecture.

Subsumption set aside the problems of modelling and planning, to emphasise the coupling of the robot's immediate sensing and actions. The robot's behaviours were decomposed in several layers, representing progressively more complex tasks. Each layer was, essentially, a state machine that handled raw sensor data directly. Layers were composed together, and relied on a process called suppression or inhibition, that ensured that higher-priority behaviours would override less critical ones. Its basis on very simple, stateless and low-level computations had a dramatic effect on performance. Collision-free mobile robots, for instance, were much more performant when built on top of a reactive architecture. On the other hand, it appeared to have a capability ceiling, as it had no means to manage very complex behaviours; it was not sufficiently modular.

In the early 1990s, researchers agreed that none of the two extremes would be the right answer for every problem. Their views converged, and hybrid architectures became the new norm. In particular, the Three-Layered architecture [75] (Figure 3) was one of the most prominent. This architecture is comprised of three elements:

- a reactive Controller, that tightly couples sensors and actuators, and operates with fast, stateless algorithms to implement primitive behaviours;
- a Sequencer, or plan executer, that selects the most appropriate behaviour at any given time, and supplies the necessary parameters;
- a Deliberator, that handles planning, world models, internal state representation, and all time-consuming computations in general.

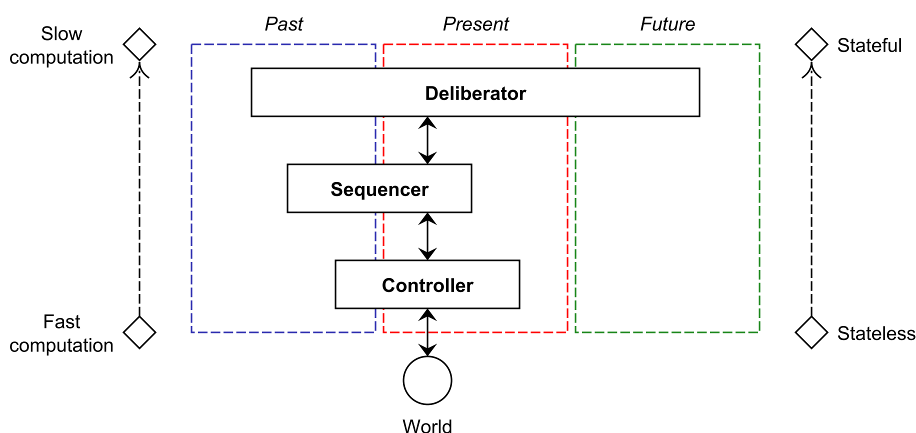


Figure 3: The Three-Layered robot architecture.

As we can see in Figure 3, the Three-Layered architecture organises its components by time and space complexity. The different layers cooperate, rather than inhibit each other.

As robots become more ubiquitous and the hardware becomes cheaper and more capable, the demands we place on robotic systems increase without an end on sight. The size and complexity of robotics software got out of hand for the existing architectures. New concepts and paradigms from other software domains were adopted, such as publisher-subscriber communications in distributed systems, and, eventually, standardised in a number of middlewares. Among the most common middlewares are YARP[1], Orocos[2], Player[3] and the Robot Operating System[4]. The last is now a de facto standard [45, 61, 73, 115], and is the application domain of this thesis.

1.1.2 *The Robot Operating System*

The Robot Operating System [143] (ROS) started in 2007 as a collaborative effort, led by Willow Garage [5], to provide the robotics community a common, free, open-ended development platform. This initiative, which in its first few iterations was essentially a communication middleware, quickly evolved, adding numerous tools, drivers, software components and libraries to its arsenal. It did not take long for ROS to match and supersede its competitors, e.g., the Player project, in terms of suitability for the task, easier learning curve, support and documentation. Now, nearing three thousand software packages in the official distribution, ROS is the de facto standard in robotics research, paving its way into commercial and government applications, backed by a large and active community.

Indeed, the community is one of the strongest aspects of ROS. Notably, some of the main developers and researchers behind ROS established the Open Source Robotics Foundation[6], a non-profit organization (the first of its kind) dedicated to sustaining and improving open source robotics software, such as ROS itself and the Gazebo[7] robot simulator. Other initiatives, such as ROS-Industrial[8], ROS-Military[9] or ROS-Agriculture[10] empower and drive the adoption of ROS in specific commercial domains. More recently, a number of other, smaller groups within the community is also emerging. Among these, the ROS Quality Assurance Working Group[11], a group focused on improving Quality Assurance in the ROS ecosystem, is the most relevant to our work. Since 2012, there is a yearly conference, ROSCon[12], where hundreds of developers of all levels (over 500 in the 2018 edition) have a chance to meet and share projects, ideas and research results centered on ROS. The ROS-Industrial initiative also has its own yearly conference since 2013, although with a smaller audience and a focus on industrial applications and software best practices. Parallel to

1 http://www.yarp.it/index.html
2 https://orocos.org/
3 http://playerstage.sourceforge.net/
4 https://www.ros.org/
5 http://willowgarage.org/
6 http://osrfoundation.org/
7 http://gazebosim.org/
8 https://rosindustrial.org/
9 https://rosmilitary.org/
10 http://rosagriculture.org/
11 https://discourse.ros.org/c/quality
12 https://roscon.ros.org

these, developers worldwide organize regular regional meetups as well. Last, but not least, the community has worked on a few invaluable online resources, such as:

- the ROS Wiki[13], where over 8 000 users share 117 000 pages of documentation and tutorials;
- ROS Answers[14], a website with more than 34 000 users where community members ask and answer ROS-related questions;
- ROS Discourse[15], a forum with more than 4 000 users, where the community discusses projects, ROS-Industrial, Quality Assurance, embedded systems, and more.

1.1.3 *A Note on ROS2*

Traditionally, ROS has followed a release cycle that is similar to Unix distributions, as we will see in Chapter 2. Newer versions introduced changes that, in many cases, did not break existing code. However, the field of robotics is in constant evolution, and its needs change as well. ROS grew way beyond its initial vision, and so, in December 2017, the development team released ROS2, a new major version, that is mostly incompatible with the former. It relies on state-of-the-art technologies, and aims to cover new use cases that ROS did poorly, or not at all. For instance, ROS2 has been designed from the start to be suitable for:

- teams of multiple robots (which is possible in ROS, but there is no standard approach);
- small embedded platforms, such as micro controllers;
- real-time systems, provided operating system support;
- networks with poor connectivity;
- production environments (again, possible with ROS, but not ideal).

The long term plan is for ROS2 to eventually replace ROS. Currently, both versions of ROS exist in parallel, and development for the original version is set to continue, at least, until 2025. Many users (especially companies) are slowly migrating towards the new version, but there are many systems already in place that work only with the original ROS. Since the original version is stable, and was the only released version when this thesis project began, our research will focus solely on ROS, not on ROS2.

1.2 Thesis Statement

It is undeniable that ROS is finding its way into various safety-critical applications. Despite being rooted in relatively harmless research robots, ROS is now the backbone of industrial robots, agriculture robots and more. Safety cages are no more; expensive, powerful robots move freely in unstructured environments, interacting closely with humans, animals, plants and precious goods. We want to be sure that our robots

13 http://wiki.ros.org/
14 https://answers.ros.org
15 https://discourse.ros.org

behave correctly, and, above all, that they are *dependable*. When confronted with the question *"Does my robot do what it is supposed to do, reliably?"*, we want to be able to answer it with some degree of certainty.

It is well known in the software engineering field that this is not an easy question to answer. In some cases, it might even be impossible to obtain a definitive answer. System dependability is often approached from a process perspective, but, ultimately, a system should only be deemed safe, or dependable, if a set of critical requirements, or properties, are considered satisfied. The satisfiability of these requirements can be checked with a number of techniques, such as Formal Verification, Model Checking, Runtime Verification, and more. These techniques have proved, over and over, to be capable of verifying a diverse range of properties in other software domains. But using one technique in isolation might not suffice to cover all desirable properties. Moreover, it is often the case that they are only able to address relatively low-level properties, confined to small units of software and not the system as a whole. Ideally, we want to specify high-level, system-wide dependability properties and have a direct means of showing that they hold. This is where *dependability cases* [89, 129] come in.

A dependability case (also known as *safety case*) is an end-to-end argument that a system satisfies a given critical property, supported by concrete evidence. The argument spans the system both horizontally, considering various inputs and outputs, and vertically, from design level down to the source code. This approach is flexible, in the sense that it can incorporate various methods to provide the necessary evidence. It is, thus, a way to systematically combine the previous techniques and play to the strengths of each, as needed. For instance, Model Checking can be used for sub-properties related to the system's general behaviour, while Runtime Verification, or Testing, can be used to provide evidence for real-time requirements. Furthermore, this approach is also useful in the development process of the system, biasing design decisions that make the case stronger or easier to provide evidence for, thus yielding a safer product, with higher assurance. The introduced bias may also extend to coding standards, generating guidelines that make it easier for other approaches to analyse and provide the necessary evidence.

The major obstacles to structuring a dependability case are the identification of which system components are involved (and how) in a given property, as well as using the analysis techniques themselves to provide evidence. One characteristic that often defines these analysis techniques is their steep learning curve; they have a hard requirement on expert knowledge. When considering a large community, such as the ROS community, coming from vastly different backgrounds (e.g., computer scientists, electrical engineers, hobbyists, etc.), it is expectable that a considerable part of it is unaware of the standard, established practices in software quality assurance [6]. This is concerning, given the fact that ROS thrives on being open source, and promoting the reuse of existing components. If the reusable components are of dubious quality, if they are built on shaky foundations, it is hard to trust the end product. In fact, this problem has already been observed in practice [61]. Some of the factors that hold back the true potential of ROS include:

- the existence of bugs in released components;
- the lack of basic documentation; and
- developers abandoning their projects.

If the standard development processes adopted by the ROS community already included some form of quality assurance, some form of dependability cases (if only for a small set of properties), the released components would be more dependable. Even if the developers end up abandoning the projects, and there are no active maintainers, the issue would be alleviated, to an extent, by the initial boost in dependability. Thus, the question stands:

> How can existing software analysis techniques (such as Model Checking, Runtime Verification, etc.) and tools be used by non-experts, to improve the quality of ROS applications and to provide the basis for dependability cases?

This is the main research question that drives our work. We know that there are many efforts to migrate standard techniques to the domain of ROS applications, as we will see in Chapter 2. But, as we mentioned, these approaches require expert knowledge. They are made by experts for experts. The notable exception is, perhaps, Model-driven Engineering (MDE). MDE can make use of easy-to-learn tools, such as box and line diagrams, that mask some of the underlying complexity, while reaping the benefits of code generation and automatic enforcement of code conventions. Still, no particular approach has thrived to the point of standardisation. Tool support is either scarce or scattered; most developers end up reverting to traditional development processes [73], where models are informal and source code is the core development artefact.

In a world where the general robotics community is struggling to adopt models as the norm, we defend the use of analysis techniques that are based on source code at their core. Such techniques would be able to handle current and past systems, while, at the same time, being compatible with future systems built on models (even MDE approaches end up generating source code). In addition, they are adequate for large ecosystems of open source projects, such as ROS. However, ROS systems are composed of a number of domain-specific concepts (introduced in Chapter 2) that general-purpose tools can hardly capture without significant investment; again, requiring expert knowledge. Over the course of this dissertation, we will show that:

> Standard software analysis techniques can be employed behind an interface that caters to ROS roboticists. Namely, an interface that *(i)* takes source code as input, *(ii)* reverse engineers (formal) models as needed, and that *(iii)* uses a high-level property specification language that addresses ROS concepts directly.

1.3 Running Example: Fictibot

Over the course of this dissertation, we discuss various concepts and techniques that greatly benefit from examples to make them easier to understand. In order to keep examples consistent and relatable, we will use a fictitious robot, called Fictibot, as our running example. Conceptually, Fictibot is really simple. It is a mobile robot platform that moves in two dimensions, using two wheels, and incorporates three basic sensors to perceive its surroundings: a bumper sensor, to detect collisions; a laser sensor, to detect nearby obstacles; and a wheel drop sensor, that detects whether the wheels are in contact with the floor. It is

very similar to the Kobuki robot[16] (Figure 4), an actual mobile platform that is commonly used in the ROS community.

Figure 4: The Kobuki robot.

In terms of its software architecture, Fictibot only has four elements, as shown in Figure 5.

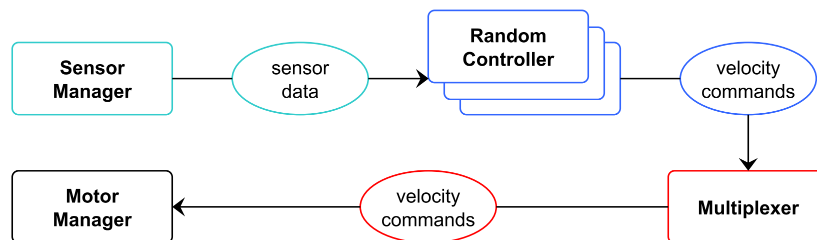


Figure 5: The Fictibot robot's architecture.

Namely, in this architecture, we are able to identify:

- a *Sensor Manager* that reads sensor data and redistributes it to other components;
- a *Random Controller* that consumes sensor data and issues random commands, possibly paired with other controllers, such as a Safety Controller;
- a *Multiplexer* that receives commands from multiple sources and lets through the highest-priority one;
- a *Motor Manager* that receives velocity commands and relays them to the actuators.

The full source code for this example is provided in Appendix A.

1.4 Contributions and Document Structure

This thesis focuses on the application of lightweight Formal Methods, in a code-first paradigm. The idea is to get the most out of the source code, with a low formal specification effort. We want to provide a single workflow where source code is the starting point, and properties can be checked through the use of various techniques, providing the basis for the construction of a depedability case. The construction of the dependability cases themselves, however, is beyond the scope of this thesis. The proposed workflow should alleviate the property specification burden, by allowing both partial specifications and approachable

16 http://kobuki.yujinrobot.com/about2/

specification methods. To this end, we leverage existing techniques that are based on source code, and extend them with ROS-specific domain knowledge. In addition, we explore model extraction techniques, so that we can reverse engineer models out of code. This way we are not completely losing out on the benefits of model-based analyses. At the same time, the use of automatically generated models can help ROS developers familiarise themselves with these concepts, possibly promoting (indirectly) the future adoption of model-based development processes.

Our main contributions can be summarised as follows.

1. A preliminary study on how the various ROS features are used in practice, published in [149]. This study played an important role in helping us decide which features should be prioritised in new analyses.
2. A metamodel for ROS applications, published in [151]. The intent of this metamodel is to raise analyses to a level of abstraction that is more familiar to the ROS developer, as well as to enable the use of model-based techniques.
3. An automatic procedure to build instances of the metamodel from source code via reverse engineering, also published in [151]. The use of automatic model extraction makes our approach compatible with any existing or future project. It also relieves the ROS developer from the burden of modelling, which, in itself, can be an error-prone activity.
4. A minimalistic specification language to annotate models with behavioural properties for which we want to construct dependability cases. It is based on well-known property specification patterns, so that we can capture a large number of use cases, hide the formal semantics behind it, and avoid common specification mistakes and pitfalls. A preliminary version of this language was published in [41].
5. A property-based test framework that, given a system model and a behavioural property, generates a tailor-made test whose goal is to falsify the property and present a minimal counterexample to the user. This constitutes one method of gathering evidence to support a dependability case. A preliminary version of this test framework, without taking behavioural properties into the generation process, was published in [150].
6. The HAROS framework[17] for the analysis of ROS systems. This framework implements the whole approach in a unified workflow, and provides an interface that ROS roboticists can use to improve various aspects of software quality for a given ROS project, without prior knowledge of the underlying analysis methods. The foundation for this framework, without the contributions in the previous items, was implemented prior to this thesis and published in [148].

As for the structure of this document, we start with an introduction to ROS in Chapter 2. This chapter also presents the state of the art in the application of quality assurance practices in a ROS environment, as well as our preliminary study on how the various ROS features are used in practice. Chapter 3 presents our metamodel for ROS applications and the corresponding automatic model extraction procedure. In Chapter 4 we define the specification language for behavioural properties with formal semantics. Chapter 5

17 https://github.com/git-afsantos/haros/

moves towards the technical aspects of this thesis, and serves as an introduction to the HAROS framework. It shows how the previous concepts can be integrated in the framework, and how it can be further extended. Chapter 6 shows a preliminary approach to check behavioural properties of ROS applications, using Runtime Verification. This is not a core contribution for this thesis, but rather serves as a basis for the approach presented in the following chapter. Chapter 7 shows our approach for Property-based Testing of ROS applications – one of the many possible ways to check properties – and its respective integration within HAROS, thus completing the proposed workflow. We evaluate the components of our workflow with two case studies in Chapter 8, one of them being a popular research robot in the ROS community, and the other being a robotic platform for hillside agriculture. Finally, in Chapter 9 we review our thesis and our contributions. We show their impact in the ROS community, so far, and discuss some prospects for future work.

2

THE ROBOT OPERATING SYSTEM

This chapter provides the necessary foundation for the remainder of the thesis. It is our ultimate goal to propose a workflow based on a variety of software analysis techniques that leverages domain-specific knowledge about the Robot Operating System, and is able to provide useful feedback at an early stage. Thus, we start with an introduction to the baseline concepts of ROS in Section 2.1, going over the organisation and deployment of a system, from the source code level to the runtime environment. Then, in Section 2.2 we overview the state of the art in Formal Methods and Quality Assurance for ROS systems (specifically), piecing together a picture of how systems are currently analysed and defects detected. Lastly, in Section 2.3, we present one of the first contributions of this thesis – an empirical study on the usage of the various core concepts and primitives of ROS [149]. The study sets the stage for our other contributions by putting into perspective how (and how commonly) each feature is used in open source systems, which helps us set correct priorities, i.e., what should new analyses and tools focus on.

2.1 The Basics of ROS

2.1.1 *Packaging and Distributing Software*

The Robot Operating System is, first and foremost, a collection of libraries, tools and components for the development of robotic applications. This collection comes in several different versions, called *distributions*, which are updated over time, following a well-defined release and support life cycle[1], similar to, e.g., Linux Ubuntu distributions. The main purpose of a distribution is to ensure that developers are working with a stable codebase for a specific target platform – changes to the core features, within the same distribution, are mostly limited to bug fixes and non-breaking improvements. For instance, ROS Indigo Igloo, released on May 2014, was a *Long Term Support* (LTS) distribution, targeting Linux Ubuntu Saucy (13.10) and Linux Ubuntu Trusty (14.04 LTS). Support for Indigo dropped on April 2019, at the same time that Ubuntu Trusty reached its *End of Life* status. By that time, four more distributions had already been released: ROS Jade Turtle on May 2015, ROS Kinetic Kame (LTS) on May 2016, ROS Lunar Loggerhead on May 2017, and ROS Melodic Morenia (LTS) on May 2018. Choosing the right distribution is the first step one has to take in order to use ROS.

1 http://wiki.ros.org/Distributions

In a ROS environment, software is organised in *packages*, the basic build and release units. Over time, distributions became very large, surpassing 4000 packages in ROS Melodic Morenia. Furthermore, there are many open-source packages that are not part of the official distributions. It does not come as a surprise that most users need only a relatively small number of these packages. To accomodate for this scenario, distributions typically offer installations in four different levels, as seen in Figure 6.

Figure 6: Package organisation within a ROS distribution.

- **ROS Base** is the bare bones installation; it offers the core functionality and command line tools for users to build their own ROS packages and applications.
- **ROS Desktop** offers the Base features, plus a few graphical tools and robot-generic libraries.
- **ROS Desktop-Full** offers ROS Desktop packages, plus navigation and perceptions libraries, as well as 2D and 3D simulators.
- **ROS Distribution** offers a vast range of officially indexed packages that include all of the above and more; cannot be installed directly via convenient commands as the previous levels.

In addition, individual packages, not included in the groups above, can always be hand-picked and installed at any time.

A package has only three hard requirements, as follows. Figure 7 shows the structure of a minimal package.

1. A package must have its own directory; nested packages or multiple packages in a single directory are not allowed.
2. A package must contain a manifest file – an XML file, called *package.xml*, with meta information about the package.
3. A package must contain a CMake build file (*CMakeLists.txt*), using the appropriate macros for the ROS build system.

Besides the CMake and package XML files, a package can contain nearly anything, from source code to robot models, from configuration files to standalone tools.

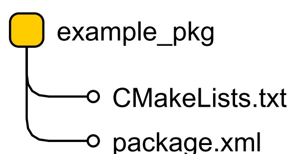


Figure 7: Example of a minimal ROS package.

The package manifest defines basic properties about a package, such as the package's name, version number, authors, maintainers, licenses, and dependencies on other ROS packages. Listing 2.1 illustrates the *package.xml* file of a controller component, for our Fictibot robot, the running example we use throughout the thesis. In this file, we can see the `<name>` tag defining the `fictibot_controller` name for the package, the `<version>` tag defining the version number 0.1.0 and the `<description>` tag providing a free text description. We can also see author and maintainer information in the `<author>` and `<maintainer>` tags, respectively, along with an MIT license in the `<license>` tag. Finally, we define some dependencies, such as `catkin` – the standard package building tool for ROS – in the `<buildtool_depend>` tag, as well as the `roscpp`, `std_msgs` and `fictibot_msgs` ROS packages, in the `<depend>` tag.

```
1  <?xml version="1.0"?>
2  <package format="2">
3    <name>fictibot_controller</name>
4    <version>0.1.0</version>
5    <description>The fictibot_controller package.</description>
6    <author email="andre.f.santos@inesctec.pt">André Santos</author>
7    <maintainer email="andre.f.santos@inesctec.pt">André Santos</maintainer>
8    <license>MIT</license>
9    <buildtool_depend>catkin</buildtool_depend>
10   <depend>roscpp</depend>
11   <depend>std_msgs</depend>
12   <depend>fictibot_msgs</depend>
13 </package>
```

Listing 2.1: An example package manifest XML file.

The `catkin` build system uses CMake, thus the need for a *CMakeLists.txt* file. There are a few requirements regarding the contents of this file, such as declaring dependencies on other packages, which end up duplicating information between *CMakeLists.txt* and *package.xml*. We will not delve into detail for all available functions and macros, as such information is not relevant for this work. Listing 2.2 shows an example CMake file for the same Fictibot controller package. In this example, we can see the name of the package, declared in `project`, package dependencies after `CATKIN_DEPENDS`, directories where C++ header files can be located, in `include_directories`, as well as which executables are built from this package, in `add_executable`, `add_dependencies` and `target_link_libraries`.

Sometimes, packages are closely related, and meant to be used in conjunction. For instance, in Fictibot, it would make sense to install both a package for the robot drivers as well as for a high-level controller, since the driver, by itself, does not do much. This is a common pattern in many robots, such as TurtleBot2[2] or

2 https://www.turtlebot.com/turtlebot2/

```
cmake_minimum_required(VERSION 2.8.3)
project(fictibot_controller)
find_package(catkin REQUIRED COMPONENTS roscpp std_msgs fictibot_msgs)
catkin_package(
  INCLUDE_DIRS include
  CATKIN_DEPENDS roscpp std_msgs fictibot_msgs
)
include_directories(include ${catkin_INCLUDE_DIRS})
add_executable(fictibot_controller
    src/controller_node.cpp src/random_controller.cpp)
add_dependencies(fictibot_controller
    ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
target_link_libraries(fictibot_controller ${catkin_LIBRARIES})
```

Listing 2.2: An example CMake build file.

the Care-O-bot 4[3]. Another example is for libraries that provide similar functionality, such as navigation algorithms. In these cases, it is possible to group packages together with a *metapackage*. A metapackage is simply a package with no contents (besides the mandatory files) that declares runtime dependencies on the packages that it is meant to group together. Thus, when a metapackage is installed, all of its dependencies should also be installed. It is a clever mechanism to circumvent the restrictions on nested packages.

2.1.2 *Deployment and Runtime*

The ROS Computation Graph

A typical ROS system is a distributed system, with various independent resources connecting to each other through various means. This network of resources is called the *ROS Computation Graph*.

Every resource in the graph is named, with a hierarchical naming structure. For instance, both the names `/max_vel` and `/robot/max_vel` represent resources named `max_vel`, but the former is said to be under the *global namespace*, while the latter is under a `/robot` *namespace*. In this case, the first example might represent a maximum velocity parameter for all components in the system, while the latter could represent the maximum velocity of an individual robot in the network.

The intent of using namespaces is to provide encapsulation. In general, a resource can create other resources within its namespace, and it can access resources within or above its namespace. I.e., `/robot` can access `/max_vel` and `/robot/max_vel`, but it can only create `/robot/max_vel`. In practice, this is just a convention, and any resource can access any other resource using its full name.

A full ROS name is a series of identifiers separated (and preceded) by forward slashes, as in the previous examples. However, ROS provides a name resolving mechanism to avoid specifying full names all the time. As such, a name is said to be:

1. a *base* name if it is a simple identifier, e.g., `max_vel`;

3 https://www.care-o-bot.de/en/care-o-bot-4.html

2. *relative* if it is a series of identifiers separated by forward slashes, e.g., `robot/max_vel`;
3. *private* if it starts with a tilde, e.g., `~max_vel`;
4. *global* if it is a full name, starting with a forward slash, e.g., `/robot/max_vel`.

From the definitions above, we can see that base names are also relative names. Private names are a shortcut notation to resolve a name relative to the querying resource's own namespace. For instance, if `/robot` wants to access `~max_vel`, ROS will look for `/robot/max_vel`. Relative names are resolved relative to the namespace the querying resource is in. For instance, if `/robot` wants to access `max_vel`, ROS will look for `/max_vel`, and not `/robot/max_vel`, because `/robot` lives in the global namespace, i.e., `/`. Global names are considered to be fully resolved.

As a final note on names, at runtime initialisation, any name can be *remapped* into any other name. This works as an intermediary look up table for each resource. It is a transparent mechanism to redirect names, without changing source code. For example, suppose there are two resources named `/robot1` and `/robot2`. If a remapping from `/max_vel` to `/max_vel1` is defined for `/robot1`, whenever `/robot1` tries to access `/max_vel`, it will be transparently accessing `/max_vel1` instead. `/robot2` works normally, as remappings are not global. Naturally, remappings play a big role when composing software from different packages into a single system, since source code does not have to be changed for resources to be able to access each other.

ROS Graph Resources

Up to this point, resources were presented in abstract terms. Now, we shall go over the four main types of resources in a ROS system: *nodes*, *topics*, *services* and *parameters*. Nodes are the main resources in a Computation Graph. They are processes that consume, process and produce data. They communicate with each other via message-passing. The other types of resources either hold shared data (parameters) or serve as message-passing channels (topics and services).

One of the main design principles of ROS is that nodes should be specific and modular, rather than monolithic components. It is normal for a single robot to have a network of many nodes. For instance, in the Fictibot example there is the `/fictibase` node that runs the robot's drivers, providing access to sensors and actuators, and the `/ficticontrol` node that provides a high-level controller. In a more realistic example, there might be nodes for localisation, navigation or perception, and each sensor and actuator might have its own individual node. Nodes are written using one of the ROS client libraries, such as *roscpp* (for C++) and *rospy* (for Python), and they typically follow a common lifecycle, shown in Listing 2.3, although this is not mandatory.

```
1  #include <ros/ros.h>
2
3  void main_loop() {
4    // Option 1
5    ros::spin();              // loop until shutdown is requested;
6                              // process incoming messages as they arrive
7    // Option 2
8    ros::Rate loop_rate(10);  // aim for a 10 Hz loop
9    while (ros::ok()) {       // repeat until shutdown is requested
10     ; // read sensors, create messages, update maps, etc.
11     ros::spinOnce();        // process all pending received messages
12     loop_rate.sleep();      // sleep the remaining time to meet the 10 Hz rate
13   }
14 }
15
16 int main(int argc, char **argv) {
17   ros::init(argc, argv, "node_name");   // register as a ROS node
18   // read and write parameters
19   // create topics and services
20   main_loop();
21   return 0;
22 }
```

Listing 2.3: Basic lifecycle of a C++ ROS node.

In Listing 2.3 we can see the use of the `ros::init` function, which registers the process as a ROS node in the Computation Graph, using the name given as its third argument. Before any real work is done, there is usually a setup phase, during which the node reads and writes ROS parameters that it might need, as well as creating (or connecting to) a series of message-passing channels (topics or services). Once this ROS interface is all set up, the node enters its main loop, which should run until a user requests the shutdown of the ROS system. There are two main options for this loop, as shown in the example. The first is simply a call to the `ros::spin()` function, which blocks the process indefinitely, simply waiting for new messages. Once a message arrives, it should automatically trigger the respective reaction function, as we will explain. The second option is to put together the loop manually, using `while (ros::ok())` to detect the user's request to shut down. Within this loop, the user is free to perform any actions. The most common actions are a call to `ros::spinOnce()`, which processes any incoming messages that might be pending at the moment, and some sort of waiting mechanism, a variant of `sleep()` (which we also explain later), so that iterations occur at a certain rate, rather than continuously.

There is a node variant, called *nodelet*, that is designed to bolster the performance of tightly coupled nodes, i.e., nodes that are meant to share many (possibly large) messages and are closely related to each other (e.g., image processing nodes). A nodelet runs on a single thread, rather than a process, and multiple nodelets can be grouped under a *nodelet manager* that spawns the main process. The main advantage of nodelets is that ROS is able to transparently share messages between nodelets via shared memory, rather than sending messages over the network. A limitation of nodelets is that, being threads of a process, they must reside on the same machine. Nodelets are able to communicate with regular nodes, using the standard mechanisms.

In every ROS system there is a special node, not implemented by the user, called the *ROS Master*, that provides the naming service for the whole Computation Graph. Nodes communicate with the Master to register themselves with a given ROS name, e.g., `/fictibase`, and to discover other nodes in the network, e.g., `/ficticontrol`. Nodes must also communicate with the Master to create and access resources in the Computation Graph, such as topics, services and parameters. For instance, when a node is interested in accessing a certain topic, the Master shares with it the list of other nodes that are also accessing the same topic name. This is all done transparently to the user, using certain functions of the ROS client libraries, as we will see in the remainder of this section.

As to how nodes actually send messages to each other, by default, there is a choice between the aforementioned topics and services. Both are typed, meaning that when a resource is created, a message type is registered along with the name, and all nodes are expected to send and receive messages of that type only. Type checking is done only at runtime, by comparing MD5 checksums for the message type. The ROS Base set of packages provides a number of standard message types (e.g., primitive data types, laser scans, 3D poses, etc.), but users can also define their own custom messages. Message types are defined in `.msg` files for topic messages and in `.srv` files for services, using a domain-specific language. The package building system provides the tools to automatically generate source code for the different client libraries. Listing 2.4 shows an example of a custom `.msg` file declaring a constant, `THE_NUMBER`, and two message fields, `a_number` and `stamp`. When messages are sent, only the fields are actually serialised and transmitted. Constants are defined for convenience, e.g., to check that `a_number == THE_NUMBER`. Field types can be primitive types, time stamps, durations or other message types, to allow the definition of complex messages via composition. Fields can also be fixed-length arrays or variable-length lists of any of the previous types.

```
1 # Lines starting with `#' are comments.
2 # Constants are defined with the `=' sign.
3 uint8 THE_NUMBER = 42
4 # Fields only declare a type and a name.
5 time stamp
6 uint8 a_number
```

Listing 2.4: Example of a custom ROS msg with a field and a constant.

Topics are the most common message-passing mechanism, as we will see in Section 2.3. They follow an asynchronous publisher-subscriber model, with many to many connections. Publishers can send messages at any time, regardless of the number of active subscribers, and subscribers are notified (via a callback function), once a new message is received. Both publishers and subscribers are backed by a message queue whose size is defined by the user. Messages are processed in order. If a message queue is full and a new message arrives, the oldest message is discarded silently. Before a node is able to publish any messages on a topic it must `advertise` said topic. In practice, this is the step where the node registers itself on the Master as a publisher, and gets back the current list of subscribers. This type of communication is ideal for continuous data streams, such as sensor data. Listing 2.5 is an adaptation of

the publisher-subscriber tutorial from the official ROS Wiki[4], and shows a minimal example of publishing to a topic `out` and subscribing to a topic `in`. The latter receives messages of the standard type `std_msgs/UInt8` (8-bit unsigned integer), while for the former we use the custom message type defined in Listing 2.4, assuming it is called `tutorials/Custom`.

```cpp
#include <ros/ros.h>
#include <std_msgs/UInt8.h>
#include "tutorials/Custom.h"

void callback(const std_msgs::UInt8::ConstPtr& msg) {
  // do something with `msg'...
}

int main(int argc, char **argv) {
  ros::init(argc, argv, "node_name");   // register as a ROS node
  ros::NodeHandle n;

  ros::Publisher pub = n.advertise<tutorials::Custom>("out", 10);
  ros::Subscriber sub = n.subscribe("in", 10, callback);

  ros::Rate loop_rate(10);  // aim for a 10 Hz loop
  while (ros::ok()) {       // repeat until shutdown is requested
    tutorials::Custom msg;
    msg.stamp = ros::Time::now();
    msg.a_number = Custom::THE_NUMBER;
    pub.publish(msg);       // `msg' is sent at this point

    ros::spinOnce();        // process incoming messages with `callback'
    loop_rate.sleep();      // sleep the remaining time to meet the 10 Hz rate
  }
  return 0;
}
```

Listing 2.5: A minimal node with a publisher and a subscriber.

We can see in lines 13 and 14 the creation of a `Publisher` channel and a `Subscriber` channel, respectively. Both use the creation functions with the least possible parameters. For the `Publisher`, in C++, we have to instantiate the template function `advertise` on the spot, by providing the desired message type (in this case, `tutorials::Custom`). The first parameter, given the argument `"out"`, defines the desired ROS name for the topic (which is subject to name remappings), while the second parameter defines the desired message queue size. The call to `subscribe` is similar, except that the message type is given from its third parameter – a pointer to a callback function that processes incoming messages.

Services are the second message passing method provided by the core ROS interface. This method implements synchronous one to one communication, using remote procedure calls. So, in this case, there is a notion of server (the node that provides the service) and client (the node that uses the service). It is considered an error to have multiple servers for a service with the same name. Messages are exchanged in request-response pairs, and clients block while waiting for a response. Ideally, services are used for quick tasks, such as querying the current state of a node. Long tasks should avoid using services because

4 http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29

it might lead to unresponsiveness, as other messages queue up. Similar to publishers, a service server must also advertise the service to the ROS Master before clients can request it. Listings 2.6 and 2.7 are an adaptation of the service tutorial[5], and show minimal nodes advertising and requesting a service that adds two integers, respectively.

```
#include <ros/ros.h>
#include "beginner_tutorials/AddTwoInts.h"

bool add(beginner_tutorials::AddTwoInts::Request  &req,
         beginner_tutorials::AddTwoInts::Response &res) {
  // request `req' contains two integers `a' and `b'
  // response `res' contains a field `sum' to hold the result
  res.sum = req.a + req.b;
  return true;
}

int main(int argc, char **argv) {
  ros::init(argc, argv, "add_two_ints_server"); // register as a ROS node
  ros::NodeHandle n;

  ros::ServiceServer server = n.advertiseService("add_two_ints", add);

  ros::spin(); // loop, processing messages, until shutdown is requested
  return 0;
}
```

Listing 2.6: A minimal node using a service server.

```
#include <ros/ros.h>
#include "beginner_tutorials/AddTwoInts.h"

int main(int argc, char **argv) {
  ros::init(argc, argv, "add_two_ints_client"); // register as a ROS node
  ros::NodeHandle n;

  ros::ServiceClient client =
    n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_ints");

  beginner_tutorials::AddTwoInts srv;
  srv.request.a = 12;
  srv.request.b = 15;

  if (client.call(srv)) {
    ROS_INFO("Sum: %ld", (long int) srv.response.sum);
  } else {
    ROS_ERROR("Failed to call service add_two_ints");
    return 1;
  }
  return 0;
}
```

Listing 2.7: A minimal node using a service client.

5 http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28c%2B%2B%29

There is a third communication mechanism worth mentioning, called *actions*. Actions are not provided by the core ROS interface, but rather by the `actionlib` library, that has been part of the official distributions since ROS Indigo Igloo. They use several topics under the hood, but are similar to services, in the sense that there is a client-server model. The main difference is that actions are intended to fill the gap left by services for long-running tasks on demand. In this case, request and response (called goal and result, respectively) are asynchronous, i.e., the client sends a goal, and does not block while waiting for a result. Rather, the client is notified, via callback functions, of the progress towards the goal, and, later, of the result itself. The main advantage of actions is that, at any time, the client has the option to preempt the computation. Also, at any point, the client is able to block waiting for a result, just as if using services. Good examples of actions are moving a robot to a certain location, or identifying an object from visual feedback (perception). Seeing as they are not a core communication mechanism, but, rather, an abstraction built on top of the other primitives, this thesis will be focusing only on topics and services.

The final type of resource, parameters, is used to share data with (or between) nodes, but without messages, or any kind of explicit communication for that matter. The ROS Master holds a shared key-value store, called the *Parameter Server*, where keys are ROS names and values can be any primitive type, ISO 8601 dates, base64-encoded binary data or lists of such values. Any node (or user, e.g., via a command line) can read and write values on the server. The Parameter Server is not designed for high-performance access; its main purpose is to hold static configuration parameters. For instance, if a node needs to read a file at runtime, instead of hard-coding the filename in the source code, a path can be fetched from a string parameter. Users would set the appropriate parameter before starting the node, resulting in a much more modular solution. Listing 2.8 illustrates this example.

```
1  #include <string>
2  #include <iostream>
3  #include <fstream>
4  #include <ros/ros.h>
5
6  int main(int argc, char** argv) {
7    ros::init(argc, argv, "file_writer");
8    ros::NodeHandle n;
9    std::string file_path;                          // storage for parameter value
10   if (n.getParam("~file_path", file_path)) {     // getParam reads the parameter
11     std::ofstream input_file(file_path);          // open file for writing
12     if (input_file.is_open()) {
13       input_file << "This file was given via ROS parameters.\n";
14       input_file.close();
15       n.setParam("~succeeded", true);             // write true on the parameter
16     }
17   } else {
18     n.setParam("~succeeded", false);              // write false on the parameter
19   }
20   return 0;
21 }
```

Listing 2.8: A minimal node reading and writing parameters.

When reading and writing parameters, namespaces have a special meaning. They are treated as sub-mappings (or dictionaries, or data structures) within the root mapping. Suppose that a user defines the following parameters.

```
1 "/rgb/r": 255
2 "/rgb/g": 140
3 "/rgb/b": 105
```

Whenever a user accesses, e.g., `/rgb/r`, they would get back 255. But if a node tries to read the top-level `/rgb` name, the Parameter Server returns the whole mapping, i.e.,

```
1 {"r": 255, "g": 140, "b": 105}
```

Managing Time

Time and durations are central aspects to the design and implementation of cyber-physical systems. For instance, a common approach to programming ROS nodes is to set the node's main loop at a fixed rate. This ensures that publishers always produce a relatively steady stream of messages, and also has the potential to serve as a *heartbeat* mechanism, to ensure that the node is continuously making progress. Thus, ROS provides a few convenience classes in the client libraries to ease the management of time, durations and rates, namely:

- `ros::Time` to represent a moment, an instant in time, such as the current time;
- `ros::Duration` to represent a period of time, such as a period of 10 seconds;
- `ros::Rate` a convenience to represent a fixed rate, and making a best effort at keeping it by accounting for the time used to do work in a loop.

These time primitives are all based on a *Time Server* – an internal ROS entity that, by default, keeps track of time as any normal clock would. However, this behaviour can be changed. It is useful to change this behaviour in simulation, for instance, either to slow down the passage of time, or to speed it up. For such cases where real time is needed, though, even if running in simulation, ROS provides a *wall* variant of all primitives above, that bypass the behaviour of the time server – i.e., `ros::WallTime`, `ros::WallDuration` and `ros::WallRate`. Listing 2.9 illustrates the basic interface and usage of the time primitives.

```
ros::Time today = ros::Time::now(); // the current time
ros::Duration five_seconds(5.0);    // a duration of 5 seconds

// Time arithmetic
ros::Duration ten_seconds = five_seconds + five_seconds;
ros::Duration a_full_day = ros::Duration(24*60*60);
ros::Time tomorrow = today + a_full_day;

// Blocking for a period of time
five_seconds.sleep();   // sleep for 5 seconds
ros::Rate ten_hz(10);   // 10 hz
while (ros::ok()) {
  ten_hz.sleep();       // keep a rate of 10 iterations per second
}
```

Listing 2.9: The ROS time primitives.

In practice, `ros::Rate` tends to be combined with active publishers (i.e., producers of data). For instance, Listing 2.5 shows an example of a node using a `ros::Rate`, named `loop_rate`, to define a fixed rate of approximately 10 Hertz. Within the loop, the node calls `pub.publish(msg)` to publish a new message, and `ros::spinOnce()` to process any pending messages that its subscriber might have received. Then, `loop_rate.sleep()` is called as the final instruction of the loop, possibly blocking the process for some time, and making the best effort to meet the defined rate. `ros::Time` is used for time arithmetic or, as in this case, to set a time stamp for a new message, possibly to help distinguish stale data from current data. Here we repeat the relevant excerpt of Listing 2.5.

```
  ros::Rate loop_rate(10);  // aim for a 10 Hz loop
  while (ros::ok()) {       // repeat until shutdown is requested
    tutorials::Custom msg;
    msg.stamp = ros::Time::now();
    msg.a_number = Custom::THE_NUMBER;
    pub.publish(msg);       // `msg' is sent at this point

    ros::spinOnce();        // process incoming messages with `callback'
    loop_rate.sleep();      // sleep the remaining time to meet the 10 Hz rate
  }
```

Durations are seldom used with publishers or subscribers. A more practical use is to set a time limit to wait for the result of an Action. In a similar fashion, durations are used to set a maximum wait time until a service becomes available, using `ros::service::waitForService(service_name, timeout)`.

Deploying a System

There are two ways to start nodes in ROS. The first is to start each node individually, using a command line tool called `rosrun`. In practice, though, this is not really feasible, given that a system may comprise several nodes, each requiring setup. The second way, which is the standard, is through the definition of *launch files* and the `roslaunch` tool. These are XML configuration files used to deploy standalone applications or

components for more complex systems. A launch file can start any number of nodes, by defining <node> tags, as well as providing a common place to define a number of settings, such as:

1. including other launch files, via a <include> tag;
2. defining name remappings, via a <remap> tag;
3. settings parameters, via a <param> or <rosparam> tag;
4. defining namespaces, via a <group> tag;
5. making tags conditional, via the `if` and `unless` tag attributes.

Allowing a launch file to include other launch files is a straighforward step to increase modularity and composition. Parameters are set before launching the nodes, so that they are available as soon as the first node starts. The main difference between <param> and <rosparam> is that the former is suited for single parameters, while the latter allows, e.g., reading a list of parameters from YAML files. Remappings apply to all following node tags, if defined at top level, or to the current node, if defined within a node tag. Groups act as local scopes. That is, a <remap> within a <group> affects only nodes defined within that group. Finally, the `if` (resp. `unless`) attribute declares that a tag is only evaluated if the given condition evaluates to true (resp. false). Listing 2.10 shows a minimal launch file to launch the `/fictibase` and `/ficticontrol` nodes.

```
1 <launch>
2   <node name="fictibase" pkg="fictibot_drivers" type="fictibot_driver" />
3   <node name="ficticontrol" pkg="fictibot_controller" type="fictibot_controller" />
4 </launch>
```

Listing 2.10: A minimal launch file for Fictibot.

There is a substitution mechanism for launch files that essentially mimics the use of variables. This is done with the syntax `$(cmd arg)`, where `cmd` is one of a pre-defined list of commands, and `arg` is the respective argument for that command. A common example is `$(find pkg)`, which is automatically replaced with the path to the package `pkg` in the local file system. This makes it easy to reuse content (e.g., nodes and launch files) from other packages in a portable way. Listing 2.11 shows a more complex example of a launch file, adapted from the Kobuki[6] robot.

```
1  <launch>
2    <node pkg="nodelet" type="nodelet"
3        name="mobile_base_nodelet_manager" args="manager"/>
4    <node pkg="nodelet" type="nodelet" name="mobile_base"
5        args="load kobuki_node/KobukiNodelet mobile_base_nodelet_manager">
6      <rosparam file="$(find kobuki_node)/param/base.yaml" command="load"/>
7      <remap from="mobile_base/odom" to="odom"/>
8      <remap from="mobile_base/joint_states" to="joint_states"/>
9    </node>
10 </launch>
```

Listing 2.11: A minimal launch file for Kobuki.

6 http://kobuki.yujinrobot.com/

2.2 Formal Methods and Quality Assurance for ROS

Robots are, without a doubt, safety-critical systems. With ROS spreading to a large variety of application domains, especially industrial and medical robots, quality assurance and formal methods for safety verification have been gaining a lot of traction in the community.

Dependability, and software quality in general, can be improved with a number of different techniques. Some follow a top-down approach, reasoning at the design level, while others follow a bottom-up approach, starting from the implementation level. The former set of techniques is rooted in abstract and high-level reasoning, and is commonly seen in model-based approaches. Such techniques often discover potential faults during design, or generate correct source code automatically (correctness by construction). The latter set of techniques is based on program analysis methods, but the analyses branch out in two variants: static analysis, the verification of desired properties without executing the program; and dynamic analysis, the verification of desired properties during program execution. In safety-critical systems, however, a combination of techniques is the most sensible choice, as it overcomes the limitations of any single technique and improves overall coverage. In this section we provide an overview of the quality assurance and formal methods panorama for ROS, along the three axes described above.

2.2.1 *Model-based Techniques*

System models can be used for various purposes and with varying degrees of formalism and expressiveness. Traditional Model-driven Software Development processes use modelling languages – sometimes Domain-Specific Languages, or DSLs – to design a system, perform consistency checks, and automatically generate source code, in part or in its entirety. These methodologies are a hot research topic in robotics, despite their low adoption, after their success in the automotive and avionics industries, and for good reason. With such a complex problem and solution space, and being the union of skill sets that it is, the field of robotics can greatly benefit from abstracting away the details, to focus more on the functionality. For instance, someone implementing custom mission logic should not be required to know the mathematical details of kinematics or navigation; they just need components that offer such features with a clear interface.

In ROS, it is easy to see a number of small, immediate things that could potentially go wrong when coding systems manually. Topic, service and parameter names are soft identifiers that lack compile-time checks and fail silently at runtime, besides being scattered across multiple files. This is precisely the problem tackled in [103], where the authors propose a modelling DSL to specify the interfaces of ROS nodes and connect them to form systems. An analysis tool is then able to check the consistency of the architecture, by comparing the models with manually written launch files, and adding some static analysis traits to the approach.

A very common approach is to use graphical modelling languages, similar to UML diagrams, to model systems as a set of components, connected to each other via ports. In the case of ROS, components typically correspond to nodes, while ports correspond to topics (and, less often, to services). We can see instances of this approach in [36, 39, 99, 169, 172]. The main advantage of graphical models is that they

alleviate users from the burden of writing the necessary boilerplate code – besides ensuring that it is error free. As distinguishing notes between these approaches:

1. BRIDE's component model [39] follows the model abstraction levels defined by the Object Management Group (OMG[7]), enabling a transition from platform-independent models to platform-specific ones;
2. ROSMOD [99] considers both software components and hardware systems;
3. RAFCON [36] supports the specification of systems as hierarchical state machines;
4. ReApp [169, 172] extends components with an ontological classification, to set up a cloud-based semantic component repository and to simplify the discovery (and reuse) of components.

At the time of writing, only ROSMOD and RAFCON seem to have been updated beyond the initial release period.

A disadvantage of the approaches seen so far, is that the proposed models are not entirely formal. Although they are a step forward in terms of improving the quality of the product, they waste the potential of verifying critical safety properties. For instance, modelling ROS systems with the Robot Architecture Definition Language [102] allows the verification of real-time properties (e.g., maximum message latency) using SMT-based infinite-state bounded model checking. The trade-off is that nodes are assumed to follow a relatively strict life cycle (although common, in practice), and require more specification effort (e.g., worst-case execution time). An alternative approach, much in the same spirit, is to use AADL [63] to model and develop ROS applications [16]. Since AADL models can include hardware components and have well-defined analysable semantics, it is a more natural fit for robotics over, e.g., UML.

Timed automata are another common formalism in robotics. They can be used with Uppaal[8] to verify real-time properties [78] (e.g., avoiding message loss due to overflow) or to synthesise C++ code too [108].

Stepping up in terms of the required expert knowledge, it is possible to generate ROS code that is provably correct by construction [9, 121], using proof assistant tools such as Coq[9]. The first approach [121] focuses on model-based deployment of platform-independent control code on a specific platform (ROS, in this case). Various modelling and code generation tools generate components that need to interact with the remaining components of the ROS system. The necessary ROS wrappers for the platform-independent code are automatically generated from a node modelling DSL, and proved correct with Coq. Alternatively, Coq can be used to build a physical and behaviour model of the system [9], which can be used later to prove properties and to generate correct code. This is a unique approach in that it accounts for the events of the physical domain and uses exact computations with real numbers, instead of the traditional floating point arithmetics. However, despite being well suited for safety-critical systems, it is an approach with a steep learning curve for the average ROS developer.

Lastly, [167, 168] presents a case study with Care-O-bot where they model a high-level planning and scheduling system for human-robot interaction scenarios. It is a rule-based system that is shown to behave correctly in the modelled scenarios, which include reminding a person to take medication at a given time,

7 http://www.omg.org/
8 http://www.uppaal.org/
9 https://coq.inria.fr/

or moving to a specific location in the house, told by the person. For the modelling aspect, the authors formalised requirements in LTL, and verified them with the SPIN model checker [83].

2.2.2 *Static Analysis Techniques*

Static analysis can be broadly defined as any kind of program analysis that does not require execution of said program. It can target either source code or binary code. The former is vastly more common, both in robotics and otherwise. To our knowledge, there is no published research on static binary analysis tailored for ROS.

One of the most common themes in static analysis is the measurement of various metrics or coding standards compliance. For instance, the ROSEco project [52] provides an overall measurement of package impact and health, calculated from an aggregation of package dependencies, number of maintainers and repository metrics (e.g., number of issues, number of watchers). A similar approach is proposed in [139], in which the authors scan a large collection of ROS packages to identify packages that are themselves poorly maintained or depend on poorly maintained packages. In terms of coding style and standards compliance, *roslint*[10] is a package provided in the ROS distribution that enables lint-style checks in C++ and Python code, using *cpplint*[11] and PEP8[12], respectively. Lastly, our own previous work, the plugin-based framework HAROS [148], initially focused on applying a number of general-purpose tools to ROS repositories, in order to gather quality metrics and verify compliance with coding standards.

Overall, the previous approaches offer little in terms of ROS- or robotics-specific insight. In contrast, Phricky Units [133, 134, 135] is a tool that automatically identifies dimensional inconsistencies in ROS code. This tool has a mapping of physical units that are expected for certain fields of ROS messages, and then checks whether these fields, or variables derived from them, are misused in calculations with values of different units (e.g., adding meters to meters squared). Lastly, [142] and [156] use static analysis to partially reconstruct the ROS Computation Graph from C++ code and launch files (only the latter). They further traverse control flow and function call graphs to try to determine dependencies between publishers and subscribers. In [142] the goal is to document the publisher-subscriber architecture as a model, and to find differences between two versions of the same component. In [156] the goal is to analyse the impact of changes to the system, in order to determine which nodes are affected by, e.g., a change to the message publication rate of a specific node. Conservative approaches, that do not take the publication rate into account, could potentially recommend reviewing many (if not all) nodes of a system when a single node is changed during development. With this approach, the authors achieve a more precise set of affected nodes.

10 http://wiki.ros.org/roslint
11 https://github.com/cpplint/cpplint
12 https://www.python.org/dev/peps/pep-0008/

2.2.3 *Dynamic Analysis Techniques*

Dynamic analysis is the complement to static analysis. The analysed target is the actual system implementation, as is the case with static analysis, but the analysis requires execution. It can happen while the system is executing, in which case it is called *online*, or it can happen after the fact (e.g., by analysing log files), in which case it is called *offline*. Generally, dynamic analysis falls into two broad categories: testing and runtime monitoring.

Testing techniques specifically tailored for ROS are not very common, presumably because general purpose techniques can be applied, to some extent, and because ROS offers tools for traditional unit and integration testing[13] [14]. As such, new solutions [26, 60, 93] focus on the common theme of better integrating simulation in tests, as well as supporting model-based tests and other modern testing methods. Notably, [60] and [93] explore the use of timed automata to test navigation and localisation capabilities. The latter also presents a testing toolkit that achieves scalability by enabling multiple tests to run in parallel, possibly in a cloud-based environment, using simulation and Docker[15] containers.

Runtime monitoring for ROS, on the other hand, is a hot research topic [4, 5, 27, 84, 85, 106, 171]. Some approaches focus more on the system monitoring aspects [27, 171] (e.g., system performance and status), while others are used to detect violations of desirable behaviour properties [4, 5, 84, 85, 106], a form of runtime monitoring that is more commonly known as *runtime verification*. For instance, [171] uses hardware observers to monitor CPU and memory usage, diagnostic observers (with the ROS `/diagnostics` topic) to check diagnostics messages, and node observers to determine whether a node is running. Similar system introspection capabilities can be seen in [27], where the solution also monitors message latency and checks the structure of the computation graph against a reference state (e.g., to detect missing nodes).

Runtime verification solutions aim at ensuring that a running system respects desirable safety (and possibly liveness) properties. In comparison with static analysis, it is not held back by, e.g., the halting problem, but it is limited to a single execution trace at a time. The main distinguishing factors between these approaches are as follows.

1. ROSRV [85] does not offer a property specification language; properties are coded in manually, but it can also monitor security policies, besides functional safety.
2. DeRoS [4, 5] has its own architecture and property specification DSL, based on the previous work in [103]; in this case, the generated monitor is capable of monitoring temporal properties and enforcing safety actions, albeit without an explicit semantics given in terms of standard temporal logic.
3. RMoM [84] is capable of monitoring discrete-time Metric Temporal Logic properties over a finite set of predicates for ROS robot swarms; predicates include whether certain nodes and topics are present, the current swarm formation, as well as the position and velocity of robots within the swarm.

13 http://wiki.ros.org/rostest
14 http://wiki.ros.org/rosunit
15 https://www.docker.com/

4. Lesire et al. [106] propose a specification DSL that allows runtime verification of Past-time Linear Temporal Logic properties with real-time constraints; this approach is benchmarked against DeRoS, and found to be more effective and more efficient.

Finally, in [170] the authors tackle a different problem: the reconstruction and consistency checking of the ROS computation graph. This approach is mixed, in that it uses static analysis for launch files – to determine which nodes make up the system – and dynamic analysis to extract the interfaces of the nodes themselves. Nodes are executed within a sandbox environment that also loads a library that intercepts calls to the ROS Master. From these intercepted calls, they are able to determine which topics and services are advertised and used by each node, and thus are able to put together the whole graph. The main disadvantages of this approach are the assumption of a standard node life cycle (i.e., all topics and services are created at setup), and, given that it is based on dynamic analysis, it can only extract information from a single execution trace.

2.3 ROS in Practice

To push the quality of ROS systems forward, advanced analysis tools are undoubtedly required. Static analysis, in particular, seems to be a promising approach, despite being generally hard to implement. It is suitable for all systems and components, new and existing ones, as long as the source code is available. In an open source ecosystem such as ROS, generic components tend to be reused, and application-specific components tend to be custom made; in both cases, source code availability should not be an issue. Besides, the ROS community is large and diverse, and many roboticists are not well-versed in formal methods or standard software engineering practices, such as Model-driven Software Development, despite the number of proposed approaches for either [9, 36, 39, 78, 99, 108, 121, 169, 172]. This leads to a development process with little to no modelling and jumping straight into coding – an approach that favours static analysis.

Building powerful, ROS-specific static analyses is possible [156, 170] but far from trivial. A first challenge is that a ROS system is completely open. I.e., there is no real concept of application, since a group of nodes could operate on their own in one context, or represent just a subsystem in another context. A second challenge, as we can see in Section 2.1, is that ROS allows a high degree of freedom when it comes to system design, making it difficult to reverse engineer the computation graph. In particular, launch files are customisable with, e.g., environment variables or conditional statements. As a third challenge, the ROS client libraries also provide a myriad of *primitives*, used to create topics and services or to read and write parameters, that can be called at any point in the program.

Our first contribution is, thus, an empirical study, published in [149], with the goal of detecting common usage patterns of ROS features and primitives, both in launch files and in the source code. The main outcome of this study is a ranking of the most frequent usage patterns, making it easier to identify where new analysis tools need to invest their effort. To address our first challenge, and for the purposes of this study, we consider a ROS application to be a *top-level* launch file, i.e., a launch file that is not directly

or indirectly included by any other launch file in the corpus. Thus, we conduct the study, driven by the following research questions.

RQ1 *Which ROS communication* **primitives** *are actually used, and how commonly?*

RQ2 *In which* **context** *are these primitives used and how are their* **arguments** *defined?*

RQ3 *What kind of features are typically used in ROS* **launch files** *to deploy applications?*

RQ4 *How and how commonly is the ROS* **parameter server** *used?*

RQ5 *How commonly are* **custom message and service types** *used in ROS communications?*

From a code quality and analysis standpoint, the answers to these questions dictate how *knowledgeable* static analysis (and the respective developers) must be. Knowing which primitives are used more often can help prioritise their support in tools. For static analysis, it is also very relevant to know in which context such primitives appear (namely, whether they are within control flow) and how are they parameterised (literal arguments versus variable arguments). In particular, analyses become complex, if not impossible, when values originate from the ROS parameter server. Parameters are dynamic by definition, but, contrary to regular dynamic variables (which can be altered at various points within the same process), ROS parameters can also change, without notice, by action of other processes in the network. There are only a few scenarios in which one can determine, statically, the value of a ROS parameter, e.g., when it is undefined or when it is defined in a launch file and no nodes redefine the value at any point. Finally, the usage and definition of non-standard message types limits the domain-specific knowledge tools can leverage. For example, the previously mentioned Phricky Units [133, 134, 135] is a tool that automatically identifies dimensional inconsistencies in ROS code. To do so, it relies on a mapping of physical units that are expected for certain fields of standard ROS messages. The introduction of non-standard message types, naturally, limits its usefulness and capability to act.

2.3.1 *Corpus of the Study*

To put together a corpus of ROS packages would seem straightforward, at first glance. Given that ROS is deeply rooted in the world of open source software, a corpus could be as simple (and as vast) as any freely available ROS package. For instance, given that we worked with the ROS Indigo Igloo distribution, our starting point could be to traverse the distribution's index and collect all indexed packages. However, this is an empirical study, backed by automatic mining tools, on the expected practices in standard ROS application development – and here we hit a few issues that have us trim down the corpus significantly.

First, and foremost, *the corpus should be representative of actual ROS systems*. We focus on packages that provide building blocks for robotic applications, i.e., controllers, drivers, planners, etc.. More specifically, we chose packages from robot systems that are indexed in either the ROS Indigo Igloo or the ROS-Industrial repositories (the latter is not a subset of the former), mostly based on popularity within the community and

source code availability. Examples include the TurtleBot2, or the Fraunhofer IPA Care-O-bot[16]. But not all robots are equally interesting to study; some have most of their functionality confined to proprietary, ROS-agnostic drivers, exposing only a small ROS wrapper for integration purposes. Ideal systems, such as the two aforementioned examples, have most of their source built for ROS and with ROS.

Our initial package corpus, based on concrete robotic systems, was then refined. Some packages are just wrappers for ROS-agnostic libraries and utilities, their purpose being to streamline the process of managing dependencies and making said utilities easily available for ROS developers. Such packages should be avoided because they do not contain any ROS nodes or primitives. Including them would: *(i)* skew the results (e.g., percentages relative to all packages); *(ii)* increase the overall time needed to process all source code; and *(iii)* answer none of our research questions.

Moreover, we must *take automation and technical limitations into account*. At the time this study was conducted, our prototype mining tools were limited to parsers for launch files and C++ source code. Consequently, any components written in other languages (with Python being the most common alternative) have to be left out of the analysis. The C++ mining tool relies on the Clang compiler, and most ROS packages default to compiling with GCC. We had to ensure that every package could be properly compiled beforehand, which means that, in some cases, we had to make changes to the source (mostly limited to CMake files) in order to solve incompatibilities. This is an additional layer of manual work that severely hinders our capacity to process a large corpus.

The initial selection yielded 480 packages. After filtering the selection, by removing metapackages and some ROS-agnostic libraries, we settled on a corpus composed of 380 indexed packages. As per our definition of ROS application within this study (a top-level launch file), the corpus contains 365 launchable applications. Table 1 shows how the different packages and applications are distributed among the various robot systems[17]. In this table, the entry *Other Packages* denotes packages that are used by one or more robots, but are not part of any system in particular, such as ROS implementations of certain hardware communication protocols.

Overall, 175 out of 380 packages, 46.05% of the corpus, contain C++ source code, and are, thus, subject to the analysis pertaining to ROS primitives. In total, these represent 3,377 C++ files. The analysis of launch file features is performed on 200 packages (52.63% of the 380) containing 901 files. By comparison, the full ROS Indigo Igloo distribution index contains 2106 packages. After excluding 294 metapackages, we found that 907 (50.06%) contain C++ source code and 757 (41.78%) contain launch files. In relative terms, our corpus seems to be a representative sample of the ecosystem.

A preliminary analysis of our corpus shows that 40.51% of all launch files (365 in 901) fall into our definition of launchable application. Unsurprisingly, we found that the package-application relation is many-to-many, i.e., a single application depends on multiple packages, and a single package can contain multiple applications.

16 https://www.care-o-bot.de/en.html

17 Some inconsistencies and unclear aspects were detected in this data after publishing the study. Here we present the correct values.

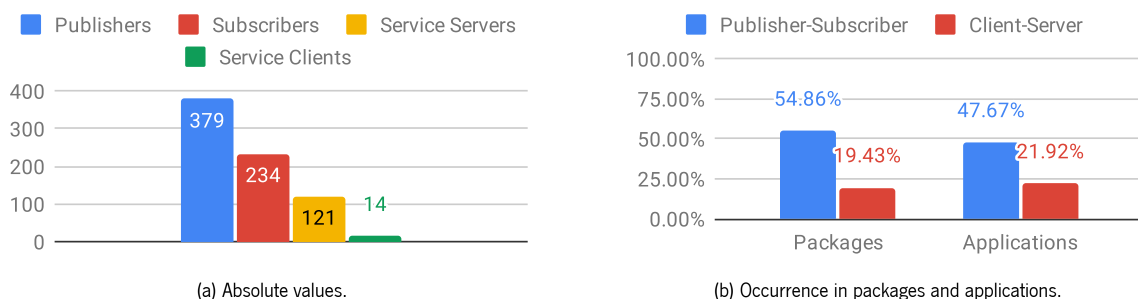
Name	Packages	Applications	C++ LOC
Aubo	11	9	4,893
Clearpath Grizzly	9	14	1,510
Fraunhofer IPA Care-O-bot	68	27	43,765
Kinova MICO	5	5	4,265
Robotiq Adaptive Gripper	14	3	3,294
Robotnik AGVS	8	9	2,113
Robotnik GUARDIAN	10	19	5,555
Robotnik RB-1	13	23	525
Robotnik RBCAR	9	11	933
Robotnik SUMMIT	13	7	2,837
Shadow Dexterous Hand	57	43	33,983
TurtleBot2	71	101	38,739
Universal Robot	9	19	1,418
Yaskawa Motoman	10	29	8,747
Other Packages	73	46	543,776
	380	365	696,353

Table 1: Corpus of packages used for the empirical study.

The collected data set, relative to the analyses over C++ and launch files, is available online[18]. We describe the main results of the study and answer our research questions in the following subsections.

2.3.2 *RQ1 – Which ROS communication primitives are actually used, and how commonly?*

This study is limited to the two default communication primitives offered in ROS, the publisher-subscriber paradigm and the client-server paradigm. Other mechanisms, such as Actions, were left out entirely; they were not even counted towards the publisher-subscriber paradigm which is the foundation for their implementation. We mined the source code, looking for calls to the various ROS C++ API functions that create publishers, subscribers, service clients or service servers. Our initial guess, from reading the ROS documentation, tutorials and other reference material, was that the publisher-subscriber paradigm would be the most common. Indeed, our findings confirm this hypothesis, as shown in Figure 8.

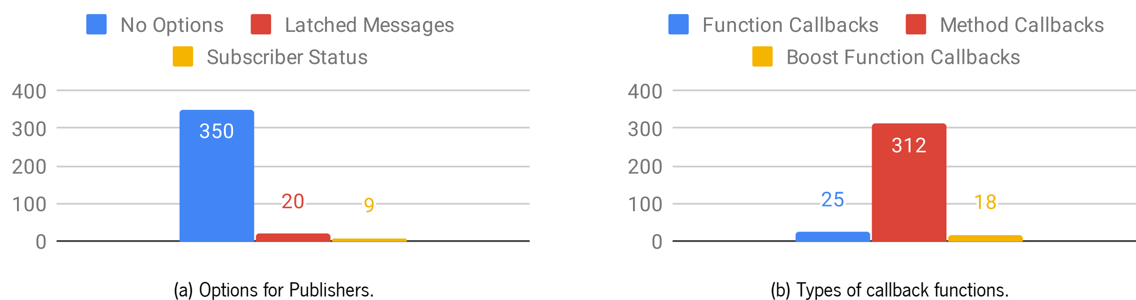


(a) Absolute values.

(b) Occurrence in packages and applications.

Figure 8: Number of Publishers, Subscribers, Service Clients and Service Servers in the corpus.

18 https://github.com/git-afsantos/ros_data



(a) Options for Publishers.

(b) Types of callback functions.

Figure 9: Usage of alternative primitive overloads.

We have registered a total of 379 publishers, 234 subscribers, 121 service servers and 14 service clients, as shown in Figure 8a. Combined, we have 613 uses of the publisher-subscriber paradigm, as opposed to 135 uses of the client-server paradigm. In relative terms, this means that nearly 82% of all ROS communications create publishers and subscribers. Figure 8b shows that, across our corpus, 96 out of the 175 analysed packages (54.86%) create ROS topics and 34 packages (19.43%) create ROS services. The results are similar for launchable applications; 174 applications (47.67%) create topics, but only 80 applications (21.92%) create services.

The prevalence of topics is an expected result, since the publisher-subscriber paradigm allows for a greater degree of flexibility. This model is more lenient in terms of system performance, since message processing happens asynchronously, and components can be built regardless of whether there are counterparts on the other end of the topic. For instance, a node that subscribes to a topic is, in theory, perfectly functional regardless of the number of publishers on that topic. On the other hand, if a node is reliant on a service, it blocks until a server is available.

Looking at the collected figures, we are also able to see that these systems produce more information than they consume. The publisher to subscriber ratio is of 1.62 calls to `advertise` (producers) per call to `subscribe` (consumer). For services, the ratio goes up to 8.64 calls to `advertiseService` per call to `serviceClient`. This discrepancy could be due to robotic systems being typically designed in pyramidal hierarchical approaches. Often, the number of low-level components is larger than the number of high-level components (e.g., drivers and low-level controllers versus trajectory planners). In addition, the former produce large amounts of information (e.g., from various sensors), to be collected and processed by higher-level components, while consuming less types of information (e.g., velocity commands). We have mentioned how most of the corpus is composed of building blocks for more complex (or custom) robotic systems, so this is likely the reason behind these ratios.

We now look into the different primitive overloads provided in the ROS C++ client library in Figure 9.

The chart in Figure 9a shows the usage of optional parameters for `advertise`, the creation of a new ROS publisher. It is evident that the default version, with no optional arguments, is the preferred choice; 92.35% of all instances fall into this category. Only 20 publishers latch messages – the automatic re-publication of the last published message when a new subscriber joins the topic. This is useful for data that does not change often, like static maps, allowing a fire-and-forget type of behaviour. The other option, *Subscriber*

Status, allows the topic advertiser to define custom callback functions for when a topic subscriber joins or leaves the topic. It is a rather uncommon feature; there are only 9 advertisers that make use of it, distributed between two packages from Fraunhofer IPA Care-O-bot for image processing. The two options are not mutually exclusive, but we have found not a single topic advertiser using both at the same time. There are 11 Care-O-bot applications, for different versions of the robot, that use a combination of all types of calls to `advertise`.

The other chart, in Figure 9b, shows the preferences of users regarding the types of callback function registered on `subscribe` (to create a subscriber) and on `advertiseService` (to create a service server). It is clear that the vast majority of callback functions are member functions (methods) of a C++ class, with 312 functions out of 355. Other variants include global functions, with 25 instances scattered across 10 packages and 28 applications, and 18 Boost functions – a function wrapper created with the Boost C++ library, to create closures, i.e., to bind some variables to specific values whenever the wrapped function is called. The likely reason for the dominance of member functions is to keep internal state from one function call to the other, or to easily share state between different program points, or even threads. Global functions are disfavoured in this case, since they would either be stateless, or have to rely on global variables to share state (a discouraged practice, in general). Boost functions are a rather uncommon middle ground alternative, that, most of the time, can be replaced with the simpler member function.

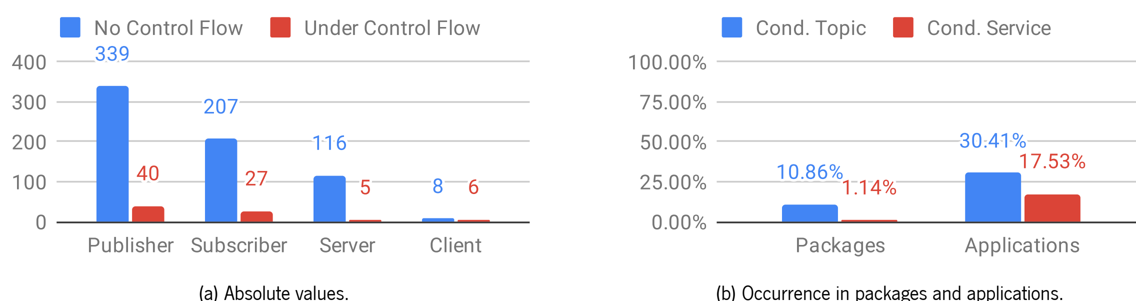
As a last remark, we have registered no occurrences of any ROS primitive using a transport protocol other than TCP, which is the default.

2.3.3 *RQ2 – In which context are these primitives used and how are their arguments defined?*

This research question is more closely tied to the required complexity of static analysis tools. In terms of context, we want to assess whether primitives are always called from the main control flow path, or whether there might be primitives called conditionally, or even within loops. Analyses step up in complexity when there are dynamic conditions involved, as the structure of the ROS interface is no longer static. A similar reasoning applies to the arguments supplied to primitives, such as topic names and message queue sizes. If the arguments are literal values, defined on the spot, not only is the source code more readable but implementing a precise static analysis tool also becomes much more accessible. If, on the other hand, the arguments stem from dynamic variables (or function calls, even), sophisticated data flow and control flow analyses become a requirement.

We cannot have accurate data on the distribution of primitives under control flow constructs, relying solely on automated mining methods. To gather accurate numbers, we would require a full-fledged control flow analysis tool for C++. Building such a tool is a complex and daunting task, and one of the aims of this research question is precisely to understand whether such an endeavour is worth the time it requires. So, with this *chicken and egg* problem on hands, we settled for a middle-ground approach. We have only accounted for control flow constructs, e.g., **`if`** or **`for`**, within the same function as the primitive; we did not perform a full control flow analysis of the entire program. With this limitation in mind, we have found that the number of conditional primitives is rather low, as shown in Figure 10a, amounting only to 74 out of

748 primitives (about 10.43%). This seems like a rather promising figure for static analysis tools, if the actual numbers, when considering the control flow of the entire program, are not much higher.



(a) Absolute values.

(b) Occurrence in packages and applications.

Figure 10: Distribution of primitives under control flow.

In terms of packages, as shown in Figure 10b, we have found 19 packages (10.86%) that use conditional publishers or subscribers, while only 2 packages (1.14%) use conditional services. One of these packages belongs to Yaskawa Motoman, in which services are advertised whithin a loop, and another belongs to Aubo, in which a service is created if the driver initialises successfully. Also in Figure 10b, we can see that nearly one third of all applications (111, 30.41%) use conditional topics, and 64 applications (17.53%) use conditional services. Considering how few packages use these features, here we can see just how often different applications reuse the same packages.

The analysis of argument types is shown in Figure 11. This time, we consider not only the arguments for communication primitives, but also for the ROS time primitives. We studied ROS names (both for topic and services), message queue sizes and time amounts for `ros::Time`, `ros::Duration`, `ros::Rate` and their `Wall` variants. The time primitives are relevant, for instance, to perform analyses related to publishing rates, as in [156].

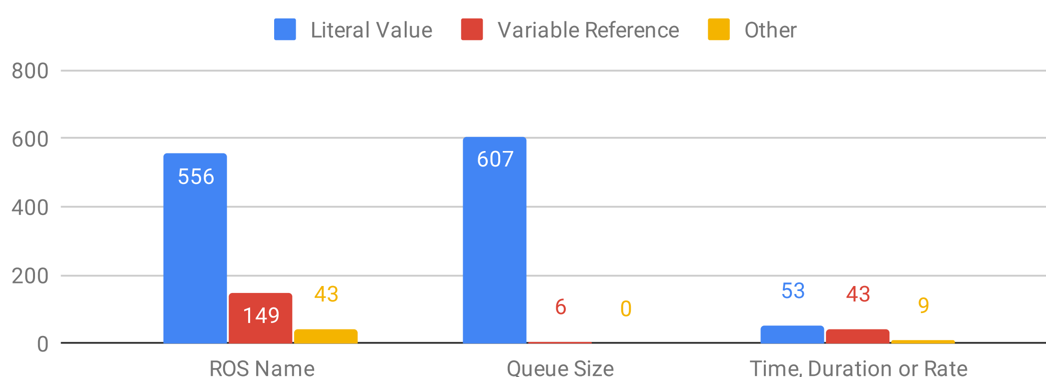


Figure 11: Types of arguments for various ROS primitives.

The results show that, for ROS names and queue sizes, the vast majority of arguments is defined as literal values: 556 out of 748 ROS names (74.33%) and 607 out of 613 message queue sizes (99.02%) are literals. For the remaining arguments, we can see that 149 names and 6 queue sizes result from reading variables, and yet 43 names result from other expressions (e.g., string concatenation, function calls, etc.).

Non-literals can be found in 36 packages (20.57%) and 150 applications (41.10%). The figures for the time primitives are much more balanced. Only 53 out of 105 primitives are created from literal values (50.48%), while 43 are created from the values of variables and the remaining 9 come from other sources. Non-literal time values can be found in 29 packages (16.57%) and 88 applications (24.11%).

Further inspection of the argument values shows a few interesting facts. Among the literal values for ROS names, we looked for the presence of global ROS names. Their use is discouraged, in general, as it makes components less reusable. We registered 38 publishers, 40 subscribers and 4 service servers that use global names, for a total of 82 names out of 748 (10.96%). They were found in 23 packages (13.14%) and 61 applications (16.71%). Among the literal values for queue sizes, we registered 9 instances of unbounded queues (a highly discouraged practice) and a surprising 331 queues of size 1 (about 54%). While singleton queues are not harmful per se, they are worthy of notice and further thought. Singleton queues should only be used when there is no interest in processing all messages, but rather only the most recent ones, since ROS discards the oldest messages when a queue is full. Judging by the message types associated with the majority of these occurrences, we can estimate that singleton queues tend to be used for control messages (e.g., teleoperation commands) or highly volatile data (e.g., sensor data, especially camera related).

2.3.4 *RQ3 – What kind of features are typically used in ROS launch files to deploy applications?*

As described in Section 2.1, launch files are XML files with a specific set of tags and attributes. Launch files are interpreted, often with the `roslaunch` tool, and have their own semantics. Building an analysis tool that understands the whole launch file language is not overly complex, as the language itself is not very extensive. However, it still poses a few challenges, such as dynamic, external variables (e.g., environment variables) and conditionals. By measuring the use of the various tags and attributes of a launch file we can gain insight on how relevant each feature is, and how systems are deployed in practice.

Note that, for the purposes of this study, we focus on launchable applications, as previously defined, rather than individual launch files. We consider applications in their entirety; `<include>` tags are resolved, so that we can analyse systems as they are deployed in practice. This means that the contents of any launch files that are included multiple times will be counted multiple times as well.

Our corpus contains 365 top-level launch files, and 901 launch files in total. The primary use of launch files – and, consequently, of applications – is to deploy nodes, so we start our analysis with the `<node>` tag and the chart in Figure 12.

Overall, we registered a total of 1418 nodes, of which only 275 (19.39%) are unique. Such a low number is the result of compositional construction of applications. We have 901 launch files in total, but only 365 are top-level ones; this means that many launch files are reused as parts of a larger system, and thus, any nodes they launch end up repeating (possibly with different parameters) over various applications. Nodelets are seldom used, amounting to 119 out of the 1418 nodes. Besides being a relatively niche feature, they are also mostly used in the TurtleBot2, a system with 101 top-level launch files and 90 uses

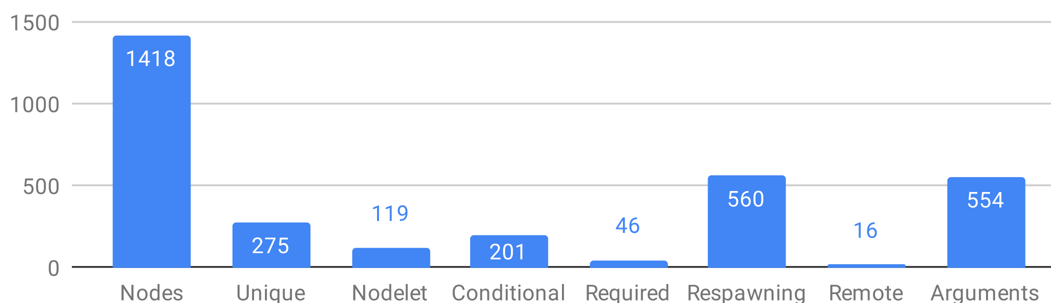


Figure 12: Uses of the `<node>` tag.

of nodelets in total. Out of the remaining 29 nodelets, 26 belong to three applications of the Fraunhofer IPA Care-O-bot system, and 3 belong to an application of the Clearpath Grizzly.

The most notable among the remaining node features are, perhaps, the use of custom command-line arguments, used in 554 nodes (39.07%), and marking nodes as able to *respawn*, used in 560 nodes (39.49%). To be able to respawn means that if the node terminates, for any reason, a new copy is started automatically, until the ROS system as a whole is shut down. Other node-related features are not really common. Only 46 nodes are marked as *required*, meaning that if the node terminates the whole launch application should be brought down. We also registered 201 *conditional* nodes, i.e., nodes under a conditional statement within a launch file, and 16 *remote* nodes, nodes that are launched on a machine specified by its network address that is (possibly) different from the one interpreting the launch file.

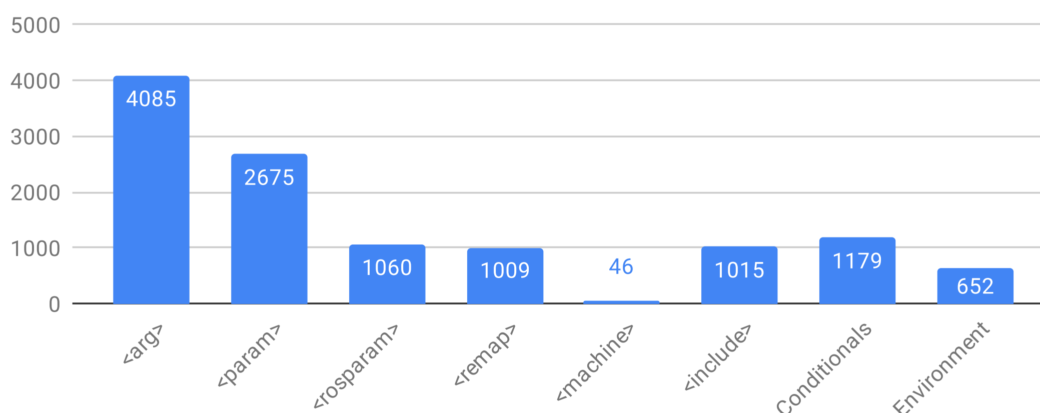


Figure 13: Overall use of launch file features.

The chart of Figure 13 shows the overall usage of other launch file tags and features within applications. Immediately, we are able to see that the use of the `<arg>` tag, whose purpose is to define local variables within a launch file, is commonplace with 4085 registered uses. A large number of these, 2069 to be precise, is used to pass along to `<include>` tags, i.e., variables defined at a high level to pass to lower-level launch files. As for the `<include>` tag, we observed 1015 instances. Both of the tags related to the ROS parameter server, `<param>` and `<rosparam>`, are also abundantly used, with 2675 uses and 1060 uses, respectively. Other frequently used features of launch files are remappings (1009 instances), reading environment variables (652 instances), and conditional tags (1179 instances). In contrast, the `<machine>`

tag, used to define or reference other machines within the network, is seldom used, with only 46 instances. Four of these appear on the Shadow Dexterous Hand system, and the remaining 42 are part of Fraunhofer IPA Care-O-bot applications.

Based on the observed figures, we can estimate that, on average, each launchable application remaps between two and three names, reads about two environment variables, and declares over three entities conditionally. There are implications from these statistics, as these features require additional analysis steps. For remappings, tools are likely forced to implement the same name resolution mechanisms that ROS uses. For environment variables, either user input is required (no automation) or the analysis has to take place in the same environment where the launch file is meant to be used (reduced portability). For conditionals, tools have to resolve arbitrary values, which may come, e.g., from environment variables.

2.3.5 *RQ4 – How and how commonly is the ROS parameter server used?*

The previous research question already sheds some light on how commonly the parameter server is used in the context of ROS applications (see Figure 13). Namely, the simpler `<param>` tag is used 2675 times, while the more complex `<rosparam>` tag is used 1060 times. On average, this amounts to over 10 parameters per application, if we took each tag to be equivalent to one parameter. However, in some contexts, either tag can define multiple parameters at once, so the actual number is likely much higher. In addition, out of all 3735 parameter-related tags, we observed that 978 (26.18%) define parameters from the contents of YAML data files, and 240 (6.43%) define parameters from the printed output of command-line programs. In summary, there are 2517 parameter tags whose effects are more or less immediate, while nearly one third of the tags is not self-contained within the launch files. These tags are scattered among 232 out of the 365 applications, meaning that nearly two thirds of all applications (63.56%) use the ROS parameter server.

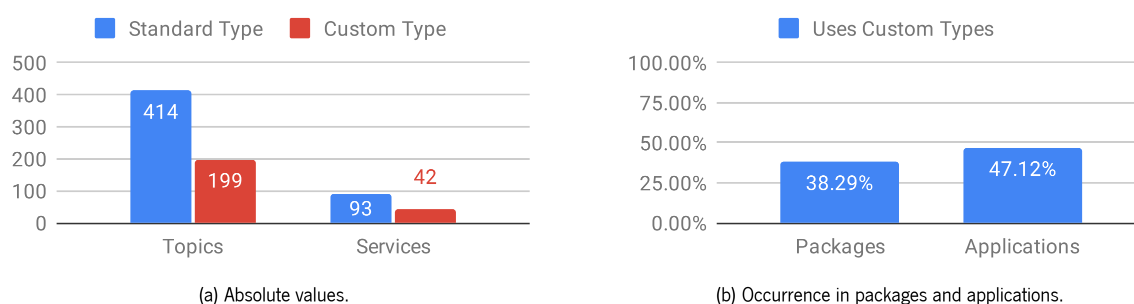
Regarding the use of the parameter server by the nodes, we have not found any surprises. As stated in Section 2.1, the parameter server is not designed for high performance, meaning that it should only provide access to static data. We found only 38 runtime write operations, compared to 705 readings. Out of these read operations, 585 (82.98%) declare a default value for when the parameter is not defined in the server, as a fallback behaviour. Most of the accessed names (666) are given as literal arguments too, which makes analysis easier. Overall, we can find accesses to the ROS parameter server in 87 out of 175 packages with C++ source code (49.71%).

Do note, however, that we have limited the study to the basic `getParam` and `setParam` variants of the ROS C++ client libraries. There are many means of indirectly using and defining parameters (e.g., with the `dynamic_reconfigure` package) that are beyond the scope of this study.

2.3.6 *RQ5 – How commonly are custom message and service types used in ROS communications?*

Defining new ROS message and service types is not overly common, although it is sometimes necessary. We have found that 74 out of the 380 packages in the corpus (19.47%) define new types. More specifically,

55 packages (14.47%) define 310 new message types, 42 packages (11.05%) define 153 new service types, and we have even looked into the definition of 35 new Action types, with 20 packages (5.26%) doing so. The chart in Figure 14a gives us more insight in terms of the absolute usage of these types. We can see that 199 out of 613 calls to `advertise` or `subscribe` use a custom message type (32.46%), while 42 out of 135 calls to `advertiseService` or `serviceClient` use a custom service type (31.11%). In relative terms, as shown in Figure 14b, we can see that a total of 67 packages out of the 175 that include C++ source code (38.29%) use custom types with calls to the ROS primitives. As for launchable applications, these packages are required for 172 out of the 365 (47.12%). Given these figures, we can conclude that using custom messages is a fairly common practice in ROS system development.



(a) Absolute values.

(b) Occurrence in packages and applications.

Figure 14: Use of custom message and service types.

2.3.7 *Impact on Static Analysis*

While feature-wise statistics are interesting, from a static analysis perspective it is more relevant to consider the combined usage of different features, namely, what would be the expected coverage of a static analysis tool if only a given set of features is supported.

Our findings show that over half of all values passed to primitives are literals. This is a fortunate prospect for analysis tools (as it avoids symbolic execution), but not too surprising. Most resources in the ROS Wiki use literal values, and people tend to learn by example. In addition, we found that, most of the time, primitives do not occur within loops or conditional blocks – another fortunate outcome for static analysis.

Two prominent and problematic usages of primitives, concerning RQ2, are conditional occurrences and non-literal arguments, especially for arguments served from ROS parameters. Our study shows that 137 of the 175 packages containing C++ code (78.29%) and 232 of the 365 applications (63.56%) do not use these features at all, and thus could be handled by a relatively straightforward static analyser. However, only 60 packages out of the same 175 (34.29%) and 42 applications (11.51%) *do* contain some primitives, with none using these features. This means that the larger part of packages and applications, that do not use either conditionals or dynamic values, does not use topics or services at all.

Unfortunately, this study also shows that the gains of supporting just one of these features, either non-literal arguments or conditionals, would hardly be noticeable: the former would improve coverage from 60 packages to 64 and from 42 applications to 55; the latter would improve coverage from 60 packages

to 79 and from 42 applications to 49. Not only that, but the parameter server is often used in packages – about half of all packages containing C++ use it with the basic read and write primitives – which means that tools likely have to be aware of it too. So, the main outcome of these statistics is that, in order to achieve a high coverage of ROS applications, it would be necessary to support a fairly complex, feature-rich analysis.

2.4 Summary

This chapter covers the necessary background for this thesis. It starts with an overview of the main characteristics of the Robot Operating System, in terms of software distribution, but also in terms of software architecture. Namely, we were able to see how ROS organises and distributes software in packages; that it is a distributed system where nodes communicate with each other via message-passing; and how systems are often orchestrated and deployed via XML launch files.

Next, we went through the state of the art in Formal Methods and Quality Assurance techniques specifically applied to ROS. Most of the published research focuses on models, with varying levels of formalism, and in many cases with the main purpose of synthesising code. This is certainly a step forward that could bring ROS closer to the norm in safety-critical systems. But, alas, roboticists come with all sorts of backgrounds, and traditional software development processes (with little to no modelling) are commonplace. Thus, static and dynamic analyses seem more promising for current and existing systems, especially when coupled with reverse-engineered models – in particular, the ROS computation graph – to support further model-based analyses (such as model checking).

Lastly, the chapter concludes with our first contribution: an empirical study on the usage patterns of ROS primitives. This study is more of an educational contribution that sheds some light on the general coding practices of ROS users. This understanding is valuable in various ways – it could be used to promote certain (perhaps less known) features, or to design coding guidelines that avoid problematic ones. In the scope of this thesis, however, its main purpose is to show that novel ROS aware analyses (including our own), when applied to real ROS applications, need to handle a large spectrum of its features. Given all the flexibility and dynamic aspects of ROS, this poses quite the challenge.

3

MODELLING AND REVERSE-ENGINEERING ROS SYSTEMS

The previous chapter introduced the necessary concepts to understand and build a ROS system. It describes how nodes communicate with each other via topics and services, forming a network called the ROS Computation Graph. This graph, being one of the main characteristics of ROS, should be at the core of any domain-specific analysis. The main challenge is that, to come up with new analysis techniques that are applicable to any ROS system, we cannot rely on typical Model-driven Software Development – where models are built explicitly and before the source code. Indeed, anecdotal experience shows that many (if not most) ROS systems are not built this way, but rather by jumping straight to writing code, after a relatively informal design phase. Thus, an opportunity for automated extraction of ROS architectural models presents itself – even if the extracted models are (likely) not complete.

Model extraction and, more broadly, reverse engineering are not straightforward at all. What makes a ROS architecture model? Is the computation graph enough, or do we need additional information? How should this information be retrieved? In this chapter we propose answers to these questions with one of our main contributions: a metamodel for ROS systems [151], and the respective automatic extraction procedure.

3.1 State of the Art

The overall topic of building and reverse engineering architectural models for a software system is immense, and has been under research for a long time. One key characteristic of the current state of the art, is that there are no truly general-purpose solutions. In the same sense that there is no single one-size-fits-all programming language – or programming paradigm, even – the same applies to the way one designs a software system. Architectural models are often designed with some domain, application or paradigm in mind. And this is not just out of convenience; models become more readable and easy to use when their traceability to the implementation is evident.

The same reasoning applies to ROS. Even though ROS reuses a number of already well-established concepts (e.g., independent components dynamically deployed over a network, the publisher-subscriber model, or the client-server model), it also features its own domain-specific vocabulary, and a number of domain-specific concepts (e.g., launch files), making room for tailor-made architectural models. In the remainder of this section, we present some of the most relevant proposals – both in modelling architectures

and in reverse engineering of such architectures – that either have been directly applied to ROS or could be applicable to some extent.

3.1.1 *Architectural Models*

By far, the most common approach to design architectures is the component-connector model [3, 36, 38, 39, 77, 99, 169, 172]. A *component* is a (typically high-level) unit of computation or data storage. Components are often independent parts of a system with input (resp. output) *ports*, where they consume (resp. produce) externally visible data. Components are composed together by establishing *connectors* between ports, where a connector is a communication mechanism with, at least, two roles. For instance, a publisher-subscriber connector provides the roles of publisher and subscriber, which, when assigned to specific ports of components, make said components the publisher and subscriber of that connection.

This model is abstract enough that it does not refer directly to source code structures (such as classes or modules), but also concrete enough that traceability is not lost. Overall, it is easy to understand, and easy to realise with both textual and graphical languages. UML, and UML-inspired diagrams are a standard approach to this kind of architectural design (see [36, 38, 39, 99, 169, 172] for examples). Another advantage of the component-connector model is that, in its simplest form, the model does not impose restrictions on the implementation. The same architecture could be used for a monolithic, single-threaded process, as well as for a distributed system with third-party components implemented in different programming languages. In ROS, this translates well to using nodes as components, and using topics, services and actions as connectors, but most approaches do not support the full catalogue of ROS features – nodes and topics, at least, are the norm.

There are a few approaches that stand out of the classical box-and-line UML component-connector model. For instance, the AADL language is proposed in [16] as a better alternative to UML for ROS applications. While UML is the most popular modelling language, and there are plenty of UML solutions for robotics, the authors argue that these are not really adopted in practice. AADL is the second most popular language [16] and benefits from having well-defined analysable semantics and making hardware components a part of the model – both desirable traits in a robotics environment. However, it has a deep focus on the cyber-physical and embedded aspects of robotics; i.e., hardware, devices, threads, memory or processors are key aspects of an AADL model, but we target the high-level software architecture.

Another approach [103] proposes a domain-specific textual language to specify ROS architectures. Its main goal is to model nodes and launch files, so that the plethora of soft identifiers given as topic or parameter names do not have to be checked manually for consistency (e.g., to ensure that remappings are correct). One of its drawbacks is that it only includes topics (not services or actions) and treats parameters as static (i.e., read-only, defined in launch files). Finally, [107] proposes the Robot Architecture Definition Language. This is a language with a precise model of computation and communication. Systems are encoded as calendar automata, which enables the verification of some real-time properties using SMT-based infinite-state bounded model checking. However, the model imposes some restrictions, such as requiring known latency bounds for all communication channels, the loop period of nodes, as well as worst-case

execution times. The main loop of nodes is also assumed to read messages from subscribed topics, execute a computation, and then publish on advertised topics.

3.1.2 *Model Extraction*

Model extraction, sometimes also referred to as reverse engineering, is the process of retrieving a model (often, an architectural model) from an existing system. One of the primary purposes of model extraction is to equip architects and developers with a model of an implementation, to check for compliance against a reference model of the intended architecture. As such, it is a useful tool during the whole development cycle, as systems evolve and significant architectural changes might be introduced. It also plays an important role in handling and understanding legacy systems that lack a reference model of the intended architecture. In this particular case, automated reverse engineering has the added benefit of enabling model-based analyses, without the (very significant) cost of building models by hand.

The extraction process itself can be applied to a system execution (dynamic analysis), to the static artefacts that compose such system (static analysis), or a combination of both. Dynamic analysis tends to be the easier approach to implement, especially for dynamic, reconfigurable architectures, as is the case in ROS. Its main drawback is the inherent limitation of dynamic analysis to extract information only from a limited number of traces, possibly not achieving full coverage. Static analysis is orthogonal to this approach. It is able to extract sound information, covering all possible executions of the system, at the cost of increased complexity [1]. Static analyses can also be hindered by dynamic values, impossible to determine without executing the system, and by false positives or over-approximation, often caused by being unable to identify and ignore unreachable code. Depending on the purpose of the reverse engineering process, it can be difficult to answer all questions strictly with either static or dynamic analysis. Some questions are better answered with static analysis, while others lend themselves better to a dynamic procedure [72].

Dynamic Analysis   The most immediate example of dynamic analysis within a ROS environment comes from one of the standard ROS introspection tools, `rqt_graph`[1]. This tool provides a graphical user interface that displays the various elements of a running ROS computation graph, such as nodes, topics or services. In addition, it is able to display runtime statistics, such as the publish frequency of a given topic or its average throughput in bytes.

Beyond the out-of-the-box ROS tools, an example of a dynamic analysis approach to ROS can be seen in [96]. The authors argue that verifying correctness of publisher-subscriber systems in robotics middlewares, such as YARP[2] [122] and ROS, depends on a correct usage of said middleware. But middleware models tend not to be available upfront, because they are often large and complex. Thus, they propose inferring finite-state automata that model the behaviour of a concrete publisher-subscriber system, and combine them with external control software models, to achieve a full picture. They apply this approach to YARP

1 http://wiki.ros.org/rqt_graph
2 https://www.yarp.it/

by analysing the interactions of the middleware with an enbedding context – essentially, a sandboxed execution. Afterwards, they use model checking to verify the absence of deadlocks and other scenarios that might lead to loss of messages between components.

Static Analysis On the other end of the spectrum, there are a few approaches based on static analysis [1, 2, 47, 127, 142, 156], although only a few are ROS-specific [127, 142, 156].

The work of Abi-Antoun and Aldrich [1] proposes a static analysis solution to extract the runtime architecture of object-oriented systems. They argue that previous static analysis solutions approach the problem from a low level of abstraction, too close to the source code constructs, where the resulting graphs are non-hierarchical and not scalable to large programs. They present an extraction process that is able to aggregate related objects into higher-level components and partition them by architectural tiers, resulting in a model that is called an Ownership Object Graph. The main drawback is that it requires developers to add type and ownership annotations throughout the source code, to guide the extraction process.

Acher et al. [2] tackle a slightly different kind of problem: how to accurately identify variability points, not in a single system, but in Software Product Lines? Software Product Lines are families of systems that share common functionality, and allow for the addition, removal or interchange of features (called variability points). Feature Models are a type of model designed precisely to capture variability, but building large Feature Models by hand is time-consuming and error-prone – thus, the need for reverse engineering. This process starts with an over-approximation of all possible feature combinations, which is then refined by analysing plug-in dependencies, known constraints, and by integrating software architect knowledge. Human intervention prevents the process from being fully automatic, but the authors advocate that full automation is not desirable in this case, for integrating the software architect's knowledge is deemed too valuable.

The problem of applying static analysis to concurrent and distributed asynchronous message-passing architectures is explored in [47]. The authors focus on the Erlang language, which has seen extensive real world use, but poses a considerable challenge, given the flexibility of its message-passing primitives, and being a dynamically-typed, higher-order language. The proposed static analysis identifies actors within the system and matches message sending primitives with the corresponding message reception primitives, thus building a *communication graph*. This analysis is completely automatic and aims for a balance between soundness and completeness. It enables the detection of the following error patterns:

1. *Receive with no messages* – a process blocks waiting for messages when it is known that its mailbox will be empty.
2. *Receive of the wrong kind* – a process expects a type of message that does not correspond to the messages it actually receives.
3. *Receive with unnecessary patterns* – a process specifies response code for message types that it will never receive.
4. *Send nowhere received* – a process sends a message to another process that does not expect any messages or does not handle such message types.

Lastly, there is also existing work applied to ROS, still with a pure static analysis approach [127, 142, 156], as presented in Section 2.2. The first published approach [142] extracts the publisher and subscriber part of the ROS computation graph, from the source code of C++ nodes. The extraction process traverses the source code and identifies function calls to the various ROS primitives (e.g., **`advertise`**). Each call is annotated with control flow information, i.e., conditions that might affect it. It further distinguishes reactive and proactive message publications by tracing back calls to **`publish`** until it finds a callback function (in which case the publication is reactive) or a leaf node in the source tree (e.g., the **`main`** function, in which case it is proactive).

The second approach [156] also focuses on the publisher-subscriber aspect of the computation graph, but the model is enriched with launch file information, such as remappings. Similarly to how the previous approach classifies message publications as reactive or proactive, this approach classifies publishers as dependent or independent. The main goal is to conduct rate impact analysis – i.e., to determine the impact of changing a single node, especially with respect to its publication frequency. The naïve stance would be to assume that all nodes transitively reachable from the changed node would be affected, but this might not be the case. If the connected subscribers do not publish messages based on the rate of incoming messages – in other words, if publishers are rate-independent – the nodes are, in principle, not affected by the changes. A publisher is deemed to be rate-independent when it cannot be traced back to a callback function or when it is used under a fixed-rate construct (e.g., timers or loops with adaptive sleep).

More recently (and published at the same time as our own approach [151]), Muscedere et al. [127] used static analysis to identify *Feature Interactions*. A Feature Interaction occurs when two features of a system, that work correctly on their own, intefere with one another when executed at the same time. Simply put, a *feature* is a high-level component that is nearly independent, e.g., a ROS node. Similar to our approach (presented in this chapter, and its implementation in Chapter 5), the proposed toolchain relies on an Abstract Syntax Tree of the C++ ROS components to extract a database of *facts* (function calls, variable assignments, control flow, etc.). With the use of relational algebra transformations, they are able to infer additional facts, relative to component dependencies and information flow. Then, a query engine is used on top of the fact database to look for user-defined patterns that represent potential Feature Interactions.

Mixed Analysis Many approaches to model extraction recognise the complementary nature of static and dynamic analysis, and end up employing a combination of the two [98, 120, 147, 157, 170].

The work of Silva and Campos [157] combines static and dynamic analyses in the context of Web application GUIs. Dynamic analysis is used to crawl the interfaces, looking for user inputs (e.g., buttons and forms), and to build the initial state diagram of the application. Since JavaScript event handlers are always available on the client side, static analysis can be exploited to identify variables that lead to different state transitions from the same user action. Conditions and value combinations are then tested, until a final version of the model is achieved.

Riva and Rodríguez argue that, in order to better describe and understand software architectures, multiple views of the system are necessary [147]. Their work proposes combining structural and dynamic information using a four-step iterative process.

1. Definition of architectural concepts. What is a component, and how do components communicate?
2. Data gathering from various sources. Employ both static and dynamic analysis. Also make use of available documentation and expert knowledge.
3. Abstraction. Transform the low-level model generated from the previous step into different high-level, domain-specific views. A rule-based Prolog system is used to formally define model to view transformations.
4. Presentation. How to present logical, process, physical and development views of the system. Directed graphs are proposed for static views (system structure), while simplified message sequence charts are the model of choice for dynamic aspects (system behaviour).

Krogmann's PhD thesis [98] addresses the problem of reverse engineering component-based software architectures for the design and evaluation of performance properties. Krogmann states that no previous satisfying approach existed to reverse engineer behaviour and performance models for component-based software architectures, based on a language independent code analysis. Behaviour models of components need to be highly parameterised with regards to: i) changing usage (number of users, user interaction with the system, varying amounts of data); ii) changing assembly (connecting different components or different component implementations); iii) changing execution platforms (fast versus slow servers). This thesis proposes combining not only static and dynamic analyses but also statistical analysis, to achieve the reconstruction of static architectures, behaviour specification and highly parameterised, abstracted performance models of component-based software systems. Such models ultimately enable a range of reasoning and prediction techniques in:

- Sizing – to estimate the necessary hardware to handle specific workloads, or to achieve a certain degree of reliability and performance;
- Legacy software extension – to estimate the overall system quality after adding new components to a legacy system;
- Reuse – to estimate the impact of adopting an existing component implementation;
- Design optimisation – to estimate the overall performance or reliability of the system.

The domain of microservice-based software is one that shares similarities with ROS. Microservices decouple network-accessible components, enhancing independent development of components, deployment and scalability. As is the case with ROS, the architectures of these systems are highly dynamic. Often they are not defined upfront, but rather emerge by dynamically assembling services into systems. This makes it hard to extract component relations from static sources and artefacts. In [120], the authors present an architecture extraction prodecure that combines static service information with infrastructure-related and aggregated runtime information for REST microservices. Architectural information is organised in three levels:

1. Service – mostly static information, such as the service API, organisational information and the domain model;
2. Infrastructure – service requirements, execution environments, deployment and scalability;

3. Interaction – observed communication among services (requests and responses) gathered from the runtime, and aggregated over long term windows.

Lastly, in the domain of ROS, as presented in Section 2.2, Witte and Tichy [170] propose a process to extract the ROS computation graph that uses static analysis for launch files and dynamic analysis for node interfaces. They deem static analysis alone to be too complex, because resources can be added to the computation graph at any time, and launch files reference node executables – finding the corresponding source code is not trivial. Their approach has nodes running within a sandboxed environment which, with the help of an additional library, intercepts calls to the ROS Master, in order to build the dependencies on topics and services. However, as stated by the authors, resources can be created at any time, and their approach assumes that nodes follow a standard life cycle, in which all resources are created during set up. A general approach would have to monitor nodes indefinitely. They acknowledge this potential weakness, and propose the countermeasure of having the user provide annotations as launch file comments, to specify the missing bits of the extracted model, as seen in Listing 3.1.

```
1 <launch>
2   <node name="listener" pkg="roscpp_tutorials" type="listener">
3     <!-- <topics>
4       <topic name="chatter" type="String" class="sub"/>
5     </topics> -->
6   </node>
7 </launch>
```

Listing 3.1: Specifying subscriber information as launch file comments [170].

3.2 A Metamodel for ROS Applications

There is no lack of research in modelling ROS systems. And with good reason, since models enable a wide variety of architectural and behavioural analysis and validation techniques. Still, we find a detrimental gap in the current state of the art. Notice how most of the existing approaches fall under (at least) one of the following categories.

1. The approach requires expert knowledge (e.g., in Formal Methods). It is not amenable to be used by a typical ROS developer.
2. Even though models are used, no metamodel is explicitly documented.
3. Some aspects of ROS are left out.
4. There are assumptions about how the concepts should be used (e.g., nodes following a specific life cycle).
5. The models only include entities from the system's runtime (e.g., nodes and topics, but not packages and source files).

The last issue, in particular, is very prominent. As described in Chapter 2, there are two sides to ROS systems: the runtime, which is mostly composed of the ROS computation graph and often tackled by other

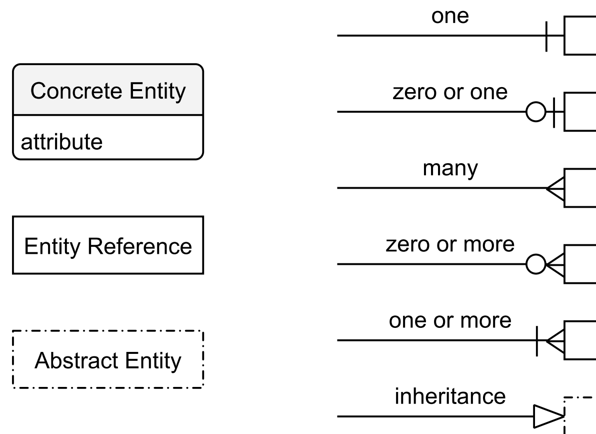


Figure 15: Diagram notation.

authors; but also the static artefacts, at the file system level, such as packages and source files. It is from the latter that nodes are built and systems are set up in the first place. A number of different analyses can take place solely at this level, even [52, 139, 148]. There is a whole graph of package dependencies to explore, not to mention the more intricate dependencies among source and launch files. A metamodel that is truly complete should encompass all these entities as first-class citizens, and exploit the relations between them. One of the most useful (and often disregarded) relations is *traceability* – identifying culprit source artefacts when a problem is encountered in a node, and vice-versa. Source artefacts and ROS runtime resources are two sides of the same coin.

In this section we propose a metamodel to describe the software components of ROS systems [151] that addresses the aforementioned issues. The various key entities of this metamodel are described with a schema for YAML data representation. For readability purposes, we defer the schema's definition to the Appendix B; in this section, we simply use examples of the schema. We further illustrate the concepts with class diagrams, where edges use the Entity-Relationship notation for cardinality, dashed boxes represent abstract entities, and solid boxes represent concrete entities. We use abstract entities for readability purposes. All abstract entities have concrete instances. We use sectioned boxes, with entity name and list of attributes, for entities defined in the current diagram. Plain boxes, just with an entity name, are used to reference entities that are defined in other diagrams. Figure 15 shows the diagram notation.

The remainder of this section goes into detail about each of the entities in the metamodel. The modelled entities are split into two groups, for the source artefacts and runtime entities. Some entities and relations are constrained by rules that cannot be captured either by the schema or by the diagram notation. In these cases, we will provide textual explanation, and assume the existence of a model validation tool. Some concepts provided by libraries, such as Actions and Dynamic Reconfiguration, were left out for future extension, i.e., we do not have concrete entities in the metamodel for these features. In spite of this decision, note that these features are often implemented in terms of the basic Topic, Service and Parameter primitives – meaning that, at a risk of losing precision, the metamodel is flexible enough so that they can still be captured.

3.2.1 *Source Entities*

A *Source Entity* is any entity that is static and whose purpose is to build the ROS system. They are often easily identifiable via artefacts in the file system (e.g., files and directories). We will go over familiar entities first, related to the development process itself, and then transition to the entities that are more closely tied with the ROS runtime.

Source File A *Source File* is a single file in the file system. We assume that all files belong to a single ROS package, i.e., there are no orphan files. Files are characterised by their *name* (a relative path within their package) and their *language*. The *language* of the source file is one of: C++ (`cpp`), Python (`python`), Package XML (`package`), Launch File (`launch`), CMake (`cmake`), ROS Message (`msg`), ROS Service (`srv`), ROS Action (`action`), or `unknown` for unspecified or unknown languages. These are the most common alternatives, but it is trivial to extend this enumeration to allow, e.g., JavaScript, which in recent years has also seen some application in ROS.

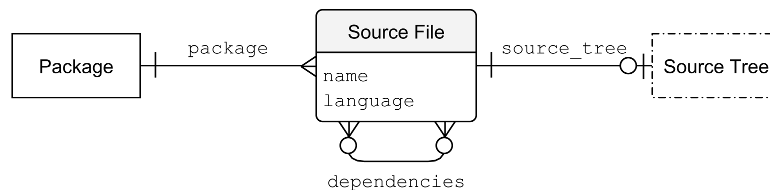


Figure 16: Class diagram for Source Files.

Optionally, each file can provide its own *source tree*, an Abstract Syntax Tree under some standard or convention, whose details are not part of this metamodel. Lastly, files can specify a set of dependencies on other files. For instance, C++ source files depend on the header files they **`#include`**, Python scripts depend on all scripts they **`import`** and launch files depend on other launch files that they **`<include>`**. Figure 16 shows the class diagram for Source Files. Listing 3.2 shows the YAML specification, according to the metamodel schema, for the manifest file of the `fictibot_drivers` package, from our Fictibot running example.

```yaml
name: "package.xml"
package: "fictibot_drivers"
language: "package"
source_tree: null
dependencies: []
```

Listing 3.2: YAML specification of a Source File.

Package A *Package* is an abstraction for a ROS Package. One of the mandatory characteristics of packages is that they must contain the package XML manifest file. From this file, we can gather a few metadata, such as the set of *authors* and *maintainers*, the *version* number, *dependencies* on other packages, the file system *path* and whether it is a *metapackage* (Figure 17). While most of this information

is not crucial for most kinds of analyses, it can be helpful, for instance, to assign issues, or to record evolution over time (via version numbers).

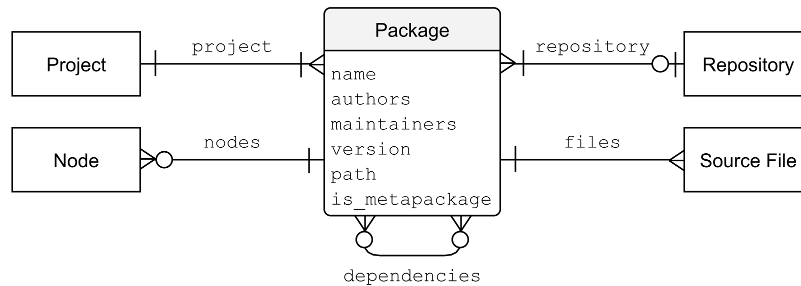


Figure 17: Class diagram for ROS Packages.

We assume that each package belongs to a *project*, may be part of a *repository*, and can build any number of ROS *nodes*. Since the names of files are relative paths within a Package, it follows that no two files within the same Package can have the same *name*. Uniqueness of the file names is enforced in the schema. Listing 3.3 shows schematic YAML for the `fictibot_drivers` package.

```
1  name: "fictibot_drivers"
2  authors: {name: "André Santos", email: "andre.f.santos@inesctec.pt"}
3  maintainers: {name: "André Santos", email: "andre.f.santos@inesctec.pt"}
4  version: "0.1.0"
5  path: "/home/ros/ws/src/haros_tutorials/fictibot_drivers"
6  is_metapackage: false
7  dependencies: ["roscpp", "std_msgs"]
8  project: "Fictibot"
9  repository: "haros_tutorials"
10 nodes: ["fictibot_driver"]
11 files:
12     - "package.xml"
13     - "CMakeLists.txt"
14     - "include/fictibot_drivers/motor_manager.h"
15     - "include/fictibot_drivers/sensor_manager.h"
16     - "src/driver_node.cpp"
17     - "src/motor_manager.cpp"
18     - "src/sensor_manager.cpp"
```

Listing 3.3: YAML specification of a Package.

Repository A *Repository* is a standard source code repository. As seen in Figure 18, it is characterised by its *name*, version control system (*vcs*, e.g., `git` or `svn`), its file system *path* and its *version* (e.g., branch name in `git`). Optionally, not shown in the figure, metadata such as the number of commits, contributors, the repository's URL or the issue tracker can also be appended. An implicit rule, not captured by the schema, is that the path of every Package under a Repository must be a (direct or indirect) child path of the Repository's path.

While repositories are not a core, or necessary, concept in ROS, they are a natural part of software development. Including them in the metamodel enables, for instance, gathering process metrics (commits

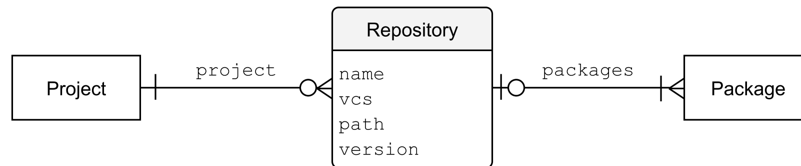


Figure 18: Class diagram for Repositories.

and contributors) as well as refining traceability, and possibly pointing issues directly to an issue tracker. It also aids in tracking evolution over different versions. Listing 3.4 shows schematic YAML for the `haros_tutorials` repository[3], the repository that contains the Fictibot packages.

```yaml
name: "haros_tutorials"
vcs: "git"
version: "master"
path: "/home/ros/ws/src/haros_tutorials"
packages:
    - "fictibot_drivers"
    - "fictibot_controller"
    - "fictibot_msgs"
    - "fictibot_multiplex"
```

Listing 3.4: YAML specification of a Repository.

Project A *Project*, like a Repository, is a concept that is not directly introduced in the ROS documentation, even though it exists (implicitly or explicitly) in practice. The main purpose of this entity is to aggregate a set of packages that should be part of the same logical unit, such as a robot with a specific application. Projects are not partitions of packages. There can be multiple projects using the same packages, although probably with different configurations. There can only be one project per model – the project is the main entity (and entry point) of the whole model.

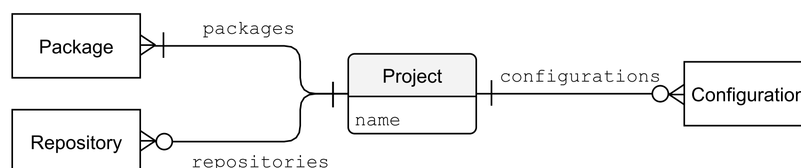


Figure 19: Class diagram for a ROS Project.

As seen in Figure 19 and Listing 3.5, besides containing a set of *packages*, a project may also contain a set of *configurations* (a runtime entity). Thus, along with Nodes (as we will see), Projects are a bridge between source and runtime entities.

Node The term *Node* in ROS is actually a bit ambiguous, as it can mean one of two things. It could be a node in the computation graph, a process interacting with the system, or it could be an executable, built

3 https://github.com/git-afsantos/haros_tutorials

```yaml
name: "Fictibot"
packages:
    - "fictibot_drivers"
    - "fictibot_controller"
    - "fictibot_msgs"
    - "fictibot_multiplex"
configurations: ["minimal"]
```

Listing 3.5: YAML specification of a Project.

from source code, from which the runtime nodes are instantiated. In this metamodel, *Node* means the latter. We use *Node Instance* for the former.

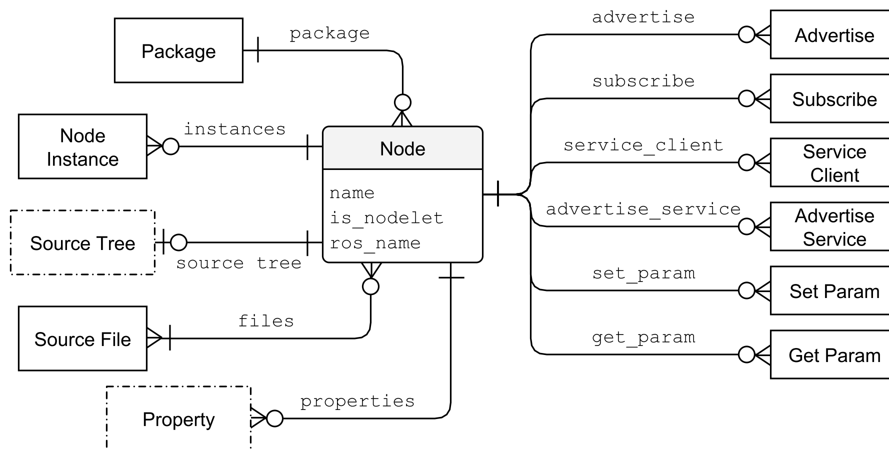


Figure 20: Class diagram for a ROS Node.

As seen in Figure 20, a node is identified by its executable *name*, always belongs to a *package* and enumerates the source *files* needed to build it (that, in most cases, belong to the same package). The model should also tell whether the node can be loaded as a *nodelet*. Optionally, as is the case with source files, the node may contain the full *source tree* of the code that builds it, i.e., a merging of the source trees from all source files. *Properties* are textual properties about the Node's behaviour. The syntax and semantics of the specification language are given in Chapter 4.

This is one of the entities (along with Project) that provides a direct connection to the runtime view, via *instances*, which is the set of all Node Instances created from a particular node. If available, nodes also provide the *ROS name* that is used by default to identify the node, when launch files do not override it at instantiation. Finally, when possible, nodes should list all calls to ROS primitives throughout their code, i.e., all calls to the ROS interface that would create or use topics (*advertise* and *subscribe*), services (*advertise service* and *service client*) and parameters (*get parameter* and *set parameter*). Listing 3.6 shows an excerpt of schematic YAML for a Node specification. We do not include the full listing, since including all calls to the ROS primitives would impact readability.

ROS Primitive Call A *Primitive Call* is a function call to any of the ROS functions that create or make use of a resource, such as `advertise` that is used to create a topic publisher. As seen in Figure 21,

```yaml
name: "fictibot_driver"
package: "fictibot_drivers"
is_nodelet: false
ros_name: "fictibot_driver"
files: ["src/driver_node.cpp", "src/sensor_manager.cpp", "src/motor_manager.cpp"]
advertise:
  - name: "bumper"
    type: "std_msgs/Int8"
    queue_size: 21
    latched: false
    traceability:
        package: "fictibot_drivers"
        file: "src/sensor_manager.cpp"
        line: 10
        column: 29
```

Listing 3.6: YAML specification of a Node (excerpt).

some attributes and relations are similar between all types of primitives (although the names of the attributes change), but the model still treats each primitive type as its own entity, so that it is more easily extensible.

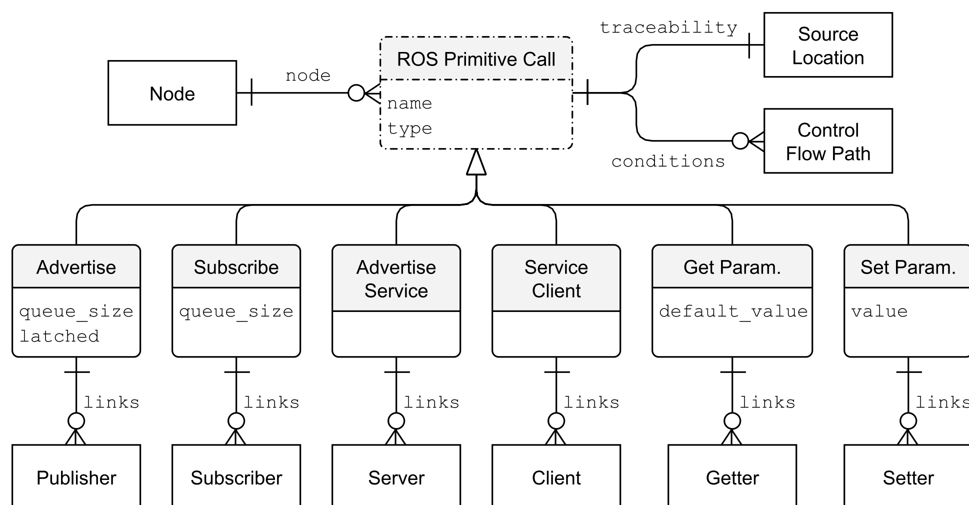


Figure 21: Class diagram for ROS Primitive Calls.

All primitives have a resource *ROS name* (e.g., the topic name given to a call of `advertise`, before applying any remappings) and a value *type* (e.g., the message type for topics). In addition, all primitives have a source code location, and possibly the control flow paths containing dynamic conditions. Primitives can be instantitated as *links*, a runtime entity of this metamodel. The *Advertise* and *Subscribe* primitives track the message *queue size*. Lastly, the Parameter primitives track default values (when reading and the parameter is not defined) and written values. Listing 3.6 already includes an example of an Advertise call in schematic YAML.

Source Condition and Source Location While not really entities per se, *Source Conditions* and *Source Locations* (Figure 22) are useful complex attributes that some other entities require.

Source locations point to specific locations in the source code, describing a *package* name, a *file* name, a *line* number and a *column* number.

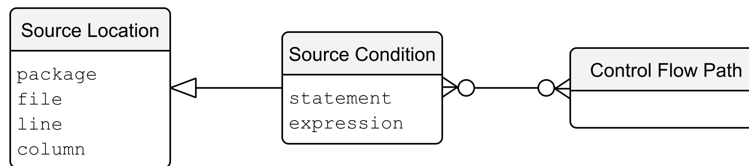


Figure 22: Class diagram for Source Condition and Source Location.

Source conditions are used to store dynamic conditions (as in the condition of an `if` or `for` statement) that might affect whether another entity is evaluated. They are essentially an extension to the source location, adding the type of *statement* and a string with the concrete *expression*. Multiple conditions can be grouped under a control flow *path*.

3.2.2 *Runtime Entities*

A *Runtime Entity*, in contrast to Source Entities, is an entity that is dynamic, and generally only exists while a ROS system is in operation. These are also some of the most iconic concepts in ROS, including node instances, topics, and the computation graph in general. Many runtime entities are *Resources* of the computation graph. A notable exception is the *Configuration* concept that we introduce as an abstraction for a ROS application, and as a slight extension of the computation graph.

Every resource, as presented in Chapter 2, has a *ROS name* that should be a unique identifier among the same class of resources. I.e., although not encouraged, a topic, a service and a parameter can all have the same ROS name. Two different topics sharing the same name, however, is not possible. In addition, every resource belongs to a single *configuration*, and might be considered conditional (i.e., it is not always part of the configuration). In that case, the resource should provide the control flow graph of dynamic conditions on which it depends. These may come either from source or launch files, depending on the case.

Node Instance A *Node Instance* is the runtime counterpart to the source entity *Node*. While the Node represents the executable file in the file system, the Node Instance represents a process spawned from a Node. It is a resource in the ROS computation graph.

As seen in Figure 23, besides including a reference to the original *node* executable, a Node Instance also provides the command line arguments given at launch to the executable and a table of remappings. Being a spawned process, its presence in the final Computation Graph might depend on a number of *conditions*, such as the conditional statements of a launch file. All ROS Primitive Calls in the original node are instantiated as ROS Links (*publishers*, *subscribers*, *servers*, *clients*, *getters*, *setters*). During this step, the namespace and remappings of the node instance are applied to the arguments of the primitives, yielding, thus, the concrete links. Listing 3.7 shows an excerpt of schematic YAML for a node instance.

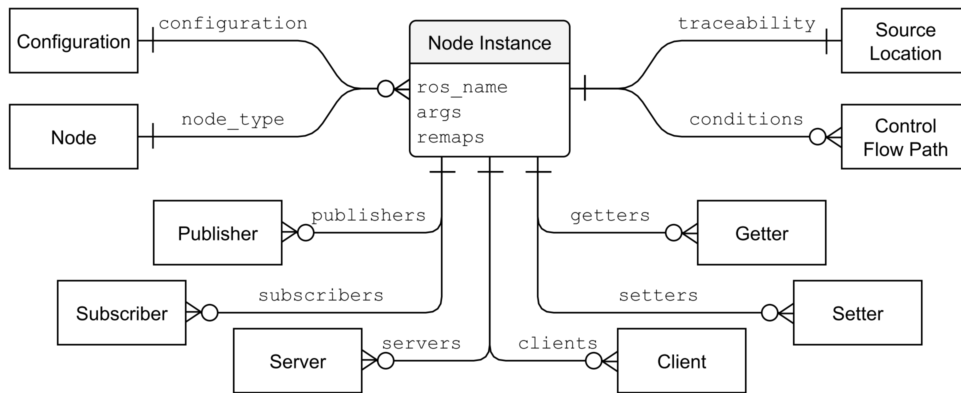


Figure 23: Class diagram for a Node Instance of the ROS Computation Graph.

```yaml
ros_name: "/fictibase"
configuration: "minimal"
node_type: "fictibot_drivers/fictibot_driver"
args: ""
remaps: {}
traceability:
    package: "fictibot_controller"
    file: "launch/minimal.launch"
    line: 2
    column: 3
publishers:
  - name: "/bumper"
    type: "std_msgs/Int8"
    queue_size: 21
    latched: false
    traceability:
        package: "fictibot_drivers"
        file: "src/sensor_manager.cpp"
        line: 10
        column: 29
```

Listing 3.7: YAML specification of a Node Instance (excerpt).

Topic A *Topic* corresponds to a ROS topic in the computation graph. As seen in Figure 24, topics are little more than convenient views of *publisher* and *subscriber* links on the same name.

Service A *Service* corresponds to a ROS service in the computation graph. As seen in Figure 25, services, like topics, are little more than a convenient view of a *server* and a set of *clients* on the same name.

Parameter A *Parameter* corresponds to a ROS parameter in the computation graph. As seen in Figure 26, a parameter provides the sets of its readers and writers. When a parameter is defined in a launch file, it should include the source location as well as any dynamic conditions.

ROS Link A *ROS Link* is yet another concept that does not exist explicitly, neither in the ROS documentation nor in the actual systems. It is the runtime counterpart to the ROS Primitive Call, and

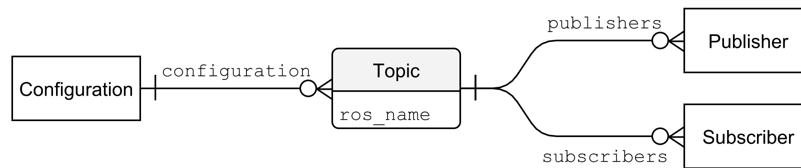


Figure 24: Class diagram for a Topic of the ROS Computation Graph.

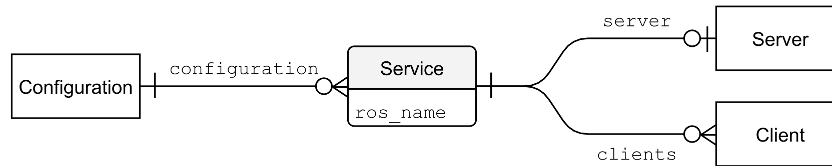


Figure 25: Class diagram for a Service of the ROS Computation Graph.

establishes a relation between Node Instances and the other Resources they use. As seen in Figure 27, a link contains a reference to the *node* instance and *primitive* call that create it. Some attributes of the original primitive call are duplicated, due to the possibility of them changing at runtime. For instance, ROS names can be remapped, queue sizes can be determined from parameters, and types can be set dynamically by passing the type itself as a variable (more easily in Python than in C++).

As is the case with primitives, most types of links do not have any special attributes beyond the common ones, but are still included to promote extensibility. Listing 3.7 already includes an example of a Publisher link.

Configuration A *Configuration* is the concept that we introduce to model a whole ROS system in runtime. As seen in Figure 28, it is very similar to a ROS computation graph, in the sense that it binds together the collections of *nodes*, *topics*, *services* and *parameters* of the system. Topics and services are given as a convenience and to promote familiarity with the traditional computation graph; a configuration can be fully determined given the sets of nodes and parameters. With the nodes, links come by extension and, thus, the whole graph of resources.

In addition to the typical computation graph information, the model also specifies its unique identifier, a human-readable *name*, the *project* it belongs to, the list of *launch commands* required to start the system, and a copy of the required *environment* variables. It can be annotated with *properties*, using the syntax and semantics given in Chapter 4, and with additional *user-defined attributes* (that have no special semantics within the metamodel). The properties, in this case, apply to the Computation Graph as a whole, rather than to single nodes. Listing 3.8 shows an excerpt of schematic YAML for a configuration.

3.3 Model Extraction

Once a metamodel is defined, we can instantiate it manually, automatically or using a combination of the two. Our ultimate goal is to improve the quality and development process of ROS systems, with as little specification effort from the user as possible, so automation (or partial automation) is our preferred

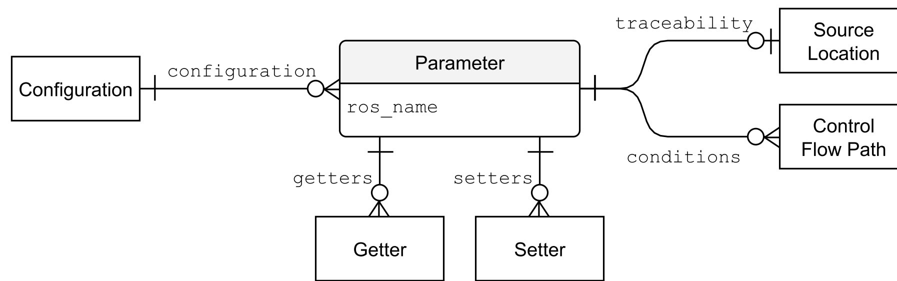


Figure 26: Class diagram for a Parameter of the ROS Computation Graph.

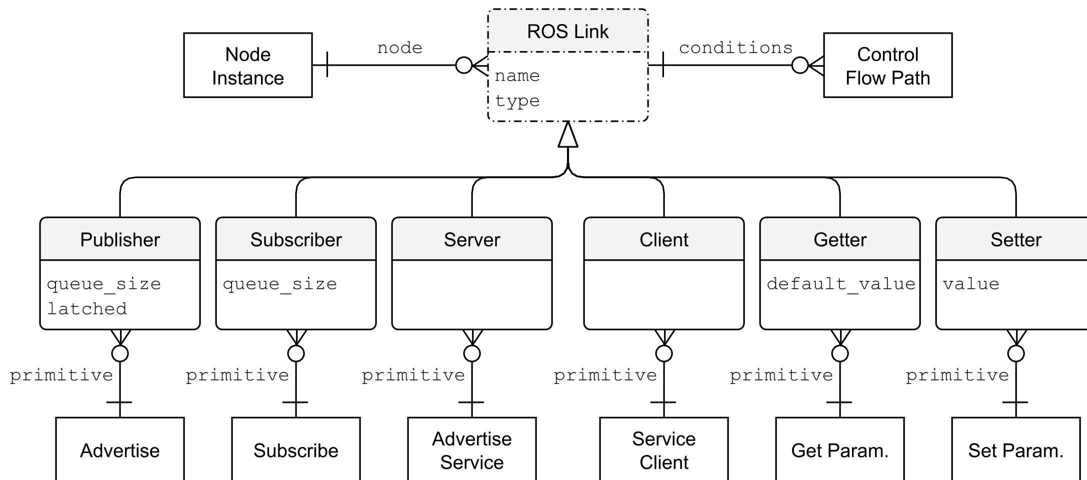


Figure 27: Class diagram for a ROS Link between Resources of the Computation Graph.

approach. Given the metamodel that we propose in the previous section, a great deal of the required information can be gathered automatically and statically – especially for the file system entities. For runtime entities, static analysis is deemed to be complex [170] but possible to some extent [142, 156].

In this section, we present an overview of the the model extraction process that is implemented in the HAROS tool (described in Chapter 5). Since dynamic analyses are incomplete by definition, we base all of the extraction process around static analysis. Despite its increased complexity, the availability of models at compile time with such a level of completeness (under optimal conditions) is something that dynamic analyses can hardly match. The fact that we already require static analysis for the file system entities is a bonus. But, even though we do not propose dynamic analysis as the core of our approach, using it as a complement to refine models is the next logical step for future improvement – which, at a high level, is the reverse approach to that of Witte and Tichy [170]. In the event that static anaysis is unable to fully resolve an entity, our approach builds entities out of partial information, as we will show. The presentation of the algorithm is accompanied by examples from our Fictibot running example. A more in-depth evaluation of the extraction capabilities of HAROS for real robot applications can be found in Chapter 8.

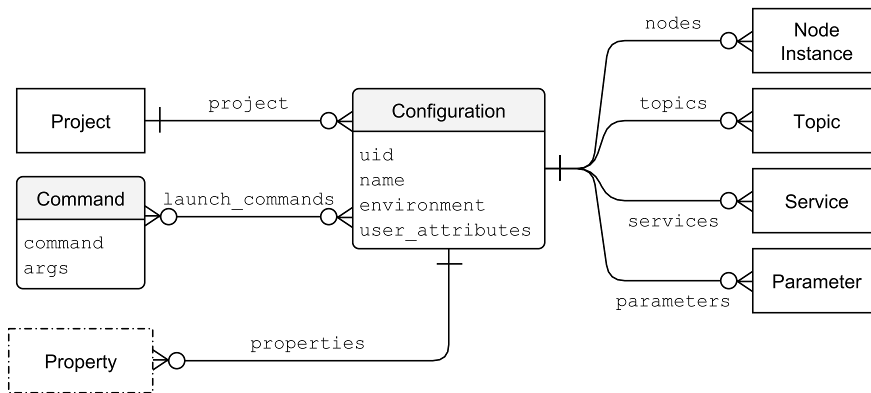


Figure 28: Class diagram for a ROS Configuration.

```yaml
uid: "minimal"
name: "Fictibot Minimal Configuration"
project: "Fictibot"
nodes:
  - ros_name: "/fictibase"
    node_type: "fictibot_drivers/fictibot_driver"
    # ...
  - ros_name: "/ficticontrol"
    node_type: "fictibot_controller/fictibot_controller"
    # ...
parameters: []
launch_commands:
  - command: "roslaunch"
    args: ["fictibot_controller/launch/minimal.launch"]
```

Listing 3.8: YAML specification of a Configuration (excerpt).

3.3.1 *Overview of the Algorithm*

Our aim is to automate the model extraction process as much as possible, and, indeed, most of the metamodel is amenable to automatic reverse engineering. However, some pieces of information must, inevitably, come from user input. In particular, our algorithm requires user specification for Configurations because ROS has no clear definition for *application* – systems are entirely dynamic and open ended. There is room for automation, for instance by considering top-level launch files as applications – as we did in [149] (see Section 2.3) – but, as we stated, this is not always accurate. Thus, we rely on user input to specify lists of launch files and `rosrun` commands (to launch individual nodes) that compose a single application.

Since the Project is the true entry point for a model, the algorithm for building a model can be seen as the algorithm for building a project. As such, piecing together all bits of user input we get what is, in fact, akin to a schematic YAML project specification (Listing 3.5). The main difference is that configurations include additional information besides their name, such as a list of launch files. The algorithm itself, at a high level, is straightforward and can be split into two complex steps – the extraction of the source code artefacts, and building configurations and runtime entities.

Listing 3.9 shows an abstract version of the algorithm in Python syntax, where the main function receives the YAML project specification and ends up building a Project, as per the metamodel. The

source code artefacts are built in the first three lines, with the `extract_packages_and_files`, `extract_repositories`, and `extract_nodes_and_primitives` functions. `P`, `F`, `R` and `N` are sets of Packages, Source Files, Repositories and Nodes, respectively. Then, configurations are built based on user input and the source entity data. For instance, Nodes are required in order to build Node Instances, and ROS Primitive Calls are required to build ROS Links. More details on `extract_configurations` are shown in Listing 3.12. Lastly, the project binds all entities together. Note that, as previously defined in the metamodel, the project needs only references to packages and configurations; all other entities are accessible from these.

```
1  def extract_metamodel(yaml_spec):
2      # ---- Source Code Entities ------------------------------------
3      P, F = extract_packages_and_files(yaml_spec["packages"])
4      R = extract_repositories(P)
5      N = extract_nodes_and_primitives(P)
6      # ---- Runtime Code Entities -----------------------------------
7      C = extract_configurations(yaml_spec["configurations"], P, F, N)
8      # ---- Project binds all Entities -------------------------------
9      project_name = yaml_spec["project"]
10     project = Project(project_name, P, C)
11     register_on_database(project, R, P, F, N, C)
12     return project
13
14 def extract_packages_and_files(package_list):
15     return # a set of packages and a set of files
16
17 def extract_repositories(P):
18     # packages are modified in place
19     return # a set of repositories
20
21 def extract_nodes_and_primitives(P):
22     # packages are modified in place
23     return # a set of nodes with primitive calls
24
25 def extract_configurations(config_specs, P, F, N):
26     return # a set of configurations, given source entities
```

Listing 3.9: Overview of the model extraction algorithm.

3.3.2 *File System Artefacts*

The first step to build a Project model is to extract all the source artefacts, as we have seen in Listing 3.9. Runtime entities are instantiated from source entities, as discussed in Section 3.2, so there is no other choice. The user input includes a set of package names that should be the main source artefacts under development for the project. This is to ensure that these entities are present in the database, to enable other kinds of analyses (Chapter 5).

Identifying packages is trivial. It boils down to a search for directories that match a given package name and the expected structure of a ROS package: the directory must contain a well-formed package XML manifest and a *CMakeLists.txt* file for the CMake build system. The `fictibot_controller` diretory, containing the package of the same name, and shown in Figure 29 is an example of such a directory. The

search for packages does not have to be blind, though. Some environment variables (e.g., ROS_PACKAGE_-PATH) and tools (e.g., rospack) can be used to easily find the path to a package.

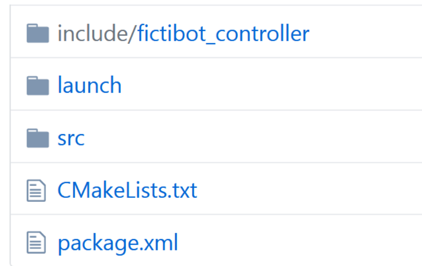


Figure 29: The fictibot_controller package directory.

Once a package is identified, traversing child directories yields the source files that belong to that package, while traversing parent directories determines whether the package is contained within a repository. In the fictibot_controller package we would identify the following files:

- ./package.xml
- ./CMakeLists.txt
- include/fictibot_controller/random_controller.h
- src/random_controller.cpp
- src/controller_node.cpp
- launch/minimal.launch
- launch/multiplexer.launch

The real challenge of this step is to identify nodes (extract_nodes_and_primitives). Nodes are the executable binaries compiled from source code (or executable scripts, in the case of Python). These artefacts are not part of the packages themselves, and there is no mention of them in the package XML manifest. The only way to identify which nodes a package contains is to parse the CMake build files. It is in these files that we can find build targets and files that are meant to be installed on the system. This is straightforward in the case of fictibot_controller, as seen in Listing 3.10 (see Listing A.10 for the full code). There is only one node, fictibot_controller, identified by the **add_executable** instruction.

```
add_executable(fictibot_controller
  src/controller_node.cpp
  src/random_controller.cpp
)
```

Listing 3.10: The fictibot_controller node, identified in the *CMakeLists.txt* file.

Still, adding to the challenge, not all CMake build targets are nodes. Some are simple libraries, others could be tools. In order to be certain, it is necessary to parse the files that compose the executables, and look for the ROS init function of the ROS client libraries (e.g., ros::init in C++ and rospy.init in Python). Sure enough, in the main function of the src/controller_node.cpp file we can see the ros::init call (Listing 3.11). The init function also provides the node's default ROS name, which is one of the attributes of the Node model.

```
1 int main(int argc, char **argv) {
2   ros::init(argc, argv, "fictibot_controller");
3   ros::NodeHandle n;
4   RandomController controller(n, 10 /*Hz*/);
5   ros::Rate loop_rate(10 /*Hz*/);
6   while (ros::ok()) {
7     controller.spin();
8     loop_rate.sleep();
9   }
10  return 0;
11 }
```

Listing 3.11: The `main` function of the `src/controller_node.cpp` file.

Note that searching for Nodes the other way around, i.e., starting with the parsing and look up of the `init` function in source code files, is not an optimal solution for a few reasons.

- The call to `ros::init` or `rospy.init` does not necessarily occur in the *main* file of the program, although it is often one of the first instructions of the `main` function.
- Identifying the file with this function call, in C++, might not suffice to identify all files that compose the node. One might be able to trace back header files, via **`#include`** directives, but other implementation files still require the full compile command (as given in CMake).
- Although unlikely, the same file with a call to `ros::init` or `rospy.init` might be reused to compile multiple (different) nodes, e.g., by retaining the same header files but changing the implementation files in C++. This could be used to produce similar nodes for different target hardware, or to distinguish between hardware and simulation nodes.

3.3.3 *ROS Primitive Calls*

When a node is identified, during the `extract_nodes_and_primitives` step, the set of source files that build it is also known, as seen with **`add_executable`**. These files must be parsed in order to extract the calls to ROS primitives, and this is the first true obstacle to static analysis. It is easy to understand that resolving all arguments of a primitive call to concrete values might be hard, or even impossible in some cases. But, sometimes, especially when layers of indirection come into play, detecting the primitives themselves could be the problem. For instance, calling functions dynamically (e.g., with function pointers) or using wrappers provided by libraries, such as the ROS `message_filters` or `dynamic_reconfigure` packages, adds complexity to the process. The latter is especially prominent in practice, as we can see in the evaluation chapter (Chapter 8).

Fortunately, as suggested by our empirical study [149] (see Section 2.3), most occurrences are rather simple to process, in terms of their arguments, at least, with the majority being literals. However, the ROS client libraries provide multiple overloads for the same primitives. In the C++ client library, for example, there are 3 overloads for **`advertise`** – which is one of the primitives with the fewest overloads. Thus, parsing primitive calls is not a matter of identifying one call of each type, but rather a roster of calls per

primitive. Ensuring that this catalogue is exhaustive (possibly including popular libraries as well) is key to ensuring the precision and recall of the extraction process.

When it is not possible to fully determine the argument ROS name (i.e., the topic, service or parameter name), the extraction process should still resolve as many parts of the name as possible. For instance, it might be the case that the namespace is known, but not the resource's own name, or vice-versa. Unresolved parts of the name are replaced with a wildcard `?`, in an effort to make the model as complete as possible, and enabling posterior analysis or user intervention to refine the names.

Another non-trivial detail of this step is how to deal with conditions. A simple implementation would only look at the control flow of the function containing the primitive call (as we did for the empirical study in Section 2.3), while a more thorough analysis would traverse all possible program paths that lead to the primitive call. Within the boundaries of static analysis, we strive for the latter. That is, if the current function body does not provide enough information (e.g., because a variable is a function parameter), we traverse the bodies of the functions that call the current function, and so on, effectively traversing as many program paths as possible, but in reverse order.

3.3.4 *Configurations and Resources*

After identifying all the source artefacts and calls to the ROS primitives, the reconstruction of the computation graph can begin (`extract_configurations`, Listing 3.12). As mentioned before, there is no way to know, for certain and in advance, which components are part of an application and which applications are part of a project. Thus, we resort to user input. At the bare minimum, a configuration specification provides the name of the configuration and the list of launch commands that compose the application. It is a list of commands, and not a set, because the order in which nodes are launched and parameters are set could be relevant (i.e., implicit dependencies between launch files).

```
1  def extract_configurations(config_specs, P, F, N):
2      C = []
3      for name, data in config_specs.items():
4          configuration = Configuration(name)
5          for item in data["launch_commands"]:
6              command = item["command"]
7              args = item["args"]
8              if command == "roslaunch":
9                  add_roslaunch(configuration, args, P, F, N)
10             elif command == "rosrun":
11                 add_rosrun(configuration, args, N)
12         hints = data.get("hints")
13         apply_hints(configuration, hints)
14         configuration.properties = data.get("properties", [])
15         C.append(configuration)
16     return C
```

Listing 3.12: Overview of the configuration construction algorithm.

To build a configuration, each launch command is interpreted in order. Launching individual nodes is relatively straightforward; the challenge resides on launch files. For each launch file, the algorithm has to mimic the behaviour of roslaunch, the utility that interprets launch files and deploys systems. That is, it must keep the semantics of all launch tags: **<remap>** tags must apply to the correct nodes; **<node>** tags are processed in the order they appear, but cannot make assumptions about which nodes are actually launched first; **<param>** tags must create parameters within the correct context (i.e., globally, or under a node's namespace); etc. This step can introduce various unresolved conditions and wildcards, due to conditional statements present in the launch files, or simply due to arguments provided via command line (manually, by the user), at launch.

Parsing launch files allows us to build all Node Instance and Parameter models, and from the pkg and type attributes of **<node>** tags we know which Node models correspond to each Node Instance. For instance, the launch/multiplexer.launch file of fictibot_controller, in Listing 3.13, contains 3 **<node>** tags, which correspond to 3 Node Instances. Only the ficticontrol node is affected by the **<remap>** tags.

```
1 <launch>
2   <node name="fictibase" pkg="fictibot_drivers" type="fictibot_driver" />
3   <node name="fictiplex" pkg="fictibot_multiplex" type="fictibot_multiplex" />
4   <node name="ficticontrol" pkg="fictibot_controller" type="fictibot_controller">
5     <remap from="controller_cmd" to="normal_priority_cmd" />
6     <remap from="/stop_cmd" to="normal_priority_stop" />
7   </node>
8 </launch>
```

Listing 3.13: The fictibot_controller launch/multiplexer.launch file.

The last fully automatic step is, for each Node Instance, to build ROS Links from the ROS primitive calls of the corresponding Node model. Links inherit most of their attributes directly from primitive calls, with the notable exception of the ROS resource name. The resource name is resolved using the name resolution rules of ROS, the name and namespace of the concrete Node Instance, as well as its remappings – with the exception of also allowing the wildcards introduced in the extraction process. Whenever the name resolution yields a name that is not yet present in the configuration, a new resource (Topic, Service or Parameter) is created and added to the graph. After creating a new resource, or retrieving an existing one with the same name, the new link is finally created between the node and the target resource. For instance, in Fictibot's Controller the primitive calls indicate topic advertisements for /controller_cmd and /stop_cmd (Listing 3.14), but, with the remappings of the previous launch file, the corresponding links would be created to /normal_priority_cmd and /normal_priority_stop instead.

```
1 RandomController::RandomController(ros::NodeHandle& n, double hz) {
2   /* ... */
3   command_publisher_ = n.advertise<std_msgs::Float64>("controller_cmd", 1);
4   stop_publisher_ = n.advertise<std_msgs::Empty>("/stop_cmd", 0);
5   /* ... */
6 }
```

Listing 3.14: Some of the topics used in the src/random_controller.cpp file of fictibot_controller.

3.3.5 *User-provided Hints*

In many cases, the model should be complete after the previous instantiation step. Depending on the precision and recall of the implementation, as well as the source code under analysis (which could have a large number of dynamic values), the resulting model could end up featuring many unresolved names. To accomodate for this, we extend the YAML configuration specifications, provided as user input, to also include optional *extraction hints* (`apply_hints` in Listing 3.12).

Hints are, in essence, partial models built by hand, with varying levels of detail. They are specified per configuration, and they follow the same YAML schema that defines our runtime metamodel entities, for the most part. We focus on runtime entities because source entities, such as packages and files, are straightforward to extract correctly. The problem lies mostly in primitive calls, as previously explained, which are used to instantiate links. Our current approach uses hints to fix the extracted model of the runtime entities that are the prime subject for analysis, although it could be extended to allow for more entities.

The major difference between extraction hints and the metamodel's YAML schema is that each entity includes a `create` flag. When set to true, the entity should include its required attributes, as per the schema; the entity is meant to be created and manually added to the extracted model, in all cases. Otherwise, the only required attribute is the ROS name, as the hint is meant to fix automatically extracted entities. ROS names are used to match with extracted entities, and if there is more than one possible match – due to duplicate entities in the graph, or wildcards in the name – we resort to traceability information to resolve conflicts. That is, the default behaviour is to match by ROS name, but, in case of ambiguity, we allow the use of the containing package and source file, down to the line and column number, if necessary. All remaining attributes are used to replace the extracted ones. We consider it an error to provide repair hints (as opposed to creation hints) for entities that do not exist, or if there are multiple possible matches (e.g., because traceability was not specified to disambiguate). Hints always overwrite extracted attributes; they are considered to be a ground truth.

As we have stated in the previous section, the runtime entities of the metamodel contain redundant information, namely Topics and Services. Thus, we limit hints to Node Instances, Parameters, and ROS Links – the essential information to build a Computation Graph. Listing 3.15 provides an example of what extraction hints for the `minimal` Configuration could look like in YAML syntax, if any hints were needed. In this example, we can see that a new Parameter `/new_param` is created (note how `create` is set to `true`). Furthermore, the model of the `/ficticontrol` Node Instance is refined regarding its Publisher and Parameter Getter Links. For the former, we just refine the `msg_type` of a Publisher on topic `/controller_cmd`, by setting it to `std_msgs/Float64`. For the latter, we create a new Link, connected to the Parameter we have just defined.

A more complex example can be seen, for instance, in the TurtleBot2 robot, one of our case studies in Chapter 8. Subscribers are created within a loop, reading topic names from a given list of strings. The automatic extraction is unable to resolve the ROS name argument because it is a dynamic value. We can

```yaml
configurations:
  minimal:
    # launch commands, etc.
    hints:
      nodes:
        /ficticontrol:
          publishers:
            - name: "/controller_cmd"
              type: "std_msgs/Float64"
          getters:
            - name: "/new_param"
              create: true
              type: "int"
              traceability:
                package: "fictibot_controller"
                file: "src/random_controller.cpp"
                line: 42
                column: 3
      parameters:
        /new_param:
          create: true
          type: "int"
          default_value: 1
          traceability: null
```

Listing 3.15: Example extraction hints in YAML syntax.

solve this by providing one repair hint that matches this primitive call, for the first entity created in the loop, and then provide the remaining entities as `create` hints.

Lastly, the application of extraction hints should follow a specific order:

1. application of extraction hints to Parameters;
2. application of extraction hints to Node Instances;
3. application of extraction hints to Links.

This approach aims to maximise the expressive power of hints, minimise name conflicts and make them more intuitive. We start with Parameters, as these are mostly independent entities, and in a ROS environment they would be set on the Parameter Server before launching nodes. Creating Parameters first also allows specified Links to use these Resources right away; the other way around would lead to Links implicitly creating new Parameters that would have to be matched against the hints. We leave the refinement and creation of Links for last because the creation of new Node Instances is again based on a Node, and on the instantiation of ROS Primitive Calls to create the initial set of Links. The fresh Links will inherit any unresolved values from the parent Primitive Calls, which can then be resolved by applying hints.

In summary, we propose a simple approach where hints are regarded as the ground truth, even though they might not describe every aspect of the model. This ground truth is used to extend the model or to refine it, entity by entity. Every part of the ground truth must be present in the final model, as mentioned. An interesting consequence of this approach is that it enables users to fully specify (the runtime entities of) a system from scratch via hints, even when the extraction process produces an empty computation

graph. This is useful, for instance, to overcome the limitations of an implementation, such as dealing with unsupported programming languages – e.g., supporting C++ but not Python, and providing all Python nodes via hints. In a sense, it plays nicely with both the traditional code-first development processes, or the more formal model-driven ones.

3.4 Summary

A domain-specific ROS analysis requires, in some form, a model of the target ROS system based around the core concepts of ROS, such as nodes, topics and services. Conveniently, the ROS documentation already lays out an abstraction for the system's runtime: the ROS Computation Graph. However, the main purpose of finding defects is to correct them, and the entities of the computation graph, by themselves, do not point at the origin of those defects, in the source code. *Traceability* is a key ingredient that any ROS architectural model should contain. The main problem is that building such a complex model by hand, in an environment that encourages dynamic architectures, entails a considerable volume of error-prone work and volatile data. Thus, automatic model extraction (or reverse engineering) becomes more of a necessity rather than a mere convenience.

In this chapter, we presented the state of the art in architectural modelling and architecture extraction, followed by a metamodel for ROS architectures and an automatic extraction procedure based on static analysis. The proposed metamodel retains traceability information and allows for a dual view of the system, from a file system perspective and from a runtime perspective, relying mostly on concepts that are already staples of ROS. This contribution, already published in [151], improves the comprehension of ROS systems and enables reasoning about their design at static time. Achieving the same, manually, through source code inspection would be infeasible for large systems under development. The proposed extraction procedure, being based on static analysis, is, in general, incomplete, although sound. We acknowledge and address this problem by introducing uncertainty points in the metamodel, as well as allowing the external specification of extraction hints in order to refine extracted models by resolving some of those uncertainties. An evaluation of the proposed extraction process is done in Chapter 8.

4

BEHAVIOURAL PROPERTY SPECIFICATION

The previous chapter was dedicated to the definition of an architectural model for ROS systems and the corresponding automated extraction process from source artefacts. It is evident that the metamodel is mostly concerned with the structure of the system. In this chapter we propose a language specially tailored for the specification of behavioural properties, both for individual nodes and full applications. Depending on the intended analysis, such properties can be taken as assumptions about the system, or as verification goals. This chapter presents the syntax and semantics for this behavioural property specification language.

4.1 State of the Art

Specifying system behaviour inevitably comes down to describing observable actions, outputs, and how they relate to each other (e.g., causality) or when they should manifest. There are various ways to accomplish it, but temporal logics are especially fit for this purpose.

Temporal Logics Temporal logics, as implied by the name, must provide an abstraction for time. Two of the most common approaches to this problem are Linear Temporal Logic (LTL) [140] and Computation Tree Logic (CTL) [49]. The former views time, and the future in particular, as pre-determined, i.e., a program execution is treated as a single trace of events, over which one encodes formulae. The latter, by contrast, is a branching-time logic, meaning that computations are structured as a tree of possible paths that a program can follow. This view is more closely related to the control flow of programs.

Both LTL and CTL have been extensively used in program verification, with applications in model checking being the most prominent. As to which is the superior choice, there is no clear answer and there are arguments in favour to both sides [165]. Both have limitations in their expressiveness; some properties can be expressed with one, but not the other. The tree nature of CTL enables efficient verification algorithms that are linear in the size of the specification, as opposed to LTL which is exponential. This is one of the reasons for CTL to be the backbone of several model checkers, and for its extensive use in industry. However, some important properties, such as strong fairness, can be specified in LTL, but cannot be specified in CTL. In addition, the branching aspect of CTL, which adds a layer of complexity to the language, is rarely a necessity for most specifications. Overall, not only is LTL more intuitive, it is also amenable to semi-formal verification [66] – verification techniques that, while not exhaustive, are effective

at finding defects by focusing on falsification. As in [165], we argue in favour of LTL and its extensions for the specification of robotic behaviour.

Standard LTL often considers only future modalities, i.e., properties are expressed in terms of the present and future time instants only. Two common examples are the $\Box$ (always) and the $\Diamond$ (eventually) operators. For a formula φ, the formula $\Box\varphi$ states that φ must always hold, from the current instant onward. Similarly, the formula $\Diamond\varphi$ states that φ must hold at the current instant or at some point in the future, at least once.

A common variant of LTL is the Linear Temporal Logic with past operators. This variant adds past-time modalities to the logic that allow specifications to refer to past states or events, besides the traditional future ones. For instance, the $\blacksquare$ (historically) and $\blacklozenge$ (once) operators are often introduced as the past equivalents of the $\Box$ and $\Diamond$ operators, respectively. Including past modalities does not make the logic any more expressive, however [71], and there are ways to convert a past-time or mixed formula to pure-future LTL [70]. The main benefits of introducing past-time operators are the syntactic reduction of some formulae (up to an exponential factor [117]) and a significant improvement regarding formulae comprehension. A common (yet simple) example to show its convenience is the property *"ψ should always be preceded by φ"*. This is expressed in pure LTL using the *until* ($\mathcal{U}$) operator.

$$\neg((\neg\varphi)\ \mathcal{U}\ (\psi \wedge \neg\varphi))$$

The formula uses a double negative to state that *"it is not true that φ never holds until the time instant when ψ holds"*. With past modalities enabled, this property is reduced to the much more intelligible *"it is always the case that if ψ holds at the current instant, φ held at least once in the past"*, written as follows.

$$\Box(\psi \rightarrow \blacklozenge\varphi)$$

Over the remainder of this chapter we will refer to the standard, Kripke-style syntax and semantics of LTL operators, both in future and past modalities. Starting with the syntax, let Σ be a countable set of atomic propositions.

Definition 1. *LTL formulae over Σ are inductively defined: (i) For $p \in \Sigma$, p is a formula. (ii) For φ and ψ formulae, $(\neg\varphi)$, $(\varphi \wedge \psi)$ and $(\varphi \vee \psi)$ are formulae. (iii) For φ and ψ formulae, $(\bigcirc\varphi)$, $(\bullet\varphi)$, $(\Diamond\varphi)$, $(\blacklozenge\varphi)$, $(\Box\varphi)$, $(\blacksquare\varphi)$, $(\varphi\mathcal{U}\psi)$, and $(\varphi\mathcal{S}\psi)$ are formulae.*

As for the semantics, formulae are evaluated over *temporal frames* $(\mathbb{N}, \leq)$ – a pair of a set ($\mathbb{N}$ in this case) and a binary relation on that set ($\leq$), called precedence. The precedence relation is a *total order*, for $a, b, c \in \mathbb{N}$:

- it is antisymmetric – if $a \leq b$ and $b \leq a$, then $a = b$;
- it is transitive – if $a \leq b$ and $b \leq c$, then $a \leq c$;
- it is connex – it is true that $a \leq b$ or $b \leq a$.

Let $\mathcal{M} = (\mathbb{N}, \leq, v)$ be a temporal model, where $(\mathbb{N}, \leq)$ is a temporal frame, and $v : \mathbb{N} \times \Sigma \mapsto \{\top, \bot\}$ is a valuation assigning a truth value to each atomic proposition $p \in \Sigma$ at a given time instant $i \in \mathbb{N}$.

The temporal model $\mathcal{M}$ is said to satisfy a formula φ, written $\mathcal{M} \models \varphi$, if and only if $\mathcal{M}, 0 \models \varphi$. Given an atomic proposition p, time instants i, j and k, and formulae φ and ψ, the semantics of LTL operators is presented in Figure 30.

$$
\begin{array}{lll}
\mathcal{M}, i \models \top & & \\
\mathcal{M}, i \not\models \bot & & \\
\mathcal{M}, i \models p & \text{iff} & v(i,p) \\
\mathcal{M}, i \models \neg\varphi & \text{iff} & \mathcal{M}, i \not\models \varphi \\
\mathcal{M}, i \models \varphi \wedge \psi & \text{iff} & \mathcal{M}, i \models \varphi \text{ and } \mathcal{M}, i \models \psi \\
\mathcal{M}, i \models \varphi \vee \psi & \text{iff} & \mathcal{M}, i \models \varphi \text{ or } \mathcal{M}, i \models \psi \\
\mathcal{M}, i \models \circ\varphi & \text{iff} & \mathcal{M}, i+1 \models \varphi \\
\mathcal{M}, i \models \bullet\varphi & \text{iff} & i > 1 \text{ and } \mathcal{M}, i-1 \models \varphi \\
\mathcal{M}, i \models \Diamond\varphi & \text{iff} & \mathcal{M}, j \models \varphi \text{ for some } j \text{ such that } i \leq j \\
\mathcal{M}, i \models \blacklozenge\varphi & \text{iff} & \mathcal{M}, j \models \varphi \text{ for some } j \text{ such that } j \leq i \\
\mathcal{M}, i \models \Box\varphi & \text{iff} & \mathcal{M}, j \models \varphi \text{ for all } j \text{ such that } i \leq j \\
\mathcal{M}, i \models \blacksquare\varphi & \text{iff} & \mathcal{M}, j \models \varphi \text{ for all } j \text{ such that } j \leq i \\
\mathcal{M}, i \models \varphi \mathcal{U} \psi & \text{iff} & \mathcal{M}, j \models \psi \text{ for some } j \text{ such that } i \leq j \\
 & & \quad \text{and } \mathcal{M}, k \models \varphi \text{ for every } k \text{ such that } i \leq k < j \\
\mathcal{M}, i \models \varphi \mathcal{S} \psi & \text{iff} & \mathcal{M}, j \models \psi \text{ for some } j \text{ such that } j \leq i \\
 & & \quad \text{and } \mathcal{M}, k \models \varphi \text{ for every } k \text{ such that } j < k \leq i
\end{array}
$$

Figure 30: Semantics of LTL operators.

Specification Based on LTL Linear Temporal Logic has been used extensively for property specification with applications in model checking [17, 50, 83, 104], runtime verification [21, 23, 32, 106, 126] or testing [125, 138, 161]. But, even though LTL is considered to be more intuitive than CTL, specification mistakes are bound to happen at some point. To overcome this threat, and, in a way, to lower the barrier to adoption of automated verification, Dwyer et al. proposed a catalogue of patterns for specification problems [58, 59] as follows.

1. **Absence:** φ is always false.
2. **Universality:** φ is always true.
3. **Existence:** φ is true at least once.
4. **Bounded Existence:** φ is true at most n times.
5. **Precedence:** φ is a necessary precondition for ψ.
6. **Response:** φ must be followed by an occurrence of ψ.
7. **Precedence Chain:** Generalisation of *Precedence* to m causes and n effects.
8. **Response Chain:** Generalisation of *Response* to m stimuli and n responses.

Such patterns were put together based on experience, and represent properties that are often desirable or required in various concurrent or reactive systems. Each pattern can be further refined with a *scope*, limiting the extent to which the property must hold. The scope can be **global** (the entire execution), **before** φ (the whole execution up to φ), **after** φ (the whole execution starting from φ), **between** φ and ψ (where

both events must occur), or **after** φ **until** ψ (similar to **between**, but ψ is not required to occur, in which case the property must hold forever after the last occurrence of φ). Mappings of these patterns to LTL, CTL and other logics have been proposed[1] [58, 59], facilitating their adoption.

Bauer and Leucker expand the idea of specification patterns further with SALT [22], a specification and assertion language that incorporates the previous patterns and extends them with notions of real time, exceptions to normal behaviour and regular expressions. In terms of expressiveness, the untimed fragment of SALT is as expressive as LTL, while its timed fragment is as expressive as Timed LTL, a temporal logic based on automata with clocks (also known as *state-clock logic* [56, 146]). Another improvement over Dwyer et al.'s patterns is the generalisation of scopes. In SALT, scopes can be nested, adding a layer of complexity in exchange for increased flexibility and specificity of the specifications.

Extensions to Linear Temporal Logic Two major limitations of LTL for the purposes of specifications are its inability to express properties in terms of real time, and its lack of first-order predicates. Metric Temporal Logic (MTL [97]) addresses the former, and First-Order Temporal Logic (FOTL [116]) was proposed to address the latter. Despite being useful extensions on their own, especially when including past-time modalities, complex systems (such as robots) inevitably require the full set of features. Metric First-Order Temporal Logic (MFOTL [46]) is a logic that incorporates the benefits of both MTL and FOTL. Given its relevance for the specification language presented in this chapter, herein we present the syntax and semantics of MFOTL.

Metric First-Order Temporal Logic Temporal operators in MFOTL can be annotated with (metric time) intervals. A temporal formula is only satisfied if it is satisfied within the bounds given by the time interval of the temporal operator, which is always relative to a time stamp.

Let Δ be the set of non-empty intervals over $\mathbb{N}_0$. We write an interval $\delta \in \Delta$ as $[\delta_1, \delta_2)$, where $\delta_1 \in \mathbb{N}_0$, $\delta_2 \in \mathbb{N} \cup \{\infty\}$, and $\delta_1 < \delta_2$, i.e., $[\delta_1, \delta_2) := \{t \in \mathbb{N}_0 : \delta_1 \leq t < \delta_2\}$. A *signature* S is a tuple $(\mathsf{C}, \mathsf{P}, a)$, where C is a finite set of constant symbols, P is a finite set of predicates disjoint from C, and the function $a : \mathsf{P} \mapsto \mathbb{N}$ provides the arity $a(p) \in \mathbb{N}$ of each predicate $p \in \mathsf{P}$. Also, let V denote a countably infinite set of variables, where we assume that $\mathsf{V} \cap (\mathsf{C} \cup \mathsf{P}) = \emptyset$, for all signatures.

Definition 2. *The formulae over S are inductively defined: (i) For $x, y \in \mathsf{V} \cup \mathsf{C}$, $x = y$ is a formula. (ii) For $p \in \mathsf{P}$, $n = a(p)$ and $x_1, \dots, x_n \in \mathsf{V} \cup \mathsf{C}$, $p(x_1, \dots, x_n)$ is a formula. (iii) For $x \in \mathsf{V}$, if φ and ψ are formulae, then $(\neg\varphi)$, $(\varphi \wedge \psi)$ and $(\exists x : \varphi)$ are formulae. (iv) For $\delta \in \Delta$, if φ and ψ are formulae, then $(\bullet_\delta \varphi)$, $(\circ_\delta \varphi)$, $(\varphi \mathcal{S}_\delta \psi)$, and $(\varphi \mathcal{U}_\delta \psi)$ are formulae.*

We provide only the syntax and semantics for a core set of MFOTL operators (Figure 31). Other common operators can be defined in terms of the core set, as shown in Figure 32. To fully define the semantics of MFOTL, a few additional notions are required. A *first-order structure* D over S consists of a domain $|D| \neq \emptyset$ and interpretations $c^D \in |D|$ and $p^D \subseteq |D|^{a(p)}$, for each $c \in \mathsf{C}$ and $p \in \mathsf{P}$. A *temporal*

1 https://matthewbdwyer.github.io/psp/patterns.html

first-order structure over S is a pair (D, τ), where $D = (D_0, D_1, \ldots)$ is a sequence of structures over S and $\tau = (\tau_0, \tau_1, \ldots)$ is a sequence of natural numbers (time stamps), where:

1. τ is monotonically increasing and makes progress, i.e.,
   $\forall i \geq 0 : \tau_i \leq \tau_{i+1}$ and $\exists j > i : \tau_j > \tau_i$;
2. D has constant domains, i.e., $\forall i \geq 0 : |D_i| = |D_{i+1}|$;
3. each constant symbol $c \in \mathsf{C}$ has a rigid interpretation, i.e., $\forall i \geq 0 : c^{D_i} = c^{D_{i+1}}$.

A *valuation* is a mapping $v : \mathsf{V} \mapsto |D|$. We abuse notation by applying valuations also to constant symbols $c \in \mathsf{C}$, with $v(c) = c^D$. Additionally, we denote $v[x \mapsto d]$ as the valuation where every mapping is unaltered, when compared to v, save for $x \in \mathsf{V}$, which should map to $d \in |D|$.

Definition 3. *Let (D, τ) be a temporal structure over S, with $D = (D_0, D_1, \ldots)$ and $\tau = (\tau_0, \tau_1, \ldots)$, φ and ψ formulae over S, v a valuation, and $i \in \mathbb{N}_0$. The temporal structure satisfies a formula φ, written $(D, \tau) \models \varphi$, if and only if $(D, \tau, v, 0) \models \varphi$. We define $(D, \tau, v, i) \models \varphi$ as shown in Figure 31.*

$$
\begin{array}{lll}
(D,\tau,v,i) \models x = y & \text{iff} & v(x) = v(y) \\
(D,\tau,v,i) \models p(x_1,\ldots,x_{a(p)}) & \text{iff} & (v(x_1),\ldots,v(x_{a(p)})) \in p^{D_i} \\
(D,\tau,v,i) \models (\neg\varphi) & \text{iff} & (D,\tau,v,i) \not\models \varphi \\
(D,\tau,v,i) \models (\varphi \wedge \psi) & \text{iff} & (D,\tau,v,i) \models \varphi \text{ and } (D,\tau,v,i) \models \psi \\
(D,\tau,v,i) \models (\exists x : \varphi) & \text{iff} & (D,\tau,v[x \mapsto d],i) \models \varphi, \text{ for some } d \in |D| \\
(D,\tau,v,i) \models (\bullet_\delta \varphi) & \text{iff} & i > 0, \tau_i - \tau_{i-1} \in \delta, \text{ and } (D,\tau,v,i-1) \models \varphi \\
(D,\tau,v,i) \models (\circ_\delta \varphi) & \text{iff} & \tau_{i+1} - \tau_i \in \delta \text{ and } (D,\tau,v,i+1) \models \varphi \\
(D,\tau,v,i) \models (\varphi\ \mathcal{S}_\delta\ \psi) & \text{iff} & \text{for some } j \leq i,\ \tau_i - \tau_j \in \delta,\ (D,\tau,v,j) \models \psi \\
 & & \text{and } (D,\tau,v,k) \models \varphi, \text{ for all } k \in [j+1, i] \\
(D,\tau,v,i) \models (\varphi\ \mathcal{U}_\delta\ \psi) & \text{iff} & \text{for some } j \geq i,\ \tau_j - \tau_i \in \delta,\ (D,\tau,v,j) \models \psi \\
 & & \text{and } (D,\tau,v,k) \models \varphi, \text{ for all } k \in [i, j)
\end{array}
$$

Figure 31: Semantics of MFOTL.

Throughout the remainder of this thesis we use standard syntactic sugar such as the omission of parentheses, infix binary predicates (e.g., $x < y$) and the standard temporal operators presented previously for LTL. In Figure 32 we show how these operators can be defined in terms of the core set of operators for MFOTL. The same Figure also shows the *weak-until* operator ($\mathcal{W}$), the *back-to* operator ($\mathcal{B}$), and the *weak-previous* operator (sometimes referred to as *weak-yesterday*, $\mathcal{Z}$), that we use to shorten some formulae. Note that the non-metric variants of the temporal operators (as in LTL) are easily defined by considering the interval to be $[0, \infty)$. For instance, $\Box\varphi := \Box_{[0,\infty)}\varphi$.

4.2 Language Overview

4.2.1 *Context*

In publisher-subscriber architectures (especially), where there are many-to-many communications, it is common to implement components in such a way that they are independent to some degree – i.e., nodes

$$
\begin{aligned}
(\blacksquare_\delta \varphi) &\equiv (\neg(\top \,\mathcal{S}_\delta\, \neg\varphi)) \\
(\Box_\delta \varphi) &\equiv (\neg(\top \,\mathcal{U}_\delta\, \neg\varphi)) \\
(\blacklozenge_\delta \varphi) &\equiv (\top \,\mathcal{S}_\delta\, \varphi) \\
(\Diamond_\delta \varphi) &\equiv (\top \,\mathcal{U}_\delta\, \varphi) \\
(\varphi \,\mathcal{W}_\delta\, \psi) &\equiv (\Box_\delta \varphi) \lor (\varphi \,\mathcal{U}_\delta\, \psi) \\
(\varphi \,\mathcal{B}_\delta\, \psi) &\equiv (\blacksquare_\delta \varphi) \lor (\varphi \,\mathcal{S}_\delta\, \psi) \\
(\mathcal{Z}_\delta\, \varphi) &\equiv (\bullet_\delta \varphi) \lor (\neg(\bullet_\delta \top))
\end{aligned}
$$

Figure 32: Common operators of MFOTL, defined in terms of the core set of operators.

should make no assumptions about other nodes in the network. Ideally, a node behaves correctly with zero, one, or multiple other components connected to it, for some definition of correctness. The number of peers should be inconsequential. All application logic should revolve around the exchanged messages and the data they carry, since these are the main observable actions that nodes take (save for direct interactions with hardware). This *decoupled* approach has a few significant advantages:

1. it allows us to treat nodes as black boxes with a *contract* over their subscriptions (inputs) and publications (outputs);
2. it makes components more robust;
3. it improves and promotes reusability and reconfiguration;
4. it lends itself naturally to event-based reasoning using temporal logics.

It follows, then, that to write down reusable and easily maintainable specifications about a component, group of components, or even the system in general, embracing this black-box, event-based, message-centric reasoning is a straightforward option. Properties should focus on the *messages*, their contents and the (causal and temporal) relations between them, not on the inner workings of nodes. And we can see that these concepts apply naturally to the message-passing nature of ROS. For instance, an informal property for a mobile robot, in the context of ROS, could state that *"whenever a message is observed on topic `/bumper` such that the robot's bumper is pressed, a message on topic `/stop` should follow, within 100 milliseconds of the first"*. Considering this example, and how messages work in ROS, a specification language for ROS systems must provide a few basic features, namely:

- references to individual resource names (e.g., topics);
- references to message contents (e.g., observing the contents of the `/bumper` message);
- temporal operators and relations (e.g., the message on `/stop` is observed after [and because of] the message on `/bumper`);
- real-time behaviour specifications (e.g., observing a message at most 100 milliseconds after observing another).

From Section 2.3, we know that the publisher-subscriber paradigm is the prevalent means of communication in ROS. For simplicity's sake, our proposal for a specification language will focus only on this type of observable behaviour. Services and actions would be relatively straightforward to include in the language, but are left as future work.

4.2.2 *Concept*

One of our goals for this specification language is to propose a syntax and semantics that are both minimalistic and domain specific, yet readable, easily extensible and similar to existing languages – a common set of desirable qualities for a specification language [124]. Such traits improve the ease of use, lower the barrier to adoption and alleviate the burden of specification in general. Another way to make specification more straightforward and less error prone is to restrict properties to a set of well-known patterns.

The property specification patterns proposed by Dwyer et al. [58, 59] have been widely adopted, due to them capturing a large number of interesting real-world scenarios. Their approach of dividing a property into a combination of predefined scopes and patterns aids in streamlining the specification process, without sacrificing much in terms of expressive power. In our experience, most properties gravitate towards an even smaller number of patterns, namely the **Absence**, **Existence**, **Precedence** and **Response** patterns. We settled on these patterns to form the core of our language. All proposed scopes seem useful, but we decided to drop the strict **before** and **between** scopes, and to retain only their weaker variants, **after-until** and **until** (the latter not in the original proposal). The main difference is that the **until** scopes do not require the termination event to happen, which is, in general, a more flexible approach to specification that is able to handle, e.g., component crashes.

Despite their proven usefulness, Dwyer's specification patterns fail to address some crucial issues in the safety-critical context of cyber-physical systems. One such issue, as previously illustrated, is that real time – and, by extension, system performance – cannot be left out of the specification. The abstract nature of time in LTL or CTL is insufficient to capture correct robot behaviour. In this context, it is not hard to see how responding to stimuli within 100 milliseconds or within 1 second could make the difference between success and disaster, even though both occur *eventually*. Thus the need to move away from LTL and other similar logics, into logics that handle real time, such as Metric Temporal Logic. This is a small extension in terms of syntax, but a significant step up in complexity in terms of semantics.

Another shortcoming of the original specification patterns, being designed for LTL and similar logics, is that they apply to atomic propositions. Again, in a system as complex as ROS, atomic propositions are not expressive enough to encode the intended observable behaviour. In this case, we define observable behaviour in terms of the messages that nodes exchange with one another, and, as previously mentioned, we must be able to reference message contents. Recall that a ROS message is allowed to: *(i)* have an arbitrary number of data fields; *(ii)* compose messages arbitrarily (i.e., use other messages as fields); and *(iii)* contain lists of values of arbitrary length. Facing such complex data structures, first-order predicates become a necessity. Without first-order predicates, it would not be possible to encode properties such as *"stop when one of the laser readings detects an obstacle within a range of 1 meter"*, or that a sequence of messages is supposed to contain monotonically increasing values. Combining this requirement with the previous requirement of handling real time, we settled on Metric First-Order Temporal Logic as the base for the language semantics.

We have discussed the importance of building the language around exchanged messages as the main events of interest. Scopes and patterns are specified in terms of events – e.g., *"after receiving a message such that φ"*, or *"there are no messages such that φ"*. In its simplest form, an event simply requires a topic name, under which messages should be observed. Any message passing through the topic should match the event. Optionally, an event may be refined with a predicate over the message's data fields, which will act as a sort of filter; only those messages that satisfy the predicate will match the event. This is how we impose, e.g., ranges of expected values for a certain field. Finally, predicates may include references to previous messages in the same timeline (depending on the property's scope and pattern). This way, we have the necessary tools to establish relations between messages and message fields, such as equality and monotonicity.

In short, even though we decided not to cover the original collection of patterns in full, we address some of the features that are lacking in Dwyer's proposal, such as real-time constraints, data parametrisation (i.e., predicates over messages) and cross-references between events. Properties are built from scopes and patterns, which in turn are specified in terms of observed events and time intervals. Putting all concepts together, we get the hierarchichal structure depicted in Figure 33 (where dashed lines represent optional elements).

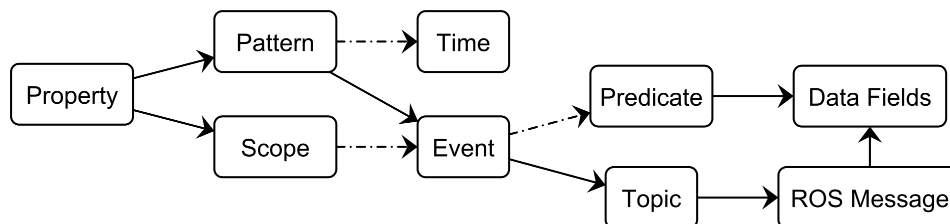


Figure 33: Structure of a property in the proposed language.

4.3 Language Syntax

The previous section, and Figure 33 in particular, show that the two major components of a property are its scope and its pattern. These are specified in terms of events, which, at their simplest, require a ROS name and (possibly) a predicate over message fields. Below we can see the top-level grammar production rule for properties, in Extended Backus-Naur form.

$\langle property \rangle$ ::= $\langle scope \rangle$ '**:**' $\langle pattern \rangle$

The remainder of this section goes into the details of the syntax for scopes, patterns, events and predicates.

4.3.1 *Scopes*

Scopes are a way to represent time windows related to certain events or system states. Besides the **global** scope, given by the keyword **`globally`**, which spans the whole system execution, we consider only

a general event-based scope, which we call **after-until**, and its two simpler variants, **after** and **until**. The grammar rules for scopes are as follows.

⟨*scope*⟩ ::= `globally` | ⟨*after-until*⟩ | ⟨*until*⟩

⟨*after-until*⟩ ::= `after` ⟨*event*⟩ [⟨*until*⟩]

⟨*until*⟩ ::= `until` ⟨*event*⟩

The ⟨*after-until*⟩ scope is analogous to its homonym, as proposed by Dwyer et al.; it denotes the window between an initial event (that we call the *activator*) and a second event (that we call the *terminator*). It can be reduced to a variant that only specifies an activator event, mirroring the *after* scope by Dwyer et al., or to a variant that only specifies the terminator event (the *until* scope).

4.3.2 *Patterns*

As previously mentioned, we build the specification language around four property patterns proposed by Dwyer et al.: **Absence**, **Existence**, **Precedence** and **Response**. Absence properties, as implied by the pattern name, forbid the occurrence of an event at any time within the scope, and are always expressed in the negative form with the keyword `no`. The Existence pattern, given by the keyword `some`, states that a certain message will be published at least once, at some point in the future (but always within the scope). The Precedence pattern is the first of the binary patterns, in the sense that it is specified not in terms of one event (as is the case with Absence and Existence), but in terms of two distinct events. This pattern is given by the keyword `requires`, stating that one event *requires* another, i.e., the first event in the specification is preceded by the second. The Response pattern is another binary pattern – the dual of the Precedence pattern. It is used to specify that one event is the cause for another, i.e., an occurrence of the *stimulus* implies an eventual occurrence of the *response*. In addition to these, we define a new pattern, called **Prevention**, which is a mix between the Absence and Response patterns. Given by the `forbids` keyword, it states that a stimulus event leads to the absence of another event. Its dual would state that an event is preceeded by the absence of another event (e.g., `b` forbids a previous `a`), but note that this is equivalent to saying that, if the first event happens, then the second must not happen (e.g., `a forbids b`). Thus, introducing a new keyword is not necessary. Every pattern can include an optional timeout, as seen in the following grammar.

⟨*pattern*⟩ ::= ⟨*unary-pattern*⟩ ⟨*event*⟩ [⟨*time-bound*⟩]
| ⟨*event*⟩ ⟨*binary-pattern*⟩ ⟨*event*⟩ [⟨*time-bound*⟩]

⟨*unary-pattern*⟩ ::= `no` | `some`

⟨*binary-pattern*⟩ ::= `requires` | `causes` | `forbids`

⟨*time-bound*⟩ ::= `within` ⟨*time*⟩

⟨*time*⟩ ::= ⟨*non-negative-integer*⟩ `ms`

4.3.3 *Events and Predicates*

Events, as previously mentioned, are specifications for ROS messages. Including operators for more complex events, such as event sequences, would allow us to incorporate the missing patterns from Dwyer et al.'s catalogue, namely, the bounded variants for the Existence, Precedence and Response patterns, but we leave them for future work. Event predicates include only a core set of operators, for illustrative purposes.

⟨*event*⟩ ::= ⟨*ros-name*⟩ [⟨*alias*⟩] [⟨*predicate*⟩]

⟨*alias*⟩ ::= ‘as’ ⟨*identifier*⟩

⟨*predicate*⟩ ::= ‘{’ ⟨*condition*⟩ ‘}’

⟨*condition*⟩ ::= ⟨*value*⟩ ⟨*binary-rel-operator*⟩ ⟨*value*⟩
| ‘(’ ⟨*condition*⟩ ‘)’
| ‘not’ ⟨*condition*⟩
| ‘forall’ ⟨*identifier*⟩ ‘in’ ⟨*msg-field*⟩ ‘:’ ⟨*condition*⟩
| ⟨*condition*⟩ ⟨*binary-connective*⟩ ⟨*condition*⟩

⟨*binary-rel-operator*⟩ ::= ‘=’ | ‘<’

⟨*binary-connective*⟩ ::= ‘or’ | ‘and’

⟨*value*⟩ ::= ⟨*boolean*⟩ | ⟨*num-expr*⟩ | ⟨*string*⟩

⟨*variable*⟩ ::= ‘$’ ⟨*identifier*⟩

⟨*msg-field*⟩ ::= ⟨*field-name*⟩ {‘.’ ⟨*field-name*⟩}
| ⟨*alias-ref*⟩ ‘.’ ⟨*field-name*⟩ {‘.’ ⟨*field-name*⟩}

⟨*alias-ref*⟩ ::= ‘@’ (⟨*non-negative-integer*⟩ | ⟨*identifier*⟩)

⟨*field-name*⟩ ::= ⟨*identifier*⟩ [⟨*index*⟩]

⟨*index*⟩ ::= ‘[’ (⟨*non-negative-integer*⟩ | ⟨*variable*⟩) ‘]’

⟨*num-expr*⟩ ::= ⟨*number*⟩
| ⟨*msg-field*⟩
| ⟨*variable*⟩
| ‘(’ ⟨*num-expr*⟩ ‘)’
| ‘-’ ⟨*num-expr*⟩
| ⟨*num-expr*⟩ ⟨*binary-num-operator*⟩ ⟨*num-expr*⟩

⟨*binary-num-operator*⟩ ::= ‘+’ | ‘*’

We can see that the proposed operators provide some expressiveness, with logical connectives (**not**, **and**, **or**, **forall**), relational operators (‘=’, ‘<’) and arithmetic operators. While we present only a small set

of operators, we can assume a number of syntactic sugar operators, such as the relational operator '>' or the division operator '/'.

With the `forall` quantifier, we provide a basic means of iterating over array fields. This is intended to iterate over all valid *indices* of the array, rather than the values directly. For instance, the syntax '`forall i in array`' is roughly equivalent to the Python `for i in range(len(array))`.

The grammar also captures references to quantified variables ('`$var`'), and to message fields (either of the message associated to the event, or to other messages with the '`@`' symbol). Message fields can be simple identifiers to refer directly to a field of the current message ('`field`'), multiple identifiers separated by dots to refer to nested fields of composed messages ('`parent.child`'), or indexed to access array positions ('`parent.array[1]`'). Regarding cross-references to other messages, every message has an implicit associated index to be used with '`@`'. Messages are numbered, starting at 1 with the activator event, and then with 2 for the first pattern event. Other events are not referenced in the current version of the language. Since numeric references can become confusing, we introduce *event aliases*, a syntactic sugar layer that replaces numeric references to other messages with human-readable names instead. For instance, an activator event '`/bumper as Bumper`' would allow predicates to refer to this message as '`@Bumper`' rather than '`@1`'. Message fields would be accessed as '`@Bumper.field`', for instance. We reiterate that this is just a syntactic replacement, and thus has no effect over the language's semantics. We also note that not all events can refer to each other arbitrarily, for temporal and semantic reasons; an event can only refer to another event that, unambiguously, happened before it. For example, activator and terminator events must be independent, but pattern events can refer to the scope activator. The semantic limitations that require independent terminator events are explained in Section 4.4.

4.3.4 *Examples*

This subsection provides some syntax examples for Fictibot properties. We start off with an example of the Absence pattern with a global scope. Consider the following property for the Fictibot Driver.

> At all times, all `data` published on `/bumper` is within the range $[0, 7]$.

The property has to be encoded in the negative form; stating that only the values between 0 and 7 are allowed is the same as stating that no other value is allowed. As such, the property becomes:

```
1 globally: no /bumper {data < 0 or data > 7}
```

Examples of the Existence pattern assert that a certain message shall be published. We can use it, for instance, to specify some properties of system initialisation.

> Within the first 500 milliseconds after launching, a message shall be published on `/bumper`.

This property does not impose any restriction on the message itself, thus it contains no predicates, but it does use the timed variant of the pattern.

```
globally: some /bumper within 500 ms
```

The Fictibot Multiplexer is, in essence a state machine. It provides the ideal setting to use the After-Until scope, to model the node's states. Combined with the Precedence pattern, we can assert properties such as the following.

> While in a high priority state (messages on `/state` carry `data` > 0), publishing a message on `/controller_cmd` is preceded by receiving an exactly equal message on `/high_priority_-cmd`, in the preceding 100 milliseconds.

```
after /state {data > 0} until /state {not data > 0}: /controller_cmd as CMD
    requires /high_priority_cmd {data = @CMD.data} within 100 ms
```

The Response pattern, conversely to the Precedence pattern, specifies an event that should happen after the stimulus event. It is adequate to specify reactive systems, such as safety controllers and some aspects of the Fictibot Random Controller.

> At all times, receiving a `/bumper` message such that `data` $\neq 0$ leads to the publication of a message on `/normal_priority_stop`, within, at most, 200 milliseconds.

The syntax for this property is as follows:

```
globally: /bumper {not data = 0} causes /normal_priority_stop within 200 ms
```

Lastly, an example of the Prevention pattern in this system would be the time-based nature of the Fictibot Multiplexer's states.

> After entering a high priority state, the Multiplexer should remain in that state for, at least, 1 second.

```
globally: /state {data > 0} forbids /state {not data > 0} within 1000 ms
```

4.4 Language Semantics

We define the semantics of our specification language in terms of the semantics of MFOTL. That is, given a syntactic construct of the language Φ, we define its *interpretation* as the equivalent MFOTL formula and denote it as $[\![\Phi]\!]$. We require only a few additional definitions beforehand.

Let $\mathbb{M}$ be the set of all *messages*, where a message consists of a unique identifier. The first-order structures D_i over S consist of a domain $|D|$ such that

- $|D|$ contains all numbers, i.e., $\mathbb{R} \subset |D|$;
- $|D|$ contains all Boolean values, i.e., $\{\top, \bot\} \subset |D|$;
- $|D|$ contains all strings, i.e., $\mathbb{S} \subset |D|$, with $\mathbb{S}$ the set of all strings;

- $|D|$ contains all messages, i.e., $\mathbb{M} \subset |D|$.

Let $\mathbb{T}$ be the set of all ROS topics. For all `/t` $\in \mathbb{T}$ we define a predicate $t \in \mathsf{P}$ such that $t(m)$ is true if and only if a message $m \in \mathbb{M}$ can be observed on topic `/t`. The relation of messages to data fields is given by predicates $\mathit{field}(m, x) \in \mathsf{P}$ such that, for a message m, $\mathit{field}(m, x)$ holds if and only if m carries the value (or message) x in a data field named `field`.

Array fields are only slightly different. To simplify quantification over the indices of an array, we redefine the usual predicate $\mathit{field}(m, k)$, to hold for all indices belonging to the array, rather than values. That is, for $k \in \mathbb{N}_0$, $\mathit{field}(m, k)$ holds if k is an index of an array named `field`. In addition, we define a predicate $\mathit{field}_k(m, x)$ for all indices k, such that $\mathit{field}_k(m, x)$ holds if the field `field[k]` carries the value x, i.e., if $\mathit{field}(m, k)$ holds and the array contains x at the index k.

We address composed messages with predicate composition. That is, for a data field `f.g`, we use the composition of predicates f and g, such that

$$g \circ f(m, x) \equiv (\exists m' : f(m, m') \wedge g(m', x))$$

To shorten some formulae, and to improve readability, we assume that all interpreted properties Φ are type-checked prior to their semantic interpretation $[\![\Phi]\!]$. For instance, in the case of arithmetic operators, for an expression $a + b$, we assume that both a and b are such that $a, b \in \mathbb{R}$. Otherwise, we would have to introduce a new predicate, $isNumber(x)$, such that for all x, $isNumber(x)$ holds if $x \in \mathbb{R}$, and replace every arithmetic-related formula φ with $isNumber(a) \wedge isNumber(b) \wedge \varphi$.

Definition 4. *Let (D, τ) be a temporal structure over S, with $D = (D_0, D_1, \ldots)$ and $\tau = (\tau_0, \tau_1, \ldots)$. Let v be a valuation, $i \in \mathbb{N}_0$ and Φ a property over S. We define $[\![\Phi]\!]$, the interpretation of Φ, as a function that translates Φ to its equivalent Metric First-Order Temporal Logic formula. Thus, we define $(D, \tau, v, i) \models \Phi$ as $(D, \tau, v, i) \models [\![\Phi]\!]$, and $[\![\Phi]\!]$ is defined as follows.*

Scopes

$$[\![\texttt{globally}\colon \Psi]\!] \triangleq [\![\Psi]\!]^{\emptyset}_{\bot}$$

$$[\![\texttt{after}\ p\colon \Psi]\!] \triangleq \Box(\forall x : \mathit{enterScope}(p, \bot, x) \rightarrow [\![\Psi]\!]^{x}_{\bot})$$

$$[\![\texttt{until}\ q\colon \Psi]\!] \triangleq [\![\Psi]\!]^{\emptyset}_{q}$$

$$[\![\texttt{after}\ p\ \texttt{until}\ q\colon \Psi]\!] \triangleq \Box(\forall x : \mathit{enterScope}(p, q, x) \rightarrow [\![\Psi]\!]^{x}_{q})$$

$$\mathit{enterScope}(p, q, x) \triangleq [\![p]\!]^{x} \wedge \neg(\exists y : [\![q]\!]^{y}) \wedge \mathcal{Z}(\neg(\exists y : [\![p]\!]^{y})\ \mathcal{B}\ (\exists y : [\![q]\!]^{y}))$$

In the presented translation semantics we can see the use an auxiliary predicate, *enterScope*, that deserves further explanation. The `globally` and `until` scopes start, by definition, at the initial instant of the trace and, thus, require the resulting formula to be true when evaluated at that instant. On the other hand, the `after` and `after-until` scopes do not start until the matching event has been observed. We

know that a property must only hold within its scope, so the resulting formulae have to account for this when evaluated at the initial instant of the trace – recall that a temporal structure (D, τ) satisfies a formula φ, written $(D, \tau) \models \varphi$, if and only if $(D, \tau, v, 0) \models \varphi$, for a valuation v. In addition, the `after-until` scope is (possibly) reentrant, which means that *the activation of a scope* might be observed multiple times. This is why these two scopes require the $\Box$ operator – so that, whenever an event marks the start of the scope of a property, we can evaluate the inner pattern. Hence, given activator and terminator events p and q, and given a message x, *enterScope*(p, q, x) holds if and only if the message x starts the scope at the current instant. More precisely, *enterScope*(p, q, x) holds if its three conjuncts are satisfied:

- x matches the event p, i.e., x is an activator message; and
- no message at that instant matches q (as it would terminate the scope immediately); and
- in the previous instants, since either the initial instant or the last terminator event (for reentrant scopes), there has been no other activator, i.e., x is the *first* match for p (further matches when within the scope are ignored).

The third conjunct of *enterScope* imposes a limitation on the language. Due to its reference to a previous scope, and due to the semantics of MFOTL, terminator events must be independent; a terminator event cannot reference previous events (i.e., the scope activator). If such cross-references were allowed, we would need to pass additional messages in the superscript vector of $[\![q]\!]^{y}$, such as $[\![q]\!]^{x,y}$. But, in the third conjunct, the back-to operator refers to the terminator of a previous scope (which would also have references), and we cannot say, at that instant, which message opened the scope of that terminator. In summary, allowing terminator events to refer to activator events would make the definition of *enterScope* recursive.

Note that, for the translation of patterns and predicates, we used superscript and subscript annotations. This is mostly a convenience to present the translation in a compositional fashion, from a top-down perspective, and to allow us to carry over to the lower levels of the translation some variables that were introduced at the higher levels. For instance, in $[\![\Psi]\!]^{\overline{x}}_{q}$, the superscript $\overline{x}$ is a vector of quantified messages. In the translation of `after` and `after-until`, this vector contains a single message, the scope activator message. In the translation of `globally` and `until`, it is $\emptyset$, representing an empty vector. Similarly, the superscript $\emptyset$ represents a vector containing zero messages. The subscript q is used to carry over the terminator event from the scope to the translation of the pattern. The same notation applies to $[\![p]\!]^{\overline{x}}$, except that p is a predicate, rather than a pattern.

Patterns

$$
\begin{aligned}
\llbracket \texttt{no}\ b \rrbracket_q^{\overline{x}} &\triangleq \neg(\exists y : \llbracket b \rrbracket^{\overline{x},y})\ \mathcal{W}\ (\exists y : \llbracket q \rrbracket^{y}) \\
\llbracket \texttt{no}\ b\ \texttt{within}\ \delta\ \texttt{ms} \rrbracket_q^{\overline{x}} &\triangleq \neg(\exists y : \llbracket b \rrbracket^{\overline{x},y})\ \mathcal{W}_{[0,\delta)}\ (\exists y : \llbracket q \rrbracket^{y}) \\
\llbracket \texttt{some}\ b \rrbracket_q^{\overline{x}} &\triangleq \neg(\exists y : \llbracket q \rrbracket^{y})\ \mathcal{U}\ ((\exists y : \llbracket b \rrbracket^{\overline{x},y}) \wedge \neg(\exists y : \llbracket q \rrbracket^{y})) \\
\llbracket \texttt{some}\ b\ \texttt{within}\ \delta\ \texttt{ms} \rrbracket_q^{\overline{x}} &\triangleq \neg(\exists y : \llbracket q \rrbracket^{y})\ \mathcal{U}_{[0,\delta)}\ ((\exists y : \llbracket b \rrbracket^{\overline{x},y}) \wedge \neg(\exists y : \llbracket q \rrbracket^{y})) \\
\llbracket a\ \texttt{causes}\ b \rrbracket_q^{\overline{x}} &\triangleq (\forall y : \llbracket a \rrbracket^{\overline{x},y} \rightarrow \llbracket \texttt{some}\ b \rrbracket_q^{\overline{x},y})\ \mathcal{W}\ (\exists y : \llbracket q \rrbracket^{y}) \\
\llbracket a\ \texttt{causes}\ b\ \texttt{within}\ \delta\ \texttt{ms} \rrbracket_q^{\overline{x}} &\triangleq (\forall y : \llbracket a \rrbracket^{\overline{x},y} \rightarrow \llbracket \texttt{some}\ b\ \texttt{within}\ \delta\ \texttt{ms} \rrbracket_q^{\overline{x},y})\ \mathcal{W}\ (\exists y : \llbracket q \rrbracket^{y}) \\
\llbracket a\ \texttt{forbids}\ b \rrbracket_q^{\overline{x}} &\triangleq (\forall y : \llbracket a \rrbracket^{\overline{x},y} \rightarrow \llbracket \texttt{no}\ b \rrbracket_q^{\overline{x},y})\ \mathcal{W}\ (\exists y : \llbracket q \rrbracket^{y}) \\
\llbracket a\ \texttt{forbids}\ b\ \texttt{within}\ \delta\ \texttt{ms} \rrbracket_q^{\overline{x}} &\triangleq (\forall y : \llbracket a \rrbracket^{\overline{x},y} \rightarrow \llbracket \texttt{no}\ b\ \texttt{within}\ \delta\ \texttt{ms} \rrbracket_q^{\overline{x},y})\ \mathcal{W}\ (\exists y : \llbracket q \rrbracket^{y}) \\
\llbracket b\ \texttt{requires}\ a \rrbracket_q^{\overline{x}} &\triangleq \forall y : \mathit{prec}(a,b,q,\overline{x},y) \\
\llbracket b\ \texttt{requires}\ a\ \texttt{within}\ \delta\ \texttt{ms} \rrbracket_q^{\overline{x}} &\triangleq \forall y : \mathit{prec}(a,b,q,\overline{x},y) \wedge \mathit{once}_\delta(a,b,q,\overline{x},y) \\
\mathit{prec}(a,b,q,\overline{x},y) &\triangleq \neg\llbracket b \rrbracket^{\overline{x},y}\ \mathcal{W}\ (\exists z : \llbracket a \rrbracket^{\overline{x},y,z} \vee \llbracket q \rrbracket^{z}) \\
\mathit{once}_\delta(a,b,q,\overline{x},y) &\triangleq (\llbracket b \rrbracket^{\overline{x},y} \rightarrow \blacklozenge_{[0,\delta)}(\exists z : \llbracket a \rrbracket^{\overline{x},y,z}))\ \mathcal{W}\ (\exists z : \llbracket q \rrbracket^{z})
\end{aligned}
$$

For the translation of patterns, we now see additional superscript annotations, not discussed for scopes. For instance, in $\llbracket \texttt{no}\ b \rrbracket_q^{\overline{x}}$, the superscript $\overline{x}$ is a vector (of arbitrary length) of quantified messages that causally precede b; such as the activator message, or no messages at all if the vector is of zero-length. Alternatively, as we will see in following translations, we can also use the notation $\llbracket b \rrbracket^{x_1,\ldots,x_n}$ to refer to a vector of length n. Furthermore, the use of the superscript $\llbracket b \rrbracket^{\overline{x},y}$ means that the translation of the predicate b should consider a vector of all messages in $\overline{x}$ plus the quantified message y.

We can see that the unary patterns, **no** and **some** have a translation that is already close a MFOTL formula, save for the translation of the event predicates, $\llbracket b \rrbracket$ and $\llbracket q \rrbracket$. The binary patterns **causes** and **forbids** are translated via composition, reusing the translations of the previous unary patterns. The last binary pattern, **requires**, is more complex, and is defined with the use of auxiliary predicates for readability purposes. The first of these predicates, $\mathit{prec}(a,b,q,\overline{x},y)$ states that the event b (the behaviour) should not happen before a (the required event) or q (the scope terminator). When adding real-time constraints, however, we also need the predicate $\mathit{once}_\delta(a,b,q,\overline{x},y)$, which states that, if the event b happens before the scope terminator, then the required event a must have happened within the specified interval (the previous δ time instants). The remaining translations, $\llbracket a \rrbracket$, $\llbracket b \rrbracket$ and $\llbracket q \rrbracket$ all represent predicate translations, which we present next.

Predicates

$$
\begin{aligned}
[\![\top]\!]^{\overline{x}} &\triangleq \top \\
[\![\bot]\!]^{\overline{x}} &\triangleq \bot \\
[\![\,\texttt{/topic}\,]\!]^{x_1,\ldots,x_n} &\triangleq \textit{topic}(x_n) \\
[\![\,\texttt{/topic \{ }\varphi\texttt{ \}}\,]\!]^{x_1,\ldots,x_n} &\triangleq \textit{topic}(x_n) \wedge [\![\varphi]\!]^{x_1,\ldots,x_n}_{\emptyset} \\
[\![(\varphi)]\!]^{\overline{x}}_{\gamma} &\triangleq ([\![\varphi]\!]^{\overline{x}}_{\gamma}) \\
[\![\texttt{not } \varphi]\!]^{\overline{x}}_{\gamma} &\triangleq \neg[\![\varphi]\!]^{\overline{x}}_{\gamma} \\
[\![\varphi \texttt{ and } \psi]\!]^{\overline{x}}_{\gamma} &\triangleq [\![\varphi]\!]^{\overline{x}}_{\gamma} \wedge [\![\psi]\!]^{\overline{x}}_{\gamma} \\
[\![\varphi \texttt{ or } \psi]\!]^{\overline{x}}_{\gamma} &\triangleq [\![\varphi]\!]^{\overline{x}}_{\gamma} \vee [\![\psi]\!]^{\overline{x}}_{\gamma} \\
[\![\texttt{forall var in} f\!: \varphi]\!]^{\overline{x}}_{\gamma} &\triangleq \forall y : {}_{y}[\![f]\!]^{\overline{x}}_{\gamma} \rightarrow [\![\varphi]\!]^{\overline{x}}_{\gamma[\texttt{var}\mapsto y]} \\
[\![\texttt{forall var in @}k.f\!: \varphi]\!]^{\overline{x}}_{\gamma} &\triangleq \forall y : {}_{y}[\![\texttt{@}k.f]\!]^{\overline{x}}_{\gamma} \rightarrow [\![\varphi]\!]^{\overline{x}}_{\gamma[\texttt{var}\mapsto y]} \\
[\![a = b]\!]^{\overline{x}}_{\gamma} &\triangleq \exists y,z : {}_{y}[\![a]\!]^{\overline{x}}_{\gamma} \wedge {}_{z}[\![b]\!]^{\overline{x}}_{\gamma} \wedge y = z \\
[\![a < b]\!]^{\overline{x}}_{\gamma} &\triangleq \exists y,z : {}_{y}[\![a]\!]^{\overline{x}}_{\gamma} \wedge {}_{z}[\![b]\!]^{\overline{x}}_{\gamma} \wedge y < z
\end{aligned}
$$

The translations of most predicates are relatively straightforward. Atomic predicates, such as $\top$ and $\bot$ are directly translated. Top-level predicates, including a topic name, translate to a predicate with the same name as the topic, as previously explained, that checks whether the last message of the superscript vector is present in that topic, at the given time instant (i.e., the message can be observed at that time). Simple logic operators, **`not`**, **`and`**, **`or`**, translate to their respective logic operators and propagate the translation of subformulae. Quantifiers and atomic formulae are not as intuitive.

In this semantics, most expressions are singleton values; the exception lies in array message fields, that can represent multiple values. Namely, the predicate with the name of the array field denotes every index of the array. In contrast, for simple message fields we use *field*(x, y) to denote a predicate that tests whether y is a value of the homonymous field in message x. Thus, when translating expressions in general, we use the notation ${}_{y}[\![a]\!]^{\overline{x}}_{\gamma}$ to represent a predicate that tests whether the value of the variable y is one of the possible values for the expression a.

To handle new variable definitions for quantification, we use an additional γ subscript. This subscript is a mapping of syntactic replacements; given a variable name *as written in the property*, we can replace its occurrences with a quantified variable in $|D|$. For instance, given the predicate '`forall i in array: array[$i] = 0`', we quantify over all values for which ${}_{y}[\![\texttt{array}]\!]^{\overline{x}}_{\gamma}$ holds, i.e., all the indices of the array, as we will see next. For such values, y, the result of $[\![\,\texttt{array[\$i] = 0}\,]\!]^{\overline{x}}_{\gamma[\texttt{i}\mapsto y]}$ must hold. This is equivalent to stating that, in the translation of `array[$i] = 0` we should replace every occurrence of `$i` with the quantified variable y (the array index).

Atomic formulae, such as $a = b$, would, intuitively, just translate the left and right operands, as in $[\![a]\!]^{\overline{x}}_{\gamma} = [\![b]\!]^{\overline{x}}_{\gamma}$. This, however, is not possible, due to expressions having multiple possible values (in

general), as previously explained. We require existential quantification to bind variables to their respective values. Then, the $=$ and $<$ operators can only be applied to expressions that denote singleton values. The use of array fields, an expression that does not represent a singleton value, is confined to quantification.

Values and Expressions

$$
\begin{array}{ll}
{}_y[\![(a)]\!]_\gamma^{\overline{x}} & \triangleq ({}_y[\![a]\!]_\gamma^{\overline{x}}) \\
{}_y[\![\text{-}a]\!]_\gamma^{\overline{x}} & \triangleq \exists z : {}_z[\![a]\!]_\gamma^{\overline{x}} \land y = -z \\
{}_y[\![a + b]\!]_\gamma^{\overline{x}} & \triangleq \exists z, w : {}_z[\![a]\!]_\gamma^{\overline{x}} \land {}_w[\![b]\!]_\gamma^{\overline{x}} \land y = (z + w) \\
{}_y[\![a * b]\!]_\gamma^{\overline{x}} & \triangleq \exists z, w : {}_z[\![a]\!]_\gamma^{\overline{x}} \land {}_w[\![b]\!]_\gamma^{\overline{x}} \land y = (z \times w) \\
{}_y[\![c]\!]_\gamma^{\overline{x}} & \triangleq y = c \\
{}_y[\![\texttt{\$var}]\!]_\gamma^{\overline{x}} & \triangleq y = \gamma(\texttt{var}) \\
{}_y[\![f]\!]_\gamma^{x_1,\ldots,x_n} & \triangleq [\![f]\!]_\gamma(x_n, y) \\
{}_y[\![@k.f]\!]_\gamma^{x_1,\ldots,x_n} & \triangleq [\![f]\!]_\gamma(x_k, y) \quad \text{if } 1 \leq k \leq n \\
[\![\texttt{field}]\!]_\gamma & \triangleq \textit{field} \\
[\![\texttt{field[}k\texttt{]}]\!]_\gamma & \triangleq \textit{field}_k \\
[\![\texttt{field[\$var]}]\!]_\gamma & \triangleq \textit{field}_{\gamma(\texttt{var})} \\
[\![f_1 . f_2]\!]_\gamma & \triangleq [\![f_2]\!]_\gamma \circ [\![f_1]\!]_\gamma
\end{array}
$$

The translations of values and expressions are the last step to obtain a MFOTL formula. There is nothing new at this point, except field references to other messages, e.g., $@k.f$. Recall that event aliases are syntactic sugar that is converted to a message index at this point, i.e., k is a number, rather than a human-readable name. The translation for this case, then, refers to the k-th message given in the superscript vector, x_k, rather than the current message, x_n. The fields themselves are translated to their homonymous predicates, and field composition translates to predicate composition, as previously explained. Arithmetic expressions are similar to the translations of atomic formulae, and c denotes a constant symbol, such as an integer.

4.4.1 *Examples*

We provide examples of the language's property interpretation semantics, $[\![\Phi]\!]$, using concrete traces of the Fictibot system. Consider the trace in Figure 34, for instance. Each circle in the diagram represents an event where the underlined text above or below it is the time stamp of the event and the dashed box contains the associated topic name and message data (between braces, e.g., `{data: 0}`). The arrows in the diagram represent the order between events in the trace, starting from the earliest event to the latest.

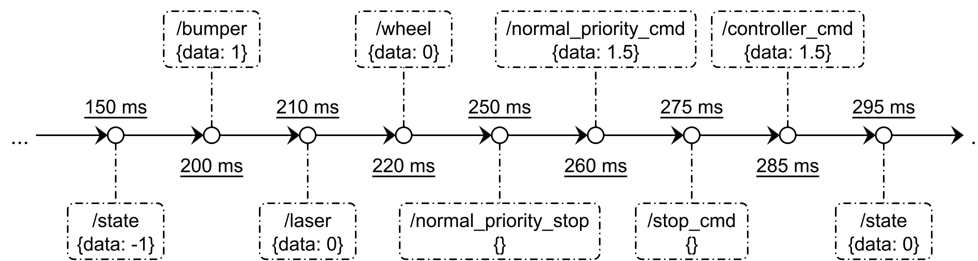


Figure 34: Example trace of the Fictibot system.

With slight modifications of the trace in Figure 34 we can illustrate examples of violated properties in Fictibot. The altered trace diagrams highlight the events causing the violation, using filled circles and red text for the time stamp (if it is a timing violation) or for the event's data (if it is a predicate violation).

Absence Figure 35 shows a trace excerpt where the following Absence property of the Fictibot Driver is violated.

```
1 globally: no /bumper {data < 0 or (not data < 8)}
```

A message is published on `/bumper` with a value of 8, but it is only expected to publish data between 0 and 7 (inclusive).

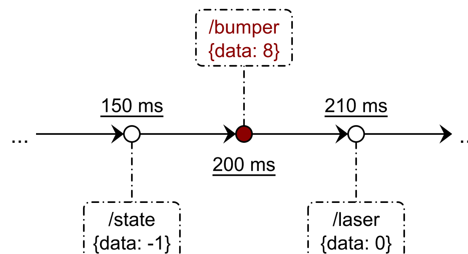


Figure 35: Example trace of the Fictibot system violating an Absence property.

We translate the property into the following formula, while also applying some simplifications to make it more readable.

$$
\begin{aligned}
& [\![\texttt{globally: no /bumper \{data < 0 or (not data < 8)\}}]\!] \\
\text{iff}\quad & [\![\texttt{no /bumper \{data < 0 or (not data < 8)\}}]\!]^{\emptyset}_{\bot} \\
\text{iff}\quad & \neg(\exists x : [\![\texttt{/bumper \{data < 0 or (not data < 8)\}}]\!]^{x}) \;\mathcal{W}\; (\exists x : [\![\bot]\!]^{x}) \\
\text{iff}\quad & \neg(\exists x : [\![\texttt{/bumper \{data < 0 or (not data < 8)\}}]\!]^{x}) \;\mathcal{W}\; \bot \\
\text{iff}\quad & \Box(\neg(\exists x : [\![\texttt{/bumper \{data < 0 or (not data < 8)\}}]\!]^{x})) \\
\text{iff}\quad & \Box(\neg(\exists x : \mathit{bumper}(x) \land [\![\texttt{data < 0 or (not data < 8)}]\!]^{x}_{\emptyset})) \\
\text{iff}\quad & \Box(\neg(\exists x : \mathit{bumper}(x) \land ([\![\texttt{data < 0}]\!]^{x}_{\emptyset} \lor [\![\texttt{not data < 8}]\!]^{x}_{\emptyset}))) \\
\text{iff}\quad & \Box(\neg(\exists x : \mathit{bumper}(x) \land ([\![\texttt{data < 0}]\!]^{x}_{\emptyset} \lor \neg[\![\texttt{data < 8}]\!]^{x}_{\emptyset}))) \\
\text{iff}\quad & \Box(\neg(\exists x : \mathit{bumper}(x) \land ((\exists y,z : {}_{y}[\![\texttt{data}]\!]^{x}_{\emptyset} \land {}_{z}[\![0]\!]^{x}_{\emptyset} \land y < z) \\
& \qquad\qquad \lor \neg(\exists y,z : {}_{y}[\![\texttt{data}]\!]^{x}_{\emptyset} \land {}_{z}[\![8]\!]^{x}_{\emptyset} \land y < z)))) \\
\text{iff}\quad & \Box(\neg(\exists x : \mathit{bumper}(x) \land ((\exists y,z : \mathit{data}(x,y) \land z = 0 \land y < z) \\
& \qquad\qquad \lor \neg(\exists y,z : \mathit{data}(x,y) \land z = 8 \land y < z)))) \\
\text{iff}\quad & \Box(\neg(\exists x : \mathit{bumper}(x) \land ((\exists y : \mathit{data}(x,y) \land y < 0) \lor \neg(\exists y : \mathit{data}(x,y) \land y < 8))))
\end{aligned}
$$

It is easy to see that when evaluated at the initial instant of the trace, this formula does not hold. The second message of the trace, with the highlights, is a message fitting for the quantified variable x. In particular, it is a message such that $\mathit{bumper}(x)$, and one such that $\neg(\exists y : \mathit{data}(x,y) \land y < 8)$. Such conditions should never hold, as given by $\Box(\neg\varphi)$, resulting in a violation.

Response Figure 36 shows a trace excerpt where the following Response property of the Fictibot Random Controller is violated.

```
1 globally: /bumper {not data = 0} causes /normal_priority_stop within 200 ms
```

In this case, it is a violation of the timing aspect of the property. The stimulus, a `/bumper` message such that `data` $\neq 0$, should have led to the publication of a message on `/normal_priority_stop`. While the system does send the message, it does not respect the 200 millisecond response window.

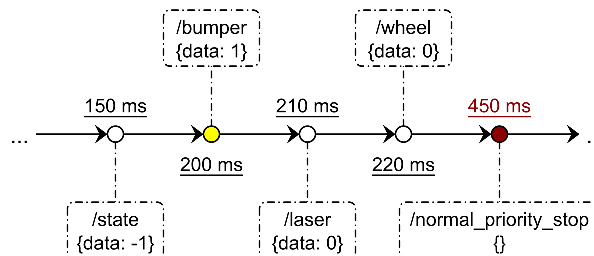


Figure 36: Example trace of the Fictibot system violating a Response property.

We translate the property into the following formula.

$$
\begin{aligned}
& [\![\texttt{globally: /bumper \{not data = 0\} causes /normal_priority_stop within 200 ms}]\!] \\
\text{iff}\quad & [\![\texttt{/bumper \{not data = 0\} causes /normal_priority_stop within 200 ms}]\!]^{\emptyset}_{\bot} \\
\text{iff}\quad & (\forall x : [\![\texttt{/bumper \{not data = 0\}}]\!]^{x} \rightarrow [\![\texttt{some /normal_priority_stop within 200 ms}]\!]^{x}_{\bot})\ \mathcal{W}\ (\exists x : [\![\bot]\!]^{x}) \\
\text{iff}\quad & (\forall x : [\![\texttt{/bumper \{not data = 0\}}]\!]^{x} \rightarrow [\![\texttt{some /normal_priority_stop within 200 ms}]\!]^{x}_{\bot})\ \mathcal{W}\ \bot \\
\text{iff}\quad & \square(\forall x : [\![\texttt{/bumper \{not data = 0\}}]\!]^{x} \rightarrow [\![\texttt{some /normal_priority_stop within 200 ms}]\!]^{x}_{\bot}) \\
\text{iff}\quad & \square(\forall x : (\mathit{bumper}(x) \wedge \neg(\exists y : \mathit{data}(x,y) \wedge y = 0)) \\
& \qquad \rightarrow [\![\texttt{some /normal_priority_stop within 200 ms}]\!]^{x}_{\bot}) \\
\text{iff}\quad & \square(\forall x : (\mathit{bumper}(x) \wedge \neg \mathit{data}(x,0)) \rightarrow [\![\texttt{some /normal_priority_stop within 200 ms}]\!]^{x}_{\bot}) \\
\text{iff}\quad & \square(\forall x : (\mathit{bumper}(x) \wedge \neg \mathit{data}(x,0)) \\
& \qquad \rightarrow (\neg(\exists y : [\![\bot]\!]^{y})\ \mathcal{U}_{[0,200)}\ ((\exists y : [\![\texttt{/normal_priority_stop}]\!]^{x,y}) \wedge \neg(\exists y : [\![\bot]\!]^{y})))) \\
\text{iff}\quad & \square(\forall x : (\mathit{bumper}(x) \wedge \neg \mathit{data}(x,0)) \\
& \qquad \rightarrow (\neg\bot\ \mathcal{U}_{[0,200)}\ ((\exists y : [\![\texttt{/normal_priority_stop}]\!]^{x,y}) \wedge \neg\bot))) \\
\text{iff}\quad & \square(\forall x : (\mathit{bumper}(x) \wedge \neg \mathit{data}(x,0)) \rightarrow \Diamond_{[0,200)}(\exists y : [\![\texttt{/normal_priority_stop}]\!]^{x,y})) \\
\text{iff}\quad & \square(\forall x : (\mathit{bumper}(x) \wedge \neg \mathit{data}(x,0)) \rightarrow \Diamond_{[0,200)}(\exists y : \mathit{normal_priority_stop}(y)))
\end{aligned}
$$

As per the formula above, whenever a `/bumper` message such that $\texttt{data} \neq 0$ is observed, a `/normal_priority_stop` message (any message) must be published at most 200 milliseconds afterwards. We know from Figure 36 that such a response message exists, but not within the time window, thus falsifying the property.

After-Until Scope For illustrative purposes, we also show the translation the following Fictibot Multiplexer property.

```
1 after /state {data > 0} until /state {not data > 0}: /controller_cmd as CMD
2     requires /high_priority_cmd {data = @CMD.data} within 100 ms
```

This is a more complex example, using both the after-until scope and a Precedence pattern. Thus, we skip the intermediate steps of the translation.

$$
\begin{aligned}
&[\![\texttt{after /state \{data > 0\} until /state \{not data > 0\}: /controller_cmd as CMD}\\
&\quad\texttt{requires /high_priority_cmd \{data = @CMD.data\} within 100 ms}]\!]\\
\text{iff}\quad &\Box(\forall x : ([\![\texttt{/state \{data > 0\}}]\!]^x \wedge \neg(\exists y : [\![\texttt{/state \{not data > 0\}}]\!]^y)\\
&\qquad \wedge \mathcal{Z}(\neg(\exists y : [\![\texttt{/state \{data > 0\}}]\!]^y)\ \mathcal{B}\ (\exists y : [\![\texttt{/state \{not data > 0\}}]\!]^y)))\\
&\quad \rightarrow [\![\texttt{/controller_cmd as CMD requires /high_priority_cmd \{data = @CMD.data\}}\\
&\qquad\quad \texttt{within 100 ms}]\!]^x_{\texttt{/state \{not data > 0\}}})\\
\text{iff}\quad &\Box(\forall x : \varphi \rightarrow \psi)\\
\text{with}\quad &\varphi \equiv (\mathit{state}(x) \wedge (\exists y : \mathit{data}(x,y) \wedge (y > 0)))\\
&\qquad \wedge \neg(\exists y : \mathit{state}(y) \wedge (\exists z : \mathit{data}(y,z) \wedge (z \leq 0)))\\
&\qquad \wedge \mathcal{Z}(\neg(\exists y : (\mathit{state}(y) \wedge (\exists z : \mathit{data}(y,z) \wedge (z > 0))))\\
&\qquad\qquad \mathcal{B}\ (\exists y : \mathit{state}(y) \wedge (\exists z : \mathit{data}(y,z) \wedge (z \leq 0))))\\
&\psi \equiv \forall y : ((\neg \mathit{controller_cmd}(y))\ \mathcal{W}\ (\exists z : (\mathit{high_priority_cmd}(z)\\
&\qquad \wedge (\exists w : \mathit{data}(z,w) \wedge \mathit{data}(y,w))) \vee (\mathit{state}(z) \wedge (\exists w : \mathit{data}(z,w) \wedge (w \leq 0)))))\\
&\qquad \wedge ((\mathit{controller_cmd}(y) \rightarrow \blacklozenge_{[0,100)}(\exists z : (\mathit{high_priority_cmd}(z)\\
&\qquad \wedge (\exists w : \mathit{data}(z,w) \wedge \mathit{data}(y,w))))) \ \mathcal{W}\ (\exists z : \mathit{state}(z) \wedge (\exists w : \mathit{data}(z,w) \wedge (w \leq 0))))
\end{aligned}
$$

The resulting MFOTL formula is complex, but it follows a clear structure. The antecedent, φ, is a subformula that tests whether the current message, x, is the scope activator – it is the translation (and expansion) of our *enterScope* auxiliary predicate. The consequent, ψ, is the translation of the pattern itself – a subformula that tests whether a message y on `/controller_cmd` is preceeded by a message z on `/high_priority_cmd` in the previous 100 milliseconds, such that the contents of the *data* field are equal.

4.5 Summary

Specifying robot behaviour is far from trivial. It raises many questions regarding the right level of abstraction, or how to handle time. Ideally, a specification is as accurate and precise as possible (e.g., real time, as opposed to logical time), but this incurs considerable complexity – both for the specifications themselves as well as the verification process. There are a number of abstractions to address this issue, based on, e.g., state machines or event traces. Most of these abstractions end up using Linear Temporal Logic, or some variant of it, for the semantics of the behavioural properties.

In this chapter we present a property specification language for ROS that aims for a minimalistic syntax, while packing powerful and relatively flexible semantics. We base our approach on the idea of event traces, where observable events amount, essentially, to the messages that ROS nodes exchange over topics. This language is heavily inspired by Dwyer et al.'s specification patterns [58, 59], although we consider real time, event parametrisation and quantification to be first-class citizens – meaning that the classical Linear Temporal Logic does not suffice for the language's semantics. We settle on Metric First-Order Temporal

Logic instead, with both future and past modalities. The fact that we focus on the publisher-subscriber aspect of ROS also plays a role on defining the syntax, helping us minimise it, but, at the same time, retaining details that are familiar to ROS developers. The proposed language is a complement to the architectural models presented in the previous chapter – it is the missing piece for a full specification of ROS software.

5

A FRAMEWORK FOR ROS-SPECIFIC ANALYSIS

One of the main goals of this thesis is to conceive and implement software analysis techniques specifically adapted for the ROS environment. Verification of software properties is a complex process that can be achieved with a variety of techniques. While some properties – often the simplest and least relevant – can be verified in a fully autonomous fashion using capable tools, others require increasing degrees of human intervention, possibly to the point where proofs and evidence are gathered entirely by hand. Our work focuses on *lightweight formal methods* – to achieve as much automation as possible, without considerable loss in terms of verification power. That is, we aim for verification tools and techniques that require little manual work, but are, at the same time, capable of finding violations of meaningful properties (behavioural and otherwise) in complex ROS systems.

In the previous chapters we described a metamodel for the software architecture of ROS applications, annotated with behavioural properties written in a domain-specific language. In this chapter we introduce HAROS [148], a framework for the analysis of ROS software that serves as the basis for our prototype analysis tools – both in terms of automatic architectural model extraction, and in terms of checking whether systems comply with behavioural properties. The introduction in Section 5.1 presents work that was mostly done prior to this thesis. New contributions start from Section 5.2 onward.

5.1 High-Assurance ROS

The High-Assurance ROS framework[1] (HAROS, for short) was initially conceived as an aggregator for existing static analysis tools. Before delving into ROS-specific software analyses – which is the theme of this thesis – we conducted a study [148] to assess the overall software quality of the ROS ecosystem (using general-purpose metrics), as well as its compliance with a number of coding standards. This study aimed to evaluate a significant corpus of the official ROS distribution and find out whether, on average, package authors cared about software quality and maintainability. Striving for high-quality, maintainable software is standard practice in most well-established software industries, especially in safety-critical contexts. But

1 https://github.com/git-afsantos/haros/

robotics is a relatively new domain, where developers are not necessarily (or likely) software engineers, but rather, e.g., electrical engineers who end up building software out of necessity. Such a study had not been conducted before in the growing ROS ecosystem, and we found out that, indeed, software quality was inconsistent.

The study on software quality was based on tried and tested metrics – e.g., lines of code, cyclomatic complexity, coupling between objects – and compliance checks with well-known coding standards – e.g., Google's C++ Style Guide, MISRA C++, High-Integrity C++. To collect this information, we used a number of existing static analysis tools, each contributing with small pieces (e.g., a few metrics for C++, or coding standards for Python) that we combined to achieve the full panorama. The main problem with using several analysis tools is that it results in processing large amounts of output, provided in diverse formats. Some tools report error messages to the standard output in a terminal, while others produce XML or HTML reports containing their findings. Furthermore, there is the issue of feeding the tools with the appropriate input. Some tools work on a file by file basis, while others scan entire directories recursively. So, to effectively run the toolset over a corpus of ROS packages and aggregate the results by package, a considerable amount of set up is required. This was one of the main motivators to implement HAROS.

5.1.1 *Architecture and Workflow*

HAROS is specifically designed for the analysis of ROS software. Even as a prototype, one of its main features was embedded knowledge about how ROS packages are structured and found in a file system. Given a list of package names for analysis, it is able to find a local version of the packages, or download them from the official distribution. It is able to navigate packages and identify source code files within, categorised per programming language. Essentially, its prototype was able to build most of the file system entities from our proposed metamodel (see *Package*, *Repository*, *Source File* in Section 3.2).

The analysis of ROS packages is based on a plug-in architecture. The intent was not to reinvent the wheel of static analysis, but to reuse existing analysis tools. A plug-in based design combines the best of both worlds – a plug-in can be a thin wrapper for an existing tool, performing the necessary format conversions, or it can be a new analysis tool itself. This, combined with the previous feature of handling ROS packages directly, took care of the issue of feeding the analysis tools with proper input. The remaining issue was that of output heterogeneity, which HAROS also handled, by having plug-ins register their findings via a single interface. After all the analysis results are gathered, HAROS produces a unified interactive report, as well as several JSON data files containing, e.g., results per package. The diagram in Figure 37 provides an overview of the HAROS workflow just described.

The implementation of HAROS is divided into two components: the *analyser*, a Python console application that does the bulk of the work (sometimes simply called HAROS); and the *visualiser* (or *viz*, for short) which handles the interactive reports using web technologies. The choice of Python for the core of HAROS was one of convenience. HAROS is intended to be used alongside a ROS installation, and ROS, by default, already requires Python and has a few useful libraries available for Python scripting – e.g., *rospack*, a ROS utility to find packages in the file system, upon which HAROS relies internally to offer the same feature.

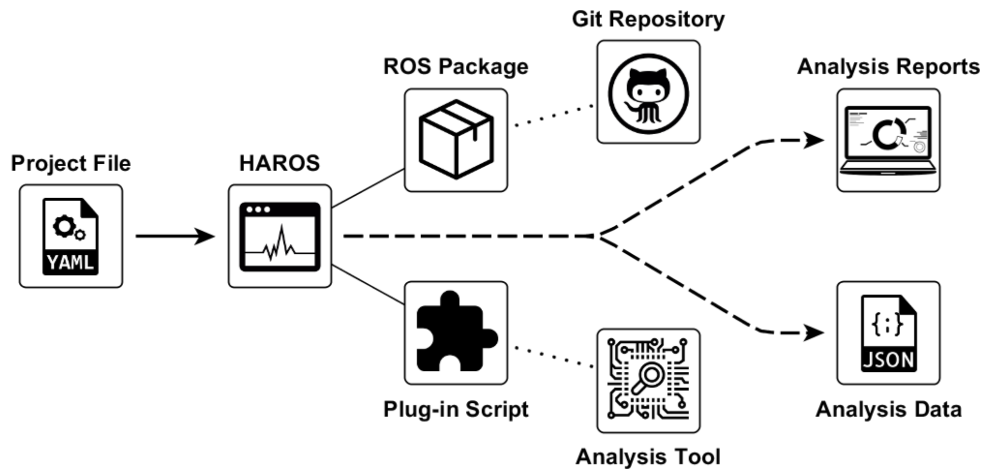


Figure 37: Workflow of the HAROS static analysis framework.

Thus, not only is Python adequate for the large amounts of configuration and scripting that HAROS boils down to, it is also readily available without introducing aditional dependencies for the end user. A similar reasoning applies for the choice of web technologies to display analysis reports.

It is typical for most console applications to have a number of options and parameters that may change their operation, and HAROS is no exception. However, most of its modes of operation have three stages in common, which we will detail: setup, analysis and reporting.

Setup Stage

During the setup stage, HAROS processes and builds all the necessary data structures for the following analysis stage. It starts by processing user options and parameters, given via a configuration file or via command-line arguments, which can be used, e.g., to determine whether it should incorporate repositories into the analysis (as opposed to isolated packages) or to turn off specific plug-ins. An important concept to take note of, not only for this stage but for HAROS as a whole, is the concept of a *project file* (included in Figure 37). This is a user-provided file, in YAML syntax, whose primary purpose, in this version of the framework, is to provide a project name and to enumerate the packages that should be analysed. Listing 5.1 shows a minimal project file for the Fictibot running example.

```
1 project: Fictibot
2 packages:
3     - fictibot_drivers
4     - fictibot_controller
5     - fictibot_multiplex
6     - fictibot_msgs
```

Listing 5.1: Minimal project file for Fictibot.

After reading the list of target packages, HAROS proceeds to locating them and building its internal data structures. This internal step, referred to as *indexing*, traverses the package directories, enumerates source code files, and organises packages in topological order, based on their dependencies – much like *catkin*,

the ROS build system, does before the actual building step. At this stage, users can also provide options to enable HAROS to locate (and download) missing packages using the official ROS distribution index.

Analysis Stage

When HAROS transitions to the analysis stage, all packages and other data structures are already loaded and in place. Thus, the first step is to load available analysis plug-ins, filtered by an optional whitelist or blacklist given by the user. In the following subsection we will go over the details of how a plug-in is structured and implemented. For now, it suffices to say that plug-ins can implement any number of a few entry point functions that are called in order by HAROS – entry points that constitute the interface a plug-in exposes to HAROS. Namely, plug-ins can provide a routine to analyse a single file, or to analyse an entire package (leaving traversal of the directories, or use of HAROS' representation, to the plug-in implementation). These entry point functions are provided with two arguments, one being the HAROS data structure that represents the intended scope (i.e., Source File or Package) and another being the opposite interface that plug-ins use to communicate back with HAROS (the *plug-in interface*).

The plug-in interface is not very extensive – as we can see from the summary in Listing 5.2. Its main purpose is to provide plug-ins with the two reporting functions, `report_violation` and `report_metric`. The former allows plug-ins to register all sorts of free-text issues, with optional arguments to refine traceability information (e.g., files, line numbers, function names). Its name stems from the coding standard oriented analysis that we intended to perform in [148]. The latter allows plug-ins to register numeric values for various metrics – another kind of analysis we performed in [148]. So, in summary, metrics and violations are the two sides of a HAROS analysis report.

```
1  class PluginInterface(LoggingObject):
2      """Provides an interface for plug-ins to communicate with the framework."""
3
4      # inherits logging functions from LoggingObject:
5      # log_debug(msg), log_warning(msg), log_error(msg)
6
7      def report_violation(self, rule_id, msg, scope=None,
8                           line=None, function=None, class_=None):
9          """Reports a rule violation detected by the plug-in during analysis."""
10
11     def report_metric(self, metric_id, value, scope=None,
12                       line=None, function=None, class_=None):
13         """Registers a metric's value, measured by the plug-in during analysis."""
```

Listing 5.2: Interface through which plug-ins communicate with HAROS.

Reporting Stage

In the reporting stage, the HAROS analyser aggregates the results obtained from plug-ins and exports a number of data files in JSON format. These files are organised by project, and then by package. There is also a summary file, containing an overview of all results, as well as a history of previous results for comparison. Listing 5.3 shows an excerpt of such a summary file.

```
"source": {
    "packages": 4,          (the project contains 4 packages in total)
    "files": 37,            (there are 37 Source Files within these packages)
    "languages": {
        "cpp": 0.559461480,  (55.95% of all code is C++)
        "python": 0
    },
    "scripts": 0
},
"history": {
    "issues": [143, 202, 208],  (number of issues in the 3 previous runs)
    "metrics": [9, 12, 12],     (number of previously registered metrics)
    "lines_of_code": [838, 1297, 1337],
    "timestamps": [
      "2018-02-24-18-31",
      "2018-02-28-20-46",
      "2018-03-08-18-12"
    ]
     (other entries were omitted)
},
"issues": {
    "total": 208,     (there are 208 issues reported by plug-ins in total)
    "metrics": 12,    (12 of these issues are related to metrics)
    "coding": 200,    (200 issues are related to coding standards)
    "ratio": "0.16",  (number of issues per line of code)
    "other": 0        (other types of issues)
}
```

Listing 5.3: Excerpt of a JSON data file exported by HAROS.

The HAROS Visualiser, as previously mentioned, produces an interactive report, based on the exported JSON data files, that users can explore. It defaults to a dashboard page (see Figure 38 for a screen capture of the most recent version of the dashboard), where the summary data is provided for a selected project. Projects can be switched on this page to load a different data set. This page sorts the information in three panels: source code statistics (e.g., number of packages and files, or the number of custom message types), analysis statistics (e.g., total number of issues reported by plug-ins) and history of several metrics.

Another page includes a package overview, where packages are drawn in a graph along with the dependencies between them, and a few panels contain general package information (mostly extracted from the XML package manifests). Yet another page is dedicated to the issues reported by plug-ins – the main item of interest in the report as a whole. This page is shown in Figure 39. Issues are organised by package and can be filtered using a menu. Each issue contains the violated rule, a location in the source code (where the issue was detected), additional free text comments provided by the analysis tools, and a set of *tags*. HAROS provides a short, central catalogue of common rules that plug-ins can reference. New rules can be added by the plug-ins themselves, in a plug-in manifest file, as we will see. Tags are associated to rules, and serve no special purpose other than helping to categorise the issues. They are created on an ad hoc fashion, i.e., there is no central catalogue of tags. Regardless, they are the main mechanism used for filtering issues – i.e., the *Filter* menu can be used to only show issues containing a number of tags, or to ignore issues containing such tags.

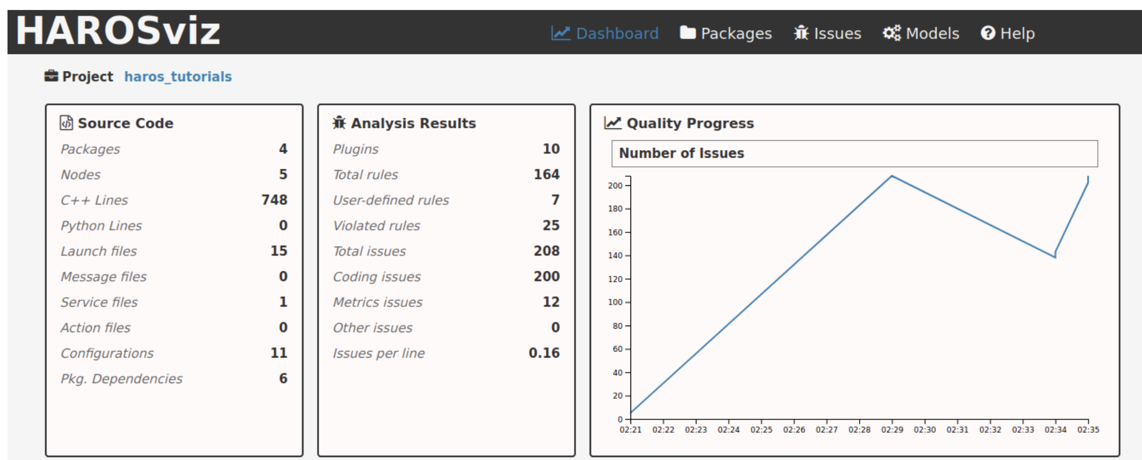


Figure 38: Summary page of the HAROS Visualiser.

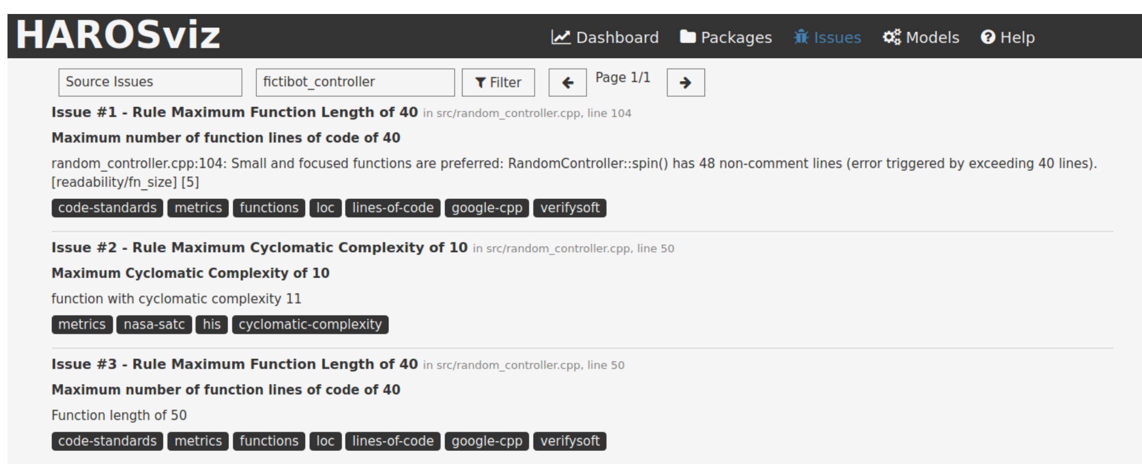


Figure 39: Issue listing with the HAROS Visualiser.

5.1.2 *HAROS Plug-ins*

Creating a plug-in for HAROS is relatively straightforward. Simply put, a plug-in is a Python package containing at least two files: `plugin.yaml` and `plugin.py`. The former is a manifest file containing metadata about the plug-in, such as its name, version, and (optionally) which programming languages does it support, as well as which metrics and rules does it analyse. New rule definitions are provided in this manifest file. As an example, consider Listing 5.4, which contains an excerpt of the manifest of one of the existing plug-ins for HAROS – a plug-in for the Radon[2] tool, a tool to gather metrics from Python source code. Most of the metadata in this file serves no special purpose other than pure information, but declaring any supported languages (Python, in this case), means that, for a file by file analysis, HAROS will only supply the plug-in with Python files. By declaring only Python support, the plug-in should never be called for analysis of C++ files, for instance.

The `plugin.py` file is a Python script that HAROS uses to look for the plug-in's entry points. From a top-down perspective, this is the main script of the plug-in's implementation. The only requirements

2 https://pypi.org/project/radon/

```yaml
name: haros_plugin_radon
version: 0.1
languages:
    - python
supported_rules:     (some entries were omitted)
    - max_file_length_400
    - min_comment_ratio_20
    - mi_below_65
    - max_cyclomatic_complexity_10
supported_metrics:   (some entries were omitted)
    - comment_ratio
    - cyclomatic_complexity
    - maintainability_index
    - sloc
```

Listing 5.4: Plug-in manifest file in YAML syntax.

imposed on this file is that it must contain at least one of the entry point functions defined by HAROS in order to be executed. Listing 5.5 shows an excerpt of this script for the Radon plug-in. We can see how it implements the `file_analysis` function, which is the entry point for file by file analysis used by HAROS. This is where the execution of the plug-in starts. From that point on, the plug-in can organise itself in any number of additional functions or modules. In this case, the plug-in is self-contained, with only a few helper functions that call the various metrics measurement functions of the underlying tool. Throughout the code, we can see how the plug-in reports metrics and threshold violations to the HAROS interface, through the use of `report_metric` and `report_violation`.

```
1  from radon.visitors import ComplexityVisitor
2  from radon.metrics import h_visit, mi_compute
3  from radon.raw import analyze
4
5  # The plug-in's entry point; file by file analysis.
6  #   iface - the interface to communicate with HAROS
7  #   scope - a HAROS Source File (language should be Python)
8  def file_analysis(iface, scope):
9      with open(scope.path, "r") as f:
10         code = f.read()
11     cc = analyse_cc(iface, code)
12     lloc, ratio = analyse_raw_metrics(iface, code)
13     h = analyse_halstead_metrics(iface, code)
14     mi = mi_compute(h, cc, lloc, ratio)
15     iface.report_metric("maintainability_index", mi)
16     if mi < 20:
17         iface.report_violation("mi_below_20", "MI of " + str(mi))
18     if mi < 65:
19         iface.report_violation("mi_below_65", "MI of " + str(mi))
20
21 def analyse_cc(iface, code):
22     visitor = ComplexityVisitor.from_code(code)
23     for f in visitor.functions:
24         iface.report_metric("cyclomatic_complexity",
25             f.complexity, line=f.lineno, function=f.name, class_=f.classname)
26         if f.complexity > 10:
27             iface.report_violation("max_cyclomatic_complexity_10", str(f.complexity))
28         if f.complexity > 15:
29             iface.report_violation("max_cyclomatic_complexity_15", str(f.complexity))
30     return visitor.total_complexity
31
32 def analyse_raw_metrics(iface, code):
33     metrics = analyze(code)
34     iface.report_metric("lloc", metrics.lloc)
35     iface.report_metric("sloc", metrics.sloc)
36     # ...
37     return (metrics.lloc, ratio * 100)
38
39 def analyse_halstead_metrics(iface, code):
40     metrics = h_visit(code)
41     # ...
42     return metrics.volume
```

Listing 5.5: Excerpt of the HAROS plug-in for the Radon Python tool.

5.2 ROS Metamodel and Model Extraction

The previous overview of HAROS shows that, despite it being designed for the analysis of ROS systems, it did not incorporate much domain knowledge besides package structure, and did not perform any domain-specific analyses per se. As such, and considering our research on modelling ROS systems for domain-specific analysis (see Chapter 3), our first step to improve HAROS is to ensure that its internal data structures follow a rich and complete metamodel. To enhance it further, we then incorporate prototype model extraction components, based on the static analysis approach described in Section 3.3. This is our first technical contribution (as part of this thesis). It ultimately enables the ROS-specific analyses we aim for, and opens new possibilities for model-based plug-ins.

5.2.1 *Changes to the HAROS Analyser*

One of the most important changes to the analyser was to implement data structures with a direct correspondence to our metamodel, presented in Section 3.2. Most of the file system structures, such as Packages and Source Files, were already in place. Runtime entities, such as Nodes and Configurations, on the other hand, had to be introduced from scratch. The notion of a Project – which was previously just a collection of packages – had to be refined to accomodate Configurations. To promote reusability between applications and compatibility with the Visualiser, all metamodel entities can be exported in JSON format.

The workflow of the analyser is unchanged in many aspects. We have extended the tool's options and modes of operation to enable model extraction. As mentioned, the process of model extraction has been implemented in accordance with the algorithm presented in Section 3.3. If enabled, extraction happens during the setup stage, after the source code indexing takes place – i.e., once HAROS has already built the file system part of the model. At the time of writing, the extraction process is not complete, although it covers most of the common use cases. There are some elements of the extensive ROS API (e.g., provided by standard packages, such as `message_filters`, but not by the C++ or Python client libraries directly) that are not yet covered. We focused on implementing extractors for C++ source code, more common amongst critical and low-level code in ROS, whereas the Python extractors were an external contribution (more on that in Chapter 9).

For model extraction to take place, we inevitably require some user input. There is no fully automatic method to determine whether a launch file represents an application, or whether it should be used in conjunction with others. Since Configurations are part of our notion of Project, we extended HAROS' project files – the user input it already required to build Projects in the first place. Consider Fictibot as an example. If we wanted to add, for instance, a Configuration corresponding to `multiplexer.launch`, one of the various launch files found in the `fictibot_controller` package, we would add a *configurations* section to the respective project file, as seen in Listing 5.6. Configurations are identified by a unique name, followed by a list of launch files. The order of the launch files is important – the model extraction process interprets files in the given order, as *roslaunch* would if actually launching the files.

```yaml
project: Fictibot
packages: ["fictibot_drivers", "fictibot_controller", "fictibot_multiplex",
    "fictibot_msgs"]
configurations:
    multiplex:
        launch:
            - fictibot_controller/launch/multiplexer.launch
```

Listing 5.6: Minimal project file for Fictibot with configurations.

We reiterate how model extraction is not complete (especially when based on static analysis). Our proposed algorithm addresses this by allowing user-provided extraction hints. Hints can be partial (i.e., it is not necessary to specify the system, or any node, in full), and are also specified in the project file, under the respective Configuration. We follow the YAML syntax for hints defined in the metamodel (Chapter 3, Section 3.3). For the sake of an example, assume that we had to fix the message type of the Fictibot Controller's publisher on the `/controller_cmd` topic. Listing 5.7 shows the resulting project file.

```yaml
project: Fictibot
packages: ["fictibot_drivers", "fictibot_controller", "fictibot_multiplex",
    "fictibot_msgs"]
configurations:
    multiplex:
        launch:
            - fictibot_controller/launch/multiplexer.launch
        hints:
            nodes:
                /ficticontrol:
                    publishers:
                      - topic: "/controller_cmd"
                        msg_type: "std_msgs/Float64"
```

Listing 5.7: Project file for Fictibot with a configuration and extraction hints.

Parsing source code to extract a model can take some time, especially when parsing complex languages such as C++. To improve the overall performance of the extraction process, we enhanced HAROS with a (work in progress) built-in database of pre-parsed Nodes. It features Nodes that belong to well documented packages (e.g., `diagnostic_aggregator`, `robot_state_publisher`), released in the official ROS distributions. Another justification for this database, besides performance, is that ROS favours the reuse of components, and third-party packages are often installed from binaries. As a consequence, source code may not be always available for such common packages, which would require users to specify the missing nodes via hints. Embedding this domain knowledge in HAROS itself is another step towards the completeness of the extracted models. At the time of writing, this database contains primitives (such as calls to `advertise`, `subscribe`, etc.) from nodes belonging to 14 packages, as shown in Table 2. To clarify, we registered standard primitives (e.g., `advertise`) used by nodes that these packages make available (e.g., the `static_transform_publisher` node of the `tf` package). We did not register new primitives that the packages themselves may introduce, to be used directly in C++ (as is the case with the `message_filters` package and its `TimeSynchronizer`, for example). The database contains entries spanning all

Table 2: Built-in parsing database for HAROS.

Package	Nodes	Primitives
controller_manager	1	5
diagnostic_aggregator	1	9
hector_mapping	1	29
husky_base	1	9
interactive_marker_twist_server	1	4
joy	1	5
lms1xx	1	3
move_base	1	20
nodelet	1	4
prosilica_camera	1	6
robot_localization	1	12
robot_state_publisher	2	18
teleop_twist_joy	1	9
tf	1	1
Total	15	134

ROS distributions, for each package, from Indigo Igloo up to Melodic Morenia. Table 2 presents the number of unique entries; if a Node's interface does not change from one distribution to another, it is not counted.

At this point, we also introduce the possibility of specifying configuration-specific data for analysis plug-ins. Under each configuration, as shown in Listing 5.8, a subsection called **plugin_data** contains a mapping where keys are plug-in names (e.g., **plugin1**, **plugin2**), and the values can be of any type, including lists and other key-value mappings. The values are passed directly to the respective plug-in and have no semantics for HAROS itself.

```
1  project: Fictibot
2  packages: [fictibot_drivers, fictibot_controller, fictibot_multiplex, fictibot_msgs]
3  configurations:
4      multiplex:
5          launch: [fictibot_controller/launch/multiplexer.launch]
6          plugin_data:
7              plugin1: [arg1, arg2, arg3]
8              plugin2:
9                  arg1: ...
10                 arg2: ...
11                 arg3: ...
```

Listing 5.8: Project file for Fictibot with plug-in-specific input data.

While the reporting stage of HAROS is mostly the same (except that it has to export Configuration data), the analysis stage was improved to allow plug-ins to perform model-based analysis. Namely, we added a new plug-in entry point for Configurations (besides the existing ones for files and packages). Since Configurations are runtime entities, and constitute a new analysis scope, we also extended the plug-in

interface with a new reporting function, specifically to report issues on runtime entities. A practical example on how to make use of this new plug-in entry point is given in Section 5.3.

5.2.2 *Changes to the HAROS Visualiser*

Graphical data visualisation is arguably one of the most valuable tools for developers in terms of obtaining immediate feedback. This is one of the main driving principles behind the HAROS Visualiser. Having equipped the analyser with model extraction capabilities, the next logical step is to extend the Visualiser to display the extracted models. Graphical architecture models of ROS systems (or software systems in general) are useful for two purposes: *(i)* helping develpers validate the implementation; *(ii)* providing documentation assets for free.

We extended the HAROS Visualiser with a Computation Graph visualisation component that renders a graph of a selected Configuration, where each graph node is a ROS Resource and the edges are the ROS Links between them. This component features individual Resource inspection (e.g., message type and traceability to the source) and visibility settings for different Resources. As an example, Figure 40 shows the rendering of the *multiplex* Configuration from Listing 5.6. White circles represent Nodes, whereas green circles represent Topics.

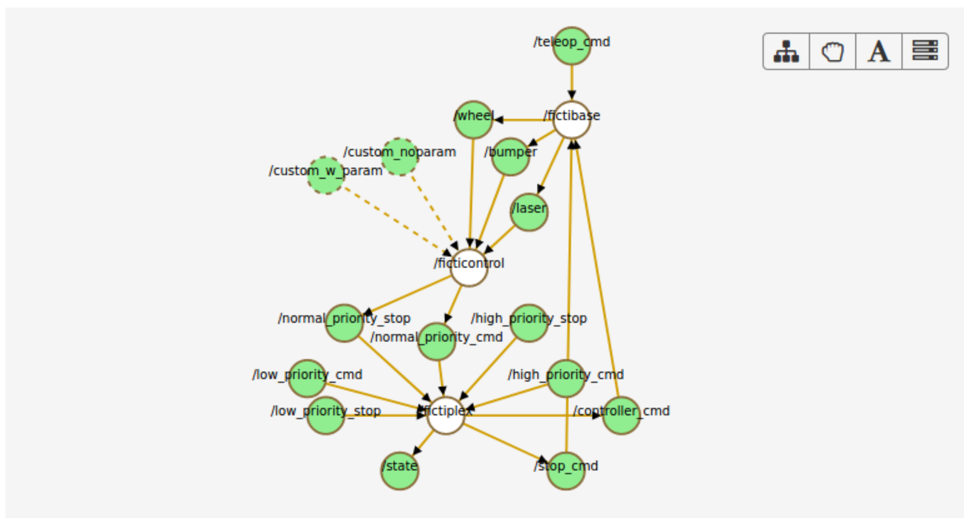


Figure 40: Display of a Computation Graph in the HAROS Visualiser.

One of the options of the Visualiser enables ROS parameters to be displayed in the graph. These are displayed as purple circles, and are disabled by default, since, in many cases, their numbers are overwhelming, making the graph unintelligible. Services are always displayed by default, just like topics, but with blue circles. This particular example does not feature any services, though. Figure 41 shows the same Configuration, but with parameters enabled and the `/ficticontrol` node selected. When a Resource is selected, only its direct links are drawn on the graph.

In the previous graphs we can see that two topics are rendered with dashed lines. Dashed lines represent conditional entities – entities that, depending on some conditions (e.g., an `if` statement) might not be

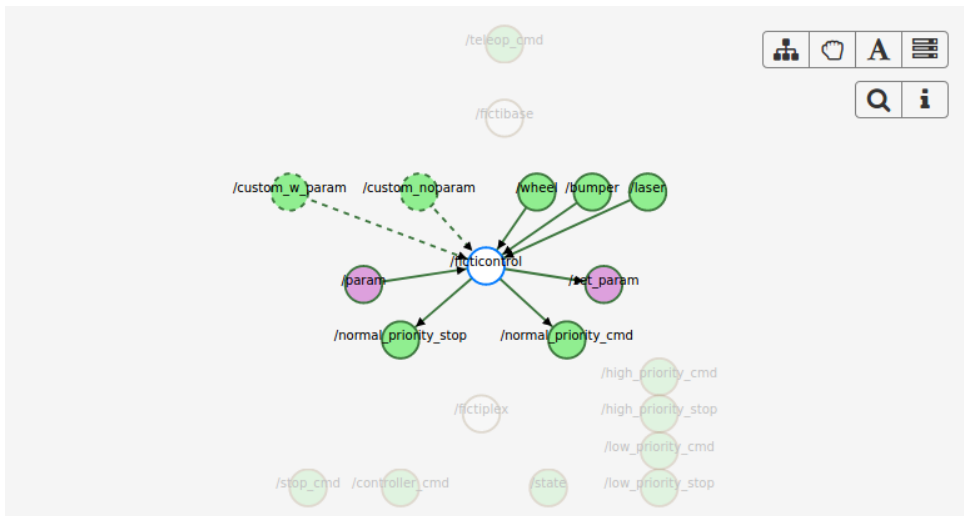


Figure 41: Computation Graph with parameters shown.

present in all instances of the Configuration. This is one of the advantages of using static analysis for model extraction. With dynamic analyses, determining conditional entities would require extensive instrumentation of all sources of code (i.e., instrumentation of traditional C++ and Python, but launch files too).

The Visualiser provides an information panel for every entity in the graph, where users can easily determine traceability details and which conditions affect the entity. Figure 42 shows the details of one of the topics, where we can see that it is affected by one condition, in the *random_controller.cpp* file.

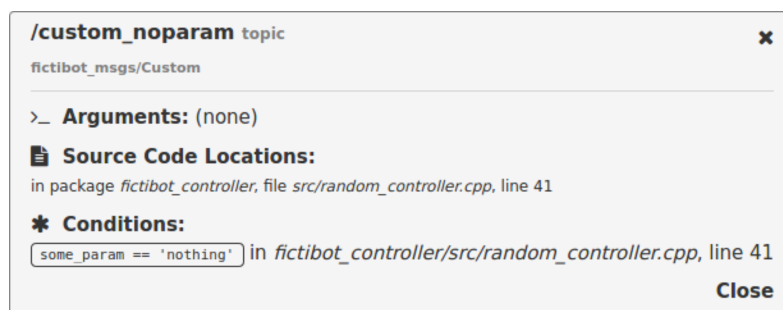


Figure 42: Resource details and traceability information.

The use of static analysis, as already discussed, sometimes faces limitations regarding name resolution. For highly dynamic values (e.g., read from parameters), it is impossible to fully determine the resulting ROS name of a topic, service or parameter. Our extraction process handles this by including the unresolved entities in the model regardless, and the Visualiser does so too. When a Resource name is unknown, it is represented with '`?`'. Since all names are resolved in the example from the previous figures, we altered the source code of the Fictibot Random Controller slightly, so that its `/stop_cmd` topic (remapped to `/normal_priority_stop` in this Configuration) becomes unresolved. As shown in Figure 43, when the unresolved topic is selected, the Visualiser performs pattern matching to find suitable candidates, ensuring that the message types match (highlighted in orange in the graph). One of the main benefits of this feature is to aid users in defining correct extraction hints, to refine the extracted models.

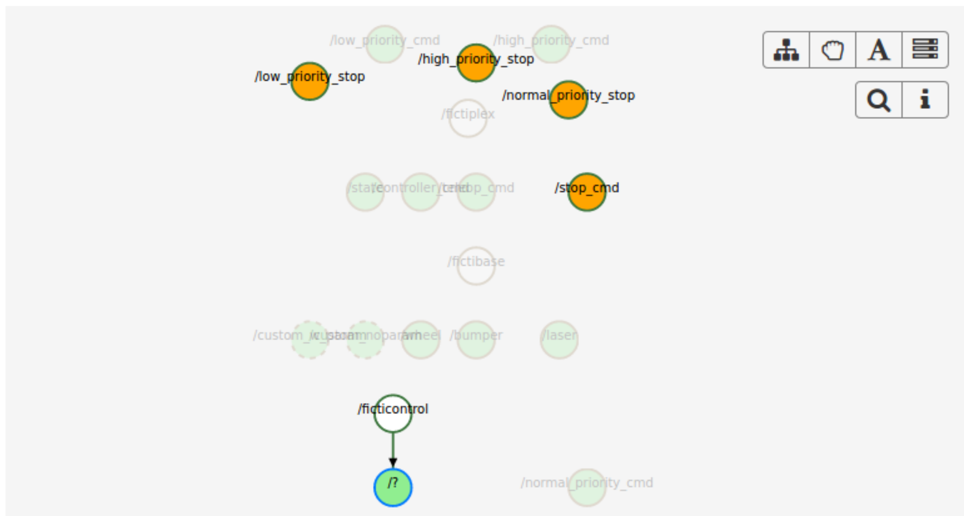


Figure 43: Pattern matching for unresolved Resources.

Lastly, since one of our additions to the analyser was a plug-in reporting feature specifically dedicated to runtime entities, it only makes sense that the Visualiser is able to display such issues. If there are available issues with a set of runtime entities to blame, users can select such issues from a drop down menu, and the graph is updated with red highlights on the entities. The highlights can be applied both to nodes and edges – i.e., to Resources or Links. Figure 44 in the next section illustrates this last feature.

5.3 A Simple Architectural Analysis Plug-in

The previous section describes how we enhanced HAROS, to equip it with model extraction capabilities. Such capabilities, coupled with a new entry point for plug-ins focusing on Configurations, play a key role in enabling ROS-specific analysis. New approaches, stemming from a model-based approach, become now feasible, whereas previously, when relying solely on pure static or dynamic analysis, they would require a lot more effort. In this section, we will build a proof-of-concept plug-in, to demonstrate how model extraction can be harnessed.

The intended plug-in is a query engine over the extracted Computation Graph to detect architectural issues. The idea behind it is not to impose (or to come up with) specific rules and checks, but rather to let users specify the rules they want to enforce over an architecture. A basic example of this (that is actually a common guideline in many systems) is to ensure that every topic has *at most one publisher*. This rule is quite reasonable because, without it, there is a risk of having two publishers flood a topic, without any means for the subscriber to tell which message belongs to which publisher. It is easy to see that, especially when handling sensor data, this would likely cause unexpected behaviour.

5.3.1 *The PyFlwor Query Engine*

Since the HAROS analyser (and, consequently, the interface to plug-ins) is implemented in Python, we adopted the *PyFlwor*[3] query engine as the basis for our plug-in. This is a Python query engine, with its own Python-like query language, that works over in-memory objects. It is a good fit for this purpose, since plug-ins handle HAROS models as Python objects too.

PyFlwor allows for two styles of queries, one based on XPath[4] (called *path expressions*) and another based on XQuery[5] (called *FLWR* expressions). Both styles of expressions operate on collections, and return sets of objects. Path expressions have a declarative flavour, whereas FLWR expressions are more programmatic.

The syntax for path expressions is relatively straightforward. Objects and collections are navigated using slash-separated identifiers, rather than the more common dot-separated variant seen in programming languages. For instance, given a Configuration as a scope, the path expression `topics` yields the set of all topics within the Configuration. The Python equivalent, assuming `c` as the Configuration, would be:

```
set(t for t in c.topics)
```

If we wanted the set of all message types used in a particular Configuration, then we would have to navigate one level further. The expression `topics/type` yields this set. Again, the Python equivalent would be:

```
set(t.type for t in c.topics)
```

It is possible to traverse an arbitrary depth of collections in this manner. For instance, to gather the set of all queue sizes used by all topic publishers, we would have to gather first the set of all topics, and then the set of all publishers, for all topics. This is given by the expression `topics/publishers/queue_size`, and by the Python equivalent of:

```
set(p.queue_size for t in c.topics for p in t.publishers)
```

A path expression can be refined with a filter (called *where condition*) and combined with various set operators (e.g., union, intersection and difference). A where condition is a Boolean expression within square brackets, following a path expression. For instance, to obtain all queue sizes greater than 10, one would write:

```
topics/publishers/queue_size[self > 10]
```

On the other hand, to obtain all queue sizes greater than 10 used by *either publishers or subscribers*, we would require a union:

3 https://github.com/timtadh/pyflwor

4 XPath (XML Path Language) is a non-XML syntax for selecting different parts of an XML document.

5 XQuery is a query language for structured or semi-structured XML data.

```
topics/publishers/queue_size[self > 10]
| topics/subscribers/queue_size[self > 10]
```

The FLWR acronym stands for *For-Let-Where-Return*, the four basic statements of this query style. FLWR expressions are more expressive than path expressions, and their syntax is closer to traditional programming. For instance, a possible encoding of the *"one publisher per topic"* rule, in the FLWR style, is given in Listing 5.9.

```
for t in <topics>
let pubs = <t/publishers>
where len(pubs) > 1
return t
```

Listing 5.9: Basic query to identify topics with multiple publishers.

A FLWR expression starts with a `for` statement, iterating over a collection of objects given by a path expression. An optional `let` statement follows, where additional variables can be defined. We defined `pubs`, to contain the set of all publishers for a topic `t`. The value of a variable can also be the result of a nested FLWR query. The third statement, an optional `where` statement, is used to define arbitrary conditions, to filter out the iterated entities or the variables defined with `let`. When a condition is satisfied, the `return` statement calculates the desired result for that iteration and adds it to the set of values that is to be returned at the end. As such, with an unsatisfiable `where` statement in place, the result of the query is always the empty set.

Since the FLWR style uses path expressions internally, we end up with a number of different ways to express the same queries. For instance, the previous query, of Listing 5.9, can be expressed entirely as a path expression,

```
topics[len(self.publishers) > 1]
```

Or as a trivial FLWR expression, where the result is essentially the same as the path expression by itself (Listing 5.10).

```
for t in <topics[len(self.publishers > 1)]>
return t
```

Listing 5.10: Basic query to identify topics with multiple publishers.

5.3.2 *Plug-in Implementation*

To implement a new HAROS plug-in, there are a few boilerplate steps we must follow. As previously mentioned, a plug-in is a Python package containing at least two files: `plugin.yaml` and `plugin.py`. The former is a manifest file, and the latter is the actual implementation. The manifest is minimal (Listing 5.11) and does not warrant much discussion. It declares the plug-in name and version, and tells HAROS that it

```yaml
name: haros_plugin_pyflwor
version: 0.1.0
rules:
    query:
        name: User-defined Query
        description: "A user-defined query found a match."
        tags:
            - code-standards
            - query
            - pyflwor
            - computation-graph
```

Listing 5.11: Plug-in manifest file for the Pyflwor plug-in.

should be reporting issues for a new rule (`query`). The rule's attributes are metadata to be displayed in the HAROS Visualiser.

For this plug-in, we want to annotate the various Configurations with their specific architectural queries. This can be done with the **`plugin_data`** section of the HAROS project file (the main source of input to the tool, shown in Listing 5.8), to be accessed later within the configuration object. As an example, Listing 5.12 shows an excerpt of the Fictibot project file, including a single query. The full source code for the project file is included in Appendix C.

```yaml
project: Fictibot
packages: [fictibot_drivers, fictibot_controller, fictibot_multiplex, fictibot_msgs]
configurations:
    multiplex:
        launch: [fictibot_controller/launch/multiplexer.launch]
        plugin_data:
            haros_plugin_pyflwor:
                - name: Query 1 - No Global ROS Names
                  details: "Found {n} Resources using global names - {entities}"
                  query: "topics/publishers[self.rosname.is_global] |
                          topics/subscribers[self.rosname.is_global] |
                          services/servers[self.rosname.is_global] |
                          services/clients[self.rosname.is_global] |
                          parameters/reads[self.rosname.is_global] |
                          parameters/writes[self.rosname.is_global]"
```

Listing 5.12: Project file for Fictibot with a configuration and Pyflwor queries.

Moving on to the plug-in itself, there is only one function that it is required to implement, `configuration_analysis`, which is the new plug-in entry point to analyse extracted Configurations, one by one. For each Configuration, all the plug-in has to do is to read the query list, execute each query, and report back any matches via the HAROS plug-in interface. Listing 5.13 shows the implementation for this function.

In Listing 5.13 we can see the use of some helper functions, namely `_setup` and `_report`. The former imports Pyflwor and performs its necessary setup, such as defining directories for generated grammars and assigning a number of Python built-in functions that should be available for the queries to use (e.g., **`len`** for sizes of collections). The `_report` function is little more than a wrapper for the reporting function of HAROS. It does some pre-processing of the query matches found by Pyflwor, since these may come in

```
1 def configuration_analysis(interface, configuration):
2     query_data = configuration.plugin_data.get("haros_plugin_pyflwor", ())
3     if query_data:
4         pyflwor = _setup()
5         for query_datum in query_data:
6             name = query_datum["name"]
7             details = query_datum.get("details", None)
8             query = query_datum["query"]
9             matches = pyflwor.execute(query, configuration, name=name)
10            _report(interface, matches, name=name, details=details)
```

Listing 5.13: Entry point function of the Pyflwor plug-in.

various return types. We identify ROS Resources and Links contained in the results, so that we can report them to HAROS and benefit from the highlighting feature in the Visualiser. Comments and white space aside, the plug-in can be implemented in under 100 lines of code. Appendix C contains the full source code for reference.

5.3.3 *Querying the Computation Graph*

In order to test our query engine plug-in, we implemented a small catalogue of queries, based on common practices in ROS development. We ran these queries over the previously mentioned *multiplex* Configuration from Listing 5.6. The Pyflwor syntax for all the presented queries is shown in Listing 5.14.

Query 1 *Are global ROS names used?*
Global ROS names (names beginning with a slash, e.g., '`/laser`') are unaffected by most name resolution rules of ROS. This requires additional attention when creating multiple instances of the same Node within a Configuration, as the global names will collide without proper remappings. For instance, if two Fictibot robots existed within the same network and they used global names to publish sensor readings, it is likely that each robot would receive sensor data from both. This situation leads to additional maintenance effort, thus justifying the query.

Query 2 *Are there any conditional publishers or subscribers?*
Conditional publishers and subscribers can be easily spotted in the Visualiser graphs (they use dashed lines). There is no justification for this query besides these constructs leading to an additional effort in understanding the architecture. In many cases, such conditional links stem from loops, where topics are subscribed to from a list parameter, and there is no significantly better alternative.

Query 3 *Do message types match on both ends?*
There is a type checking system for messages in ROS, but it is only active during runtime. When a message type mismatch occurs, a warning is given and messages of the wrong type are discarded. This lack of communication often has noticeable effects, but bringing this feedback to compile time is simple with our

query system, and is an additional step to reduce development time. The query is formulated for topics only, but implementing a similar type checking mechanism for services would be equally simple.

Query 4 *Are there any unbounded message queues?*
Message queue sizes should be carefully chosen in ROS. Avoiding unbounded message queues is a given, as they could use up all the available memory if a node does not process its queue fast enough.

Query 5 *Are there any message queues of size 1?*
Queues of size 1 are a very particular case with a niche application. They are relatively common in practice [149], but they can easily lead to message loss. Whether it is an intended effect should be analysed on a case by case basis. For instance, in teleoperation nodes, it could be intentional. We want nodes to process the last given command as fast as possible, and we count on the human operator to issue appropriate commands for the current situation. This is especially important for emergency stop commands.

Query 6 *Are there multiple publishers on a single topic?*
As previously mentioned, in most cases, having multiple publishers on a single topic is unintended, likely due to a missing name remapping. For pure data producers, such as sensors, merging multiple publishers under a single topic would likely lead to erroneous behaviour. For controllers, a single source of direct commands is also desirable, in order to avoid conflicts. This is why a common solution for multiple command sources is to introduce a multiplexer between the controllers and the actuator (as we do in Fictibot, and as other popular robots such as TurtleBot2 do). A valid use of multiple publishers on a single topic would be for sporadic (or high-level) events where the robot has multiple units capable of detecting, or producing, such events. In Fictibot, this would make sense if we had a sensor processor, that subscribed to all sensors and then published, e.g., obstacle events – which could be detected either from `/bumper` or `/laser`.

Query 7 *Are there any disconnected topics of the same type, with similar names?*
This query is slightly more complex than the others, in terms of implementation, but it shows how the query language allows for some programmatic freedom. We used some heuristics based on string comparison of the Topic names of publishers and subscribers, to try and detect those where the message types match, but the names are only slightly different – e.g., in `/laser` and `/robot/laser` the proper names match, but the namespace prefix does not. Many times, this could be an indicator that the developer applied a name remapping on one of the nodes, but forgot to match it in another node, or that they forgot to apply remappings altogether. Wrong name remappings are another issue in ROS that can be manually detected during runtime (via inspection), but for which there are no built-in compile-time checks of any kind.

Query 8 *Are there any uses of the message type* `std_msgs/Empty`*?*
The final query, asking whether the message type `std_msgs/Empty` is shown for illustrative purposes. We

know in advance that there are a few occurrences in Fictibot, and this query serves for the plug-in to report those topics (which we can later see in the Visualiser). Regardless, this is a message type that contains no data, and thus should be treated specially. It is useful as a trigger for events or actions, but one should consider whether additional data could be of use.

```
1  Are global ROS names used?
2 topics/publishers[self.rosname.is_global] |
      topics/subscribers[self.rosname.is_global] |
3 services/servers[self.rosname.is_global] | services/clients[self.rosname.is_global] |
4 parameters/reads[self.rosname.is_global] | parameters/writes[self.rosname.is_global]
5
6  Are there any conditional publishers or subscribers?
7 topics/publishers[len(self.conditions) > 0] |
8 topics/subscribers[len(self.conditions) > 0] |
9 services/servers[len(self.conditions) > 0] |
10 services/clients[len(self.conditions) > 0]
11
12  Do message types match on both ends?
13 for p in <nodes/publishers | nodes/subscribers>,
14     q in <nodes/publishers | nodes/subscribers>
15 where p.topic_name == q.topic_name and p.type != q.type
16 return p, q
17
18  Are there any unbounded message queues?
19 topics/publishers[self.queue_size == 0] | topics/subscribers[self.queue_size == 0]
20
21  Are there any message queues of size 1?
22 topics/publishers[self.queue_size == 1] | topics/subscribers[self.queue_size == 1]
23
24  Are there multiple publishers on a single topic?
25 topics[len(self.publishers) > 1]
26
27  Are there any disconnected topics of the same type, with similar names?
28 for pub in <nodes/publishers[self.topic.is_disconnected]>,
29     sub in <nodes/subscribers[self.topic.is_disconnected]>
30 where pub.type == sub.type and (pub.topic.id.endswith(sub.topic.name)
31     or sub.topic.id.endswith(pub.topic.name)
32     or pub.rosname.full == sub.rosname.full
33     or pub.rosname.full.endswith(sub.rosname.own)
34     or sub.rosname.full.endswith(pub.rosname.own)
35     or pub.rosname.given == sub.rosname.given)
36 return 'pub':pub, 'sub':sub
37
38  Are there any uses of the message type std_msgs/Empty?
39 topics[self.type == 'std_msgs/Empty']
```

Listing 5.14: Query catalogue to run over Fictibot's Computation Graph.

Query Results

The *multiplex* Configuration of Fictibot is a working configuration. Given that some of our queries are designed to catch bugs (e.g., query 3 for topic type checking), we do not expect many issues to be reported. But Fictibot's code has been designed, on purpose, to contain a few red flags, that other warning-level

queries should pick up (Figure 44). These red flags are mostly concentrated on the Fictibot Random Controller, as shown in Figure 45. Namely, we can see:

- an occurrence of one global ROS name (`/normal_priority_stop`);
- two topics under conditional statements (`/custom_noparam` and `custom_w_param`);
- a topic using an unbounded queue (`/normal_priority_stop`);
- a topic using a queue of size 1 (`/normal_priority_cmd`);
- four topics using messages of type `std_msgs/Empty` (the four `stop` topics around the Multiplexer).

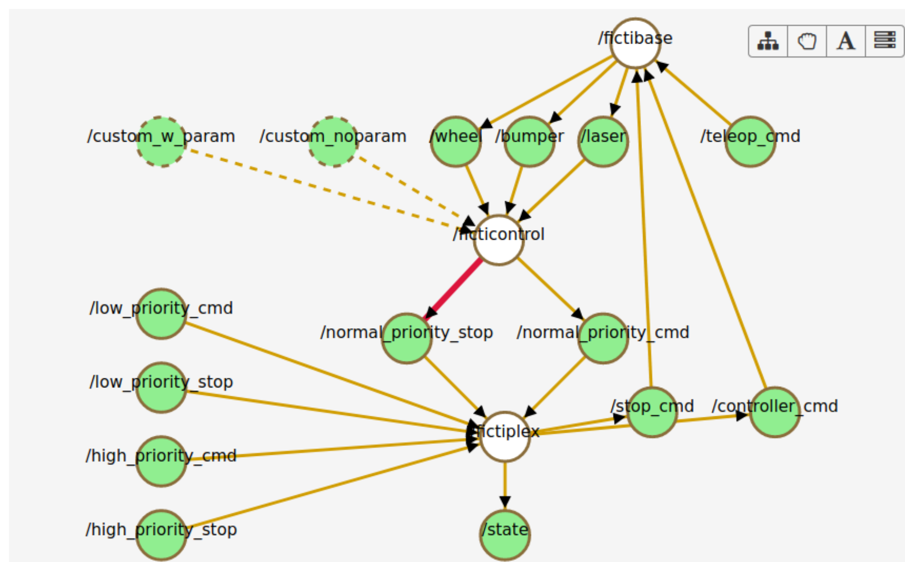


Figure 44: Computation Graph with query reports in the HAROS Visualiser. Highlighted is a Link with a queue of size 1.

HAROSviz
Dashboard Packages Issues Models Help

Runtime Issues | multiplex | Filter | Page 1/1

Issue #1 - Rule User-defined Query

A user-defined query found a match.

Found 1 Resources using global names - [Link of node '/ficticontrol' to topic '/normal_priority_stop' of type 'std_msgs/Empty']

code-standards query pyflwor computation-graph

Issue #2 - Rule User-defined Query

A user-defined query found a match.

Found [Link of node '/ficticontrol' to topic '/custom_noparam' of type 'fictibot_msgs/Custom'] under a conditional statement.

code-standards query pyflwor computation-graph

Issue #3 - Rule User-defined Query

A user-defined query found a match.

Found [Link of node '/ficticontrol' to topic '/custom_w_param' of type 'fictibot_msgs/Custom'] under a conditional statement.

code-standards query pyflwor computation-graph

Issue #4 - Rule User-defined Query

A user-defined query found a match.

Found 1 topics with infinite queues - [Link of node '/ficticontrol' to topic '/normal_priority_stop' of type 'std_msgs/Empty']

code-standards query pyflwor computation-graph

Issue #5 - Rule User-defined Query

A user-defined query found a match.

Found 1 topics with queues of size 1 - [Link of node '/ficticontrol' to topic '/normal_priority_cmd' of type 'std_msgs/Float64']

code-standards query pyflwor computation-graph

Issue #6 - Rule User-defined Query

A user-defined query found a match.

Found 1 topics of type std_msgs/Empty - [Topic('/stop_cmd', std_msgs/Empty)]

code-standards query pyflwor computation-graph

Issue #7 - Rule User-defined Query

A user-defined query found a match.

Found 1 topics of type std_msgs/Empty - [Topic('/high_priority_stop', std_msgs/Empty)]

code-standards query pyflwor computation-graph

Issue #8 - Rule User-defined Query

A user-defined query found a match.

Found 1 topics of type std_msgs/Empty - [Topic('/normal_priority_stop', std_msgs/Empty)]

code-standards query pyflwor computation-graph

Issue #9 - Rule User-defined Query

A user-defined query found a match.

Found 1 topics of type std_msgs/Empty - [Topic('/low_priority_stop', std_msgs/Empty)]

code-standards query pyflwor computation-graph

Back to Top

Figure 45: Query reports in the HAROS Visualiser.

5.4 Summary

In this chapter, we started by introducing HAROS, a plug-in driven framework for static analysis of ROS systems. Initially, this framework featured little domain knowledge of ROS systems, besides their package structure. Its plug-ins were also adaptations of analysis tools for general-purpose programming, useful for gathering quality metrics or checking compliance with certain coding conventions. Nonetheless, it had been used for a general quality study of the ROS ecosystem.

Our main technical contributions to HAROS were the implementations of our proposed metamodel and model extraction algorithm (presented in Chapter 3). This improvement changed how HAROS fundamentally handles a ROS application, and enabled new, model-based capabilities for analysis plug-ins. Most importantly, with our contribution, HAROS plug-ins are now able to perform any kind of analysis or processing of the ROS Computation Graph at compile time. To demonstrate such capabilities, we implemented a simple plug-in based on an existing Python query engine. This plug-in differs from traditional analysis tools by not hard-coding any rules, but rather allowing users to define new checks over the architecture of a ROS system.

6

ONLINE RUNTIME VERIFICATION OF ROS APPLICATIONS

One of the goals of this thesis is to design (and implement) property checking techniques, applied to the ROS domain, that could help us gather the necessary evidence to structure a dependability case. In the previous chapters, we paved the way for this goal by *(i)* establishing a metamodel for ROS applications; *(ii)* defining a language to annonate models with behavioural properties; and *(iii)* by enhancing an existing ROS analysis framework with model extraction capabilities. We have demonstrated, in Chapter 5, how architectural properties can be validated. In this chapter we address the challenge of providing evidence that a system's behaviour complies with its behavioural specification.

Up to this point, we built a foundation mostly on static analysis. To analyse how a ROS system behaves at runtime, we propose an approach based on dynamic analysis instead. Namely, we propose using Runtime Verification to detect violations of MFOTL formulae (restricted to the subset defined by our specification language in Chapter 4). While Runtime Verification is not a novel concept in the ROS ecosystem, our approach is the first to be both based on a high-level, pattern-oriented specification language, and on Metric First-Order Temporal Logic. It is integrated in the HAROS framework via a plug-in that generates the appropriate runtime monitors for a given Configuration and its behavioural specification. Furthermore, this Runtime Verification approach will provide a foundation for other techniques to build upon, as we will see in Chapter 7.

6.1 State of the Art

Runtime Verification is a verification technique that revolves around the use of *monitors* to observe a system's execution, in terms of its internal or external operations. As with any verification technique, the goal is to determine whether the system under observation satisfies or violates a given *correctness property*. The main advantages of this technique are that it is relatively lightweight, and that it acts on a system implementation, rather than a model.

The domain of Runtime Verification is a vast research topic. There are many approaches to it, and they vary wildly with regards to, e.g., how monitors observe a system, whether they interfere with said system, or how timely is their detection ability. Some taxonomies have been proposed to address this matter. We start this section by presenting the different types of Runtime Verification. Then, we discuss the existing

tools and algorithms that relate the most to our own approach. More concretely, we are interested in the state of the art in *Online Monitoring*.

6.1.1 *Taxonomies for Runtime Verification*

Cassar et al. [42] focus on the distinction between *Offline Monitoring* and *Online Monitoring*. They consider the former as its own category, and then spread Online Monitoring across a spectrum of *Completely-synchronous Monitoring (CS)*, *Synchronous Monitoring with Synchronous Instrumentation (SMSI)*, *Asynchronous Monitoring with Checkpoints (AMC)*, *Asynchronous Monitoring with Synchronous Detection (AMSD)* and *Completely-asynchronous Monitoring (CA)*.

Offline Monitoring The offline approach is completely decoupled and independent from the system under observation. Typically, an execution (i.e., a trace) of the system is recorded and persisted inside a data store – for instance, as a log file. The trace may contain a full execution of the system, or it can be partial, but either way it is finite. The monitoring solution then processes the stored trace and determines whether it satisfies the specified properties. One of the main advantages of this type of monitor is that it can view the system's execution as a whole, and it can traverse the trace multiple times, both forwards and backwards. The obvious price of such decoupling and flexibility is *late detection* – when a violation is detected, it is too late to act. Cassar et al. deem Offline Monitoring especially suitable to double check the behaviour of systems that already have a high correctness confidence.

Online Monitoring The online approach, as implied by its name, is the exact opposite of Offline Monitoring. A system is monitored during its execution – meaning that monitors have to run alongside the system (or as part of it). The most immediate advantage is that violations can be detected earlier, and, sometimes, monitors can even preempt the system from causing harm or entering an error state. This is essentially mandatory for safety-critical systems and for security purposes. In exchange, the monitoring algorithms tend to be more complex (the trace is observed incrementally) and the monitors themselves may impact the system's performance. Cassar et al. had in consideration the different levels of coupling between an online monitor and its monitored system, and categorised the existing approaches as follows.

CS Completely-synchronous monitors operate hand in hand with the system. In this tighly coupled and highly intrusive approach, the system as a whole blocks every time the monitors process an event. In the same fashion, when a violation is detected the whole system is affected (e.g., brought down). The inevitable performance overhead is the cost of having the earliest possible detection. This all-or-nothing stance is common in monolithic or embedded systems.

SMSI Synchronous Monitoring with Synchronous Instrumentation is a level only slightly less intrusive than Completely-synchronous Monitoring. The main difference between the two, is that SMSI only blocks the component whose event is being processed, letting other components execute freely.

AMC Asynchronous Monitoring with Checkpoints is the middle ground approach. Monitors are sufficiently decoupled for events to be processed asynchronously, but, after reaching a *checkpoint* (e.g., during a light load period), the system blocks, waiting for the monitors to catch up. With AMC, it is possible for users to strike a balance between timely detection (e.g., for safety-critical events), intrusiveness and performance overhead – provided that checkpoints are properly configured.

AMSD Asynchronous Monitoring with Synchronous Detection is another intermediate approach. It behaves mostly as a completely-asynchronous solution, but events that may directly contribute to a violation are marked as critical events. After a critical event is sent to the monitor, the associated component blocks, waiting for the monitor's verdict.

CA Completely-asynchronous monitors are the most efficient of all. They are loosely coupled, requiring only minimal instrumentation to listen to events. Event handling is done in the background, leaving the system free to proceed with its execution. The obvious price to pay for increased efficiency is late detection. As such, it is not an ideal solution for monitors to take preventive or mitigating measures upon detecting a violation. In this sense, it is similar to Offline Monitoring, with the main difference being the incremental monitoring nature of CA – it does not require a pre-recorded trace.

A more extensive taxonomy is proposed in [62]. Falcone et al. classify Runtime Verification tools not only in terms of Online versus Offline monitoring, but rather by taking into account seven parameters. Namely, they propose a taxonomy driven by: *specification* and *monitor* traits; *deployment* (implementation) of the monitor; how *traces* are modelled and observed; the monitor's *reaction* to violations; the level of *interference* with the monitored system; and, lastly, by the *application area* of the runtime verification tool.

Specification A specification encodes the properties that one wants to verify. It exists within the context of a system model, and is often designed prior to running the system itself. Falcone et al. distinguish specifications based on whether they are *implicit* or *explicit*. The former do not require user input, and could be used to capture general errors that lay beyond the compiler's ability, such as ensuring memory safety or the absence of deadlocks. The latter, provided by the user, is a formal encoding of functional or non-functional requirements of the system. Its main traits are: the expected *output* (Booleans, witnesses, streams); the encoding *paradigm* (declarative or operational); the supported *modalities* (current, past or future); how *data* is handled (propositional vs. parametric observations); and its concept of *time* (totally-ordered or partially-ordered logical time vs. discrete or dense physical time).

Monitor Monitors are among the main components of a Runtime Verification framework – they enable the observation of a running system and decide whether a property is satisfied or violated. Monitors are categorised by their *decision procedure*, *execution* and *generation*. Decisions can be analytical – by querying and scanning records – or operational – using automata or formula rewriting to reach a verdict. Furthermore, the procedures are distinguished by whether they are sound, complete or impartial, and whether they use behaviour prediction or anticipation. Monitors can be explicitly or implicitly generated,

and then directly executed or interpreted – based on whether they are implemented in their own piece of code, rather than being made up of parameters for a general monitoring infrastructure.

Deployment This category refers to how monitors are implemented in practice, and how and when they observe the target system. The deployment *stage* distinguishes whether the monitor acts offline or online. In the case of online monitoring, the authors distinguish only between synchronous and asynchronous monitoring. The *placement* of the monitor is said to be inline if the monitor runs as part of an *instrumented* system, or outline if it runs as a separate entity, with a dedicated means to receive observations. Deployment information also considers the *architecture* of the monitor, distinguishing between centralised architectures and decentralised ones.

Reaction A monitor's reaction to observations is either *active* or *passive*. The former is able to affect the monitored system directly, for instance by preventing violations from happening, or, if a violation happens, by taking recovery measures, raising exceptions or rolling the system back to a previous state. A passive monitor does not affect the running system. It is a simple observer, concerned with producing the designated specification output, various statistics or explanations for outputs (e.g., a witness trace for a violation).

Trace Traces are another key concept to Runtime Verification. Falcone et al. note that there is a distinction to be made between the trace *model* used in the specification, and the actual *observed* trace (i.e., the monitor's abstraction). A model could be infinite, but an observed trace is finite by definition. Monitors can sample the observed trace based on events or on time (e.g., periodic snapshots). Observations can be precise or imprecise. The *information* contained in a trace also varies. Some traces contain only events, while others can include internal system states, signals (for time-continuous information), and input-output operations. All monitored information represents an *evaluation* of the system's state, which could be relative to a point or to an interval.

Interference Interference is measured on a spectrum, between *invasive* and *non-invasive*. A monitor can never be completely non-invasive (even in offline monitoring), due to the overhead of instrumentation, to collect the trace. The degree of interference changes depending on multiple factors, but online monitors (especially with an active reaction) tend to be more invasive.

Application Area The application area of Runtime Verification mainly portraits its purpose. A few example areas include *collecting information*, *testing*, *debugging*, *failure prevention*, *recovery* and various *safety*, *security* and *liveness* analyses.

Francalanza et al. study Runtime Verification in the context of decentralised and distributed systems [68] (of which ROS is an example). When monitoring such systems, one has to face a few challenges that are not intrinsic to the simpler non-distributed programs. In particular, they emphasise the challenges of:

- *fault tolerance*, as many monitoring approaches do not consider the independent failures of different nodes;
- *global atomic observations*, which are hard to achieve when predicates can, potentially, refer to distinct parts of the system;
- *monitor orchestration*, i.e., how to optimally distribute monitors over a network; and
- *monitorability and correctness*, given that decentralised and distributed systems impose even more restrictions on runtime analysis than non-distributed systems (e.g., non-determinism).

As noted by Francalanza et al., there are multiple possible approaches to these problems. They review the state of the art on distributed monitoring, and classify existing approaches mostly based on *(i)* whether there is a single central monitor or the monitoring task is distributed; *(ii)* whether there is an assumption on a global clock; *(iii)* whether the monitoring system tolerates failures; and *(iv)* its intrusiveness. Regarding *(i)*, the matter of *monitor organisation*, their classification includes the following four classes.

Traditional Monitoring This is the simplest, most straightforward approach to monitoring. In this setup, there is a single, central monitor that observes the various processes scattered over the distributed system. Its main advantage is that it always reports a trace with a total ordering of events, despite the many possible interleavings of the monitored processes. However, it suffers from increased overhead, sub-optimal efficiency, worse tolerance to faults and weaker security.

Decentralised Monitoring The decentralised setup is the first step towards distribution, from traditional monitoring. The specification is broken down in multiple monitors, which may reside in different locations of the system, to perform local observations and build a single synchronised trace. Monitors are not required to interact among themselves, and they assume the existence of a global clock and a total order over all computations. The decomposition of a specification sometimes emerges naturally, when sub-properties pertain to different nodes.

Orchestrated Monitoring Orchestration is useful when more than one process resides in multiple locations of the system. In this setting, a trace is produced for every location in the system, based on local observations, without synchronisation or the assumption of a global clock. Monitors are central (i.e., one monitor per correctness property) and live in a remote location, where they gather and merge the local traces to produce a verdict. Centralisation trades scalability, in the form of increased communication overhead, and security, augmenting the risk of data exposure, for simplified monitor logic.

Choreographed Monitoring This setup is similar to Orchestrated Monitoring, in the sense that traces are produced at the location of the processes, based on local observations without synchronisation. The main difference is monitor distribution. Monitors are placed at the same locations as the processes, making direct observations over the local traces, but they are allowed to synchronise with each other to obtain remote information. This approach often requires less communication than the orchestrated

approach, having its main drawback in the complexity of the implementation and instrumentation of the monitored system, as well as being more intrusive in general.

6.1.2 *Runtime Verification Tools and Algorithms*

Numerous runtime verification tools and algorithms have been proposed, for all sorts of specification languages and application areas. In this section, we provide an overview of some of the most relevant approaches for our context – distributed, message-passing systems, preferably in the domains of ROS or robotics in general. Our goal is to employ runtime verification for the specification language we presented in Chapter 4, so there are a few attributes that are of particular interest to us, such as handling specifications with parametric data and physical time.

As previously mentioned, the research topic of runtime verification for distributed or decentralised systems is a challenging one. One notorious challenge is the possibility of network failures. Basin et al. address this issue [19] and propose the first monitoring algorithm that is able to tolerate such failures as well as being able to handle out-of-order messages. Their specification language is based on three-valued Metric Temporal Logic, which is able to handle time constraints, but is limited to propositional events.

Mostafa and Bonakdarpour propose the first runtime verification algorithm for asynchronous distributed programs that is both sound and complete [126], without assuming the existence of a global clock (a consequence of asynchrony). They base their specification language on three-valued LTL defined over the global state of the distributed program. A distributed program, in this context, is considered to be a set of asynchronous processes that communicate using message-passing primitives over reliable channels. A ROS system, for instance, would qualify for this definition. The algorithm is able to verify temporal properties beyond safety predicates, and is inspired by *distributed computation slicing* – the technique of abstracting distributed computations with respect to a given predicate. It is fully decentralised, in the sense that each process maintains a replica of a monitoring automaton (an intrusive approach). Each monitor deals with concurrent events by maintaining a set of all possible verdicts for a given predicate. This approach was evaluated on a simulated swarm of flying drones, and the authors found that the monitoring overhead grows only in the linear order of the number of processes and events.

DecentMon [21] is a monitoring tool for LTL specifications over decentralised systems, such as circuits and embedded systems, where there are multiple parallel components working in synchrony (i.e., a global clock can be assumed). DecentMon takes as input multiple traces of the system and an LTL formula. Formulae are then monitored in two different modes:

1. traditional monitoring, where a central monitor merges the input traces into a single, global trace; or
2. decentralised monitoring, where each trace is read by an individual monitor.

ARTiMon [145] is a monitoring tool that specialises in handling time. It was historically developed for online validation of Matlab and Simulink models, but it evolved to more general applications. In ARTiMon, the monitored system is viewed as a set of state variables and a time domain (either discrete or continuous). It interprets all terms of its input language as partial time functions over the time domain. The specification

language itself includes a few interesting features, such as temporal operators with interval bounds, the ability to aggregate values, and the ability to refer to the last observed value or the previous value for a given event. One of its drawbacks is that the specification language only considers primitive data types. Complex data types are not supported, and, thus, other features such as quantification are also left aside. ARTiMon monitors are implicitly generated and of interpreted execution.

While ARTiMon specialises in time, *DejaVu* [81] focuses more on data. It is a monitoring tool for First-Order Temporal Logic specifications with a past modality over events that carry data, that uses Binary Decision Diagrams (BDDs) to represent and manipulate observed data. This data structure comes with a few advantages, namely in terms of highly compact data representations and being highly efficient for a number of operations. The authors note that the monitor construction procedure for the propositional case naturally extends to BDDs, which comes as an added benefit for this choice.

R2U2 [153] is a runtime verification system specifically designed to monitor and diagnose security threats for unmanned aerial vehicles (UAVs). In safety-critical systems, such as UAVs, both software and hardware present potential threats and faulty components. Thus, R2U2 is proposed as a solution to verify hardware safety and software security properties, by monitoring system health and finding possible attack patterns. It is not generally applicable in other contexts, as it is implemented on its own hardware, as a hardware and software bus traffic observer. Despite this major drawback, this design decision achieves extra protection against attacks, and reduces obtrusiveness to a minimum.

Kane et al. present a case study on online monitoring of an autonomous research vehicle (ARV) system [92]. Their work addresses an important problem in safety-critical systems – that, often, these systems feature a number of black-box, commercial-off-the-shelf components. Without full access to the source code, runtime verification approaches based on instrumentation become unfeasible. Furthermore, ARV systems tend to be hard real-time, in which case instrumentation is neither available nor without an impact in performace. They propose a passive monitoring algorithm, called *EgMon*, that observes exchanged messages over a Controller Area Network (CAN) – a standard broadcast bus for ground vehicles. Properties are written in future-bounded, propositional Metric Temporal Logic, in order to handle timing related properties (which they deem to be common in this context[1]). The EgMon algorithm incrementally takes as input a system state (defined as a mapping $\mathit{Proposition} \mapsto \{\top, \bot\}$) and a MTL formula and eagerly checks the state trace for violations. EgMon's eager approach, based on dynamic programming techniques, aims to reduce the input formula as soon as possible using history summarising structures and formula-rewriting.

TeSSLa [51] takes on a different approach regarding runtime verification. Whereas most tools and specification languages are event based, TeSSLa focuses on stream based runtime verification. Stream based verification, as implied by the name, treats input data as continuous streams, or *signals* (although TeSSLa also handles sparse, event-based data under the same formalism). In cyber-physical systems, this is a common notion that nicely captures, e.g., very frequent sensor readings. In essence, TeSSLa is just a mechanism to transform input streams into output streams. Specifications define stream transformations,

1 As an example, the authors note that many properties in ARVs follow the pattern *"the system must perform action α within τ seconds of event ε"*. This could be captured by our specification language as "**globally:** ε **causes** α **within** $(\tau \times 1000)$ ms".

using standard operators, aggregation and more. Users are allowed to define any number of intermediate streams, to allow for more complex transformations. This means that TeSSLa does not output a single verdict, but rather it outputs data streams of any type. Obviously, it is possible to stream the value of a verdict over time, but it also enables, for instance, statistics based on system observation. TeSSLa employs non-intrusive monitoring, based on text input traces, or instrumented programs. The packaged tool suite supports software instrumentation for C programs. Other interesting features include the automatic timestamping of stream events (enabling reasoning for real time properties) and self-references to past values within a stream.

Hallé and Villemaire studied the problem of monitoring message-based workflows where messages contain data [79]. More specifically, they propose an approach for online monitoring of XML messages that, like the DejaVu tool, puts some emphasis in handling parametric events. Properties are expressed in LTL-FO$^+$, an LTL extension with First-Order quantification over message data. One of the main innovative aspects of their algorithm is that it works *"on-the-fly"* – i.e., unlike most other algorithms, it does not compute an automaton nor does it store any previous messages, it works only with the current state. In a follow-up work [80], the authors address some recurring issues of message-based workflows, namely: *(i)* the inability to detect lost and out-of-sequence messages; and *(ii)* the unrealizability of some messaging protocols. They propose a monitor-based messenger, which is essentially relegating message exchanges to a central monitor that stamps messages (with a time stamp, thus ensuring order) and enforces a protocol (the monitored properties).

Erlang[2] [13] is an industrial-strength programming language and environment designed for highly-concurrent, distributed programs. Programs follow an actor model, where a number of lightweight processes (the actors) communicate via message-passing – somewhat similar to nodes in a ROS environment, although much more scalable. Francalanza and Seychell studied the automated synthesis of concurrent monitors and proposed an algorithm for this end, that they prove to be correct [67]. The implementation of this algorithm, the *DetectEr* tool, provides concurrent runtime monitors for Erlang systems. Properties are specified in a subset of Hennessy–Milner Logic with recursion (HML or μHML). μHML is a branching-time logic to write properties about the computational graph of programs, and the adopted subset is, essentially, a reformulation of the modal μ-calculus that is limited to reactive (safety) properties. Monitors mostly observe the system's interactions with the environment in an asynchronous fashion, resorting to the Erlang virtual machine's tracing mechanisms, which is a natural choice for the actor communication model in a distributed setting. A follow-up work [15], from Attard and Francalanza, also proposes an algorithm and prototype tool for the synthesis of distributed monitors in Erlang, based on the same μHML formalism. One of the major differences from [67] is that they extend the specification language to a larger subset, called MHML (Monitorable Hennessy–Milner Logic).

In their PhD thesis [29], Yoann Blein addresses the issue of writing correct specifications for runtime verification. Many engineers operating safety-critical systems (such as the medical devices used in their case study) might not have the necessary background in formal methods to comfortably write properties in LTL and other, more complex logics. As such, one of their goals is to define a user-friendly, human-readable

2 https://www.erlang.org/

language to specify properties over parametric events, and then convert such specifications to runtime monitors. This language (and corresponding tools), called *ParTraP*, is based on Dwyer et al.'s specification patterns [58, 59] – much like our proposed specification language (see Chapter 4). Compared to our proposal, ParTraP is more feature rich. It allows nested scopes, a greater range of operators and even arbitrary Python expressions over event data. However, it is designed with their industrial case study in mind. In this case study, traces are obtained in an offline fashion, meaning that the runtime monitors also work offline. Offline monitoring has greater freedom when it comes to going back and forth over a trace's events, which enables and facilitates the implementation of, e.g., nested scopes. Despite being designed for untrained software engineers, the language has not been properly evaluated in terms of its approachability. As is the case with many other tools, it only produces a verdict (not a counterexample) when violations are found.

Adam et al. propose DeRoS [4, 5], another specification language and runtime verification approach, this time targeting ROS systems. It is not clear which temporal logic is used to define the semantics of the language, as it is used not only to specify safety properties but also the system architecture (i.e., the computation graph). Specifications are parametric regarding ROS messages, and are able to handle physical time. Messages (and topics) are treated akin to streams, enabling users to refer, e.g., to the event of a formula changing from being evaluated as $\bot$ to $\top$ and vice-versa, or to refer to the event of a formula holding for τ seconds. The monitor implementation, which is not publicly accessible, is based on a central, online monitor that takes an active stance. Specifications can include actions for the monitor to take when a certain formula (here called *rule*) holds. When compared to our proposed language, DeRoS lacks a number of features. For instance:

- it cannot express scopes (e.g., '`after /state`');
- it cannot express Absence patterns (e.g., '`no /velocity`');
- it cannot express Response patterns (e.g., '`/bumper causes /stop within 100 ms`');
- it cannot express Precedence patterns (e.g., '`/stop requires /bumper within 100 ms`').

DeRoS has been used as a reference to evaluate the work of Lesire et al. [106], in which they propose a runtime monitoring approach for Past-time LTL (PtLTL) specifications with physical time constraints. The argument for including only Past-time operators is that one cannot predict the future, and, thus, including future modal operators is useless (in terms of online monitoring). While true, this decision makes it harder to specify bounded liveness properties. Their monitor synthesis implementation extends beyond ROS, to the domains of Orocos[3] [37], MAUVE [105], G^{en}oM[4] [65] and SmartMDSD[5] [160]. This approach divides monitors (here called *observers*) into atomic observers (for the PtLTL subset) and high-level observers (for the real time constraints). Atomic observers observe data on ports (the equivalent to a ROS topic) over the last n states (messages). These observers are then combined into the high-level timed observers with classical and timed operators. The monitoring process is based on the last rise and fall times of each formula (the moments when the formula's evaluation changes to $\top$ or $\bot$, respectively). The proposed

3 https://www.orocos.org/
4 https://www.openrobots.org/wiki/genom
5 https://wiki.servicerobotik-ulm.de/smartmdsd-toolchain:start

specification language allows for some degree of freedom – instead of providing a limited set of operators, users can define custom predicates, which ultimately have to be manually implemented.

Manual implementation is also one of the key ingredients of ROSRV [85], a runtime verification approach for ROS based on the principles of Monitor Oriented Programming (MOP). Specifications take the form of event handlers associated to ROS topics, so that they are triggered on each incoming message. The body of each specification is operational, being essentially composed of valid ROS C++ code. Users can enable MOP logic plug-ins, to write formulae and patterns in said logic that help detect sequences of events (e.g., in LTL). While there is no direct support for timed properties, since there is some programmatic freedom given to the user, users can implement it on their own. As for monitor deployment, ROSRV takes a somewhat intrusive approach, in the sense that it replaces the original ROS Master with its own version. The regular ROS nodes, however, work normally, unaware of being monitored – monitors are placed as men-in-the-middle for all communications. Granted, despite the nodes being unaware of the monitoring process, monitors can take an active stance, for instance to drop messages that would lead to a critical system state. This approach also opens up room to implement some security measures, such as access control policies. It is possible to load the ROSRV Master node with such policies to ensure that, for example, only trusted nodes can publish on a given topic.

Hu et al. propose another approach to monitor temporal properties of ROS systems, called *RMoM)* [84], but they specialise in the context of robot swarms. Their approach provides a domain-specific language to write specifications in, based on three-valued MTL semantics. Properties have access to the typical logical operators, as well as a range of built-in functions that are organised in a hierarchy of four layers:

1. functions over system resources, for instance to assess CPU usage, memory consumption and availability of cameras and other sensors;
2. functions over ROS communications, for instance to retrieve node names, topic names or message types;
3. functions over a single robotic system, for instance to retrieve the robot's position and current speed;
4. functions over the swarm as a whole, for instance to retrieve the total number of robots in the swarm or to determine its current formation.

To cover the case where the given functions might not be expressive enough, users are given the ability to define and implement custom functions over each of these layers. The authors also extend the language with new logical operators to state, for instance that all robots in the swarm satisfy a given property, or that a specific robot within the swarm satisfies a property.

Lastly, Basin et al. propose MonPoly [18, 20], a runtime verification tool for online and offline monitoring that was designed to verify Metric First-Order Temporal Logic formulae. MonPoly takes three files as input: *(i)* a signature file that declares all types of monitorable events; *(ii)* a formula file that contains the MFOTL formula to monitor; and *(iii)* a log file containing a trace of events to monitor. By default, the monitoring takes place in an offline fashion. By replacing the log file with a stream, the monitoring process becomes an asynchronous online one.

An event is defined by a name, at minimum, and may optionally contain parameters of primitive types, such as integers, floats or strings. For instance, the following syntax declares two types of events, `a` and `b`, where the former has two parameters (an integer and a floating-point number) and the latter has none.

```
a(int, float)
b()
```

The log file consists of a sequence of ordered time-stamped events, where the timestamps are non-decreasing, and each timestamp may have multiple associated events. For instance, the following syntax shows a trace with two instants where the first instant, at time `100` registers two events, `a` and `b`, and the second instant (at time `200`) registers a third event, also an `a` event but with different arguments.

```
@100 a(1, 0.5) b()
@200 a(2, 0.8)
```

The formula file, as stated, consists of the monitored MFOTL formula, possibly containing aggregation operators such as sums or averages. However, not all MFOTL formulae are amenable to Runtime Verification. The authors focus on a safety fragment of the logic that only accepts formulae of the form $\Box\forall\overline{x} : \varphi$, such that φ does not contain unbounded future operators. Subformulae such as $\Diamond\psi$ or $\Diamond_{[a,\infty)}\psi$ are not allowed. In practice, MonPoly negates the input formula and searches the event log for a match; i.e., a counterexample to the original formula.

6.2 The Need for a New Runtime Algorithm

We did not settle with one of the many existing tools because, in general, they lack at least one of the features we require for our specification language. For instance, some are unable to handle events with data [19, 21, 92], some lack real-time constraints [79, 85], some lack cross-event references arbitrarily in the past [5, 84, 145], and some lack high-level data structures (e.g., lists) with quantification [51, 145]. There are two special cases, however, that deserve further discussion.

6.2.1 *Online vs. Offline Monitoring*

ParTraP [29] is based on a specification language that is similar to our own proposal, but it is intended for offline monitoring. In this thesis, we are interested in solving the problem of online monitoring for ROS systems, so direct reuse of Blein's work is impossible.

As previously discussed, online monitoring allows for earlier detection of failures, by running the monitors in tandem with the monitored system. This is an advantage in itself. Moreover, online monitors can also serve as building blocks for other property checking techniques to build upon. For example, in Chapter 7 we show how to approach Property-based Testing in the context of ROS, from high-level specifications, by relying on the Runtime Verification approach we present in this chapter to provide the test oracles.

For the purposes of testing, both online and offline monitoring work as a foundation, in theory. Not only that, but offline monitoring is far easier to implement. The idea behind it is rather simple – we let the system run for the full duration of each test case, while recording a message log. Then, we feed the log to the offline monitor to produce a verdict that, in turn, will dictate the outcome of the test case. But, despite its simplicity, this approach is subpar in terms of time and space efficiency.

For testing to be effective, in general (and for Property-based Testing in particular), we want to run tests for thousands of inputs, if not millions. Naturally, an offline monitoring approach can only detect violations after an example has run its entire course, and this is bound to increase the overall time required to test a property significantly. For instance, take the property '`globally: no /a within 1000 ms`'. Suppose that a faulty system publishes `/a` after 500 milliseconds. An offline monitor will always take, at least, 1 second to report a faulty test, while an online monitor is capable of doing the same in half that time. Note that this is not limited to property violations, the converse is also true. Online monitors are capable of detecting early success in bounded liveness properties, e.g., '`globally: some /a within 1000 ms`', while offline monitors cannot. Over a large number of test cases, this deficit of time efficiency adds up.

On the other hand, the need to record all exchanged messages over a relatively long period in a message log entails a waste of space. Messages containing images, point clouds or odometry information, for instance, are known to be large, in the order of kilobytes or even megabytes. If nodes publish such messages, over several topics, at a rate of tens of messages per second, we can easily obtain message logs with hundreds of megabytes. Online monitors are able to optimise this aspect in many cases. For properties such as '`globally: no /a within 1000 ms`', for instance, there is no need to store any messages at all. The monitor can simply report a failure as soon as it observes the a message on `/a` within the first second of execution.

Lastly, online monitors can also be used as core components of the system itself. Developers are able to build, e.g., safety controllers that react with an appropriate action, such as stopping the robot or slowing down its movement, when a monitor reports a property violation. In fact, this concept could be pushed to the extreme, where monitors are used not only to detect proper failures, but also to detect a number of other interesting scenarios (e.g., the robot reaching a certain location). Any number of controllers can then be built on this foundation. In practice, however, it is likely that this extreme approach is not as performant as embedding the detection of such scenarios in the client nodes, as roboticists already do. Alternatively, developers can extend the monitor implementation with the desired actions, instead of using it simply to report events.

6.2.2 *General MFOTL Monitoring vs. Custom Implementation*

The second tool worth further discussion is MonPoly [18, 20]. The semantics of our specification language are based on MFOTL, which makes MonPoly very relevant to our work. There are only a few issues preventing us from adopting it as a monitoring tool.

The first issue relates to the monitored formulae. In theory, we could translate our properties into the equivalent MFOTL formulae, and feed them to MonPoly for online monitoring, but this is not trivial. The

first step is to rewrite all formulae in our language's semantics so that they match the $\Box\forall\overline{x} : \varphi$ form. But, even in this form, some formulae contain unbounded future operators that MonPoly is unable to handle. This, right from the start, would leave out the entire '`after p until q`' scope, among other properties. The workaround is to bound the future operators with a very long timeout. The drawbacks are that the final formula is no longer logically equivalent to the original, and that, due to the large operator time bound, MonPoly might delay the verdict until the last log entry is read.

The second issue is related to event logs. To account for being a ROS-agnostic tool, we have to translate ROS messages into the event syntax that MonPoly uses, but this is also not straightforward. ROS messages can be arbitrarily composed, and may also contain array fields. Composite messages are relatively easy to handle – with a recursive traversal over the message types, we can *flatten* the fields until we are left only with primitive types. For instance, `geometry_msgs/Twist` is a velocity message with two three-dimentional vectors, `linear` and `angular`. Instead of declaring an event type with the two parameters, `linear` and `angular`, we could declare an event with six parameters, `linear_x`, `linear_y`, and so on.

```
twist(float, float, float, float, float, float)
```

Handling arrays is not so linear. Recall that arrays in ROS can either be of fixed length or of variable length. For fixed length arrays we can simply declare all fields up front, i.e., `array_0`, `array_1`, etc., as if the array was merely another nested message with primitive fields. For variable-length arrays, however, it is harder to figure out a workaround, given that MonPoly's first order operators act only on the domains of its primitive types (e.g., quantification over integers). Lists are not supported.

One way to handle variable-length arrays is to perform static analysis over the monitored properties, and save only those indexes that are directly referenced in event predicates. For instance, for the event '`/p {array[0] > 0}`', we only care about the value of the index 0 of the array. But this solution falls short if the predicates quantify over the array indexes, e.g., '`/p {forall x in array: x > 0}`'. Then, the whole array would have to be included, and there is no way of knowing the length of the array up front. An alternative approach is to declare an event with a single argument, e.g., '`p_array(int)`', and create an event per index of the message's array field. We can feed all events to MonPoly under the same timestamp, to ensure that they are treated as simultaneous events. The drawback is that this solution implies a more complex property translation; converting event predicates from our syntax to MonPoly formulae is no longer a straightforward rewrite operation.

A third issue regarding the adoption of MonPoly is related to its monitoring algorithm, or, rather, how MonPoly evaluates event logs. MonPoly is only able to evaluate a formula, at a given time instant, once it has complete information about that timestamp, i.e., once it reaches the end of the event log, or once it reads a new timestamp. Since multiple events are allowed under the same timestamp, in an online monitoring setting, where events are fed one at a time, MonPoly will always have a slightly late detection. Only when a new event is read, at a later time, can MonPoly evaluate the current events. For instance, in the small event log shown before, even if the event `a(1, 0.5)` at time `100` constitutes a violation, such violation will only be reported after MonPoly reads `a(2, 0.8)` at time 200. Until then, we could, potentially, continue feeding more events under the timestamp `100`, such as `b()`.

Also related to the theme of late detection is MonPoly's focus on safety (and co-safety) properties. As MonPoly negates the input formula and looks for a counterexample, it can only focus on detecting violations. For properties such as '`globally: some /b within 1000 ms`', MonPoly will always take one second to produce a verdict: after one second, either `b()` was observed or it was not. There is no reliable way to detect early success. The workaround is to run two monitors, one that looks for a violation, as normal – i.e., verifying $\Box\forall\overline{x} : \Diamond_{[0,1000)}\mathtt{b}$ – and another that does not negate its input formula – i.e., one that treats the desired event, `b()` as the violation and verifies $\Box\forall\overline{x} : \Box_{[0,1000)}\neg\mathtt{b}$. Once a `b()` event is detected by the second monitor, both monitors can be safely interrupted.

Finally, there are smaller, technical obstacles for the direct adoption of MonPoly. The most immediate one is that the tool is implemented in OCaml, and ROS does not provide any official bindings for the OCaml language. We would have to resort to unofficial bindings, or use the tool as a child process of a ROS node. Inter-process communication is, naturally, slower than using a single process, which further contributes to detection latency.

Weighting all these factors, we have decided not to adopt ParTraP or MonPoly but, rather, to implement our own Runtime Verification algorithm, tailored for our specification language. Regardless, we did put MonPoly to the test, and compared its performance against the approach we propose in this chapter. This evaluation can be found in Chapter 8, Section 8.3.

6.3 Approach Overview

We have seen how, according to Falcone et al.'s taxonomy [62], a Runtime Verification approach can be categorised with respect to seven parameters, namely: the specification, the monitor, the deployment process, the monitor's reaction, the monitored trace, monitor interference, and application area. We are able to identify our stance, right away, in terms of application area, property specification, monitored traces and the monitor's reaction.

Application A Runtime Verification solution can have many potential application areas. In general, one could argue that the main application of our approach is *debugging*. In Chapter 7, we show how our approach can be used for *testing* too.

Specification We use *explicit specifications*, seeing as we provide a *declarative language* tailored for this purpose. This language supports past, current and future observation modalities; handles complex, parametric events with quantification; and is able to express logical time constraints with total order, and physical time constraints with discrete time. In terms of output, the specification and monitors themselves aim to produce only a *verdict*. Within the larger scope of Property-based Testing (see Chapter 7), for instance, we can add a *witness* as well.

Trace We presented a specification language based on *infinite trace models*. During execution, *trace observation* should be *event triggered* (as we are dealing with the observation of ROS messages) and

precise – monitors observe every new event of interest. *Trace evaluation* should be done at specific points, such as when a new message is observed or when a timer expires.

Reaction Monitors, regardless of their implementation, should have a *passive reaction*, i.e., they should be limited to reporting verdicts. Our specification language does not allow the encoding of actions, unlike, e.g., DeRoS. Moreover, a passive reaction is the most flexible approach to use runtime monitoring as a building block for other purposes. For example, consider the testing approach we present in Chapter 7. If the runtime monitors were allowed an active reaction, it could possibly interfere with the system's behaviour and, consequently, with the test results.

The other parameters essentially dictate how we will approach the monitor implementation, and will be discussed in the remainder of this section.

6.3.1 *Generation and Execution*

The first design problem we face regarding runtime monitors is how to generate them. The taxonomy offers two possibilities, *explicit generation* or *implicit generation*.

Design Problem 1. *Should monitors be explicitly generated from the specification, or should they exist implicitly as part of a general monitoring system?*

Another way to put this question is to ask whether we want to have a tailor-made monitor for each property, or whether we want to implement a generic monitoring solution, able to handle all types of properties of our specification language. MonPoly, as we have seen, is an example of the latter. It exists as a generic monitoring tool for a subset of Metric First-Order Temporal Logic.

In our case, we believe that explicit generation is the better option. We have a limited specification language, with a number of patterns that we know ahead of time. As such, we can design a custom monitor for each of the available patterns. Then, we can optimise each type of monitor individually. This is one of the aspects MonPoly lacked in our previous analysis. For some formulae, MonPoly has no option but to wait until the end of the trace (or the end of the specified temporal bound) to produce a verdict. There is no reliable way to detect early success of an Existence property, for instance. By building a monitor that is tailor-made for a given type of property, we can benefit from the earliest possible detection.

One of the disadvantages of working with a domain-specific specification language is its reduced maintainability. The generation process is tied to this language. If the language evolves, to address issues or to introduce new features, such changes could have an impact in the whole generation process.

Regardless of how monitors are generated, deciding how monitors are executed is also a binary problem.

Design Problem 2. *Should monitors have their own piece of code to execute (direct execution) or should a generic algorithm be parameterised to handle individual properties (interpreted execution)?*

Monitor execution sounds as a problem that is related to monitor generation, but the two are actually independent problems. We have decided to explicitly generate monitors, but we have not yet decided

on what a monitor is. In other words, we have to decide what is the output of the generation process. A monitor could be a set of parameters for a generic algorithm (e.g., the states and transitions of an automaton), which is then interpreted, or it could be code that is directly executed.

Taking MonPoly for comparison, it uses an implicit generation process (i.e., the monitors are produced internally) with interpreted execution. The input MFOTL formulae are converted into first-order queries (the monitors) and then used to parameterise the generic monitoring algorithm.

In our case, the two options boil down, essentially, to the following:

1. *direct execution* – each property is synthesized into a full-blown monitor implementation (i.e., code), where every variable is hard-coded in the output;
2. *interpreted execution* – we provide the user with a standalone, generic monitor implementation for each pattern (i.e., a monitor for Absence properties, a monitor for Existence properties, etc.), which is then parameterised with information that is specific to each monitorable property (the result of the generation process, e.g., topic names, predicates over messages or timeouts).

Either option works. In order to tip the scales, we view this problem from the perspective of the HAROS framework. HAROS is used for the whole process that leads to monitor generation, from the initial specification to producing an annotated model. It makes sense that monitor generation should be implemented as a plug-in for the framework. Using our new entry point for the analysis of Configurations (see Chapter 5), we can take each annotated Configuration individually, and then generate the appropriate monitors. Configurations are needed, in addition to the properties, in order to provide type information, such as the expected message types of each topic. HAROS can also perform sanity checks on the specified properties, e.g., to ensure that they refer to message fields that actually exist. In this context, we believe that *direct execution* is more appropriate. Instead of having the plug-in generate a set of parameters for the user to plug into an external monitor implementation, we can produce the whole monitor in one go. It is the most portable solution, and allows for greater freedom when it comes to optimisation.

As for the technical aspects of monitor generation, we use metaprogramming and a template engine. Template engines, such as Jinja[6], are similar to compilers or transpilers, in the sense that they take source code as input to produce an output (typically source code). In this case, the input source is a *template*, i.e., an arbitrary source file, similar in content and structure to the intended output (e.g., Python, C++, etc.), in which users can insert variable points, using a mini-language provided by the engine. Given a template and concrete values for each variable, the engine performs the replacement step to produce the output file. In a way, they are more powerful versions of the same mechanisms that support *format strings*, such as:

```
x = 1; y = 2
text = '{c} = {a} + {b}'.format(c=(x+y), a=x, b=y)
print(text)
```

The main difference is that Jinja (and other template engines) support complex features embedded within this template language, such as conditionals, loops, nested templates and template inheritance, to name a

6 https://palletsprojects.com/p/jinja/

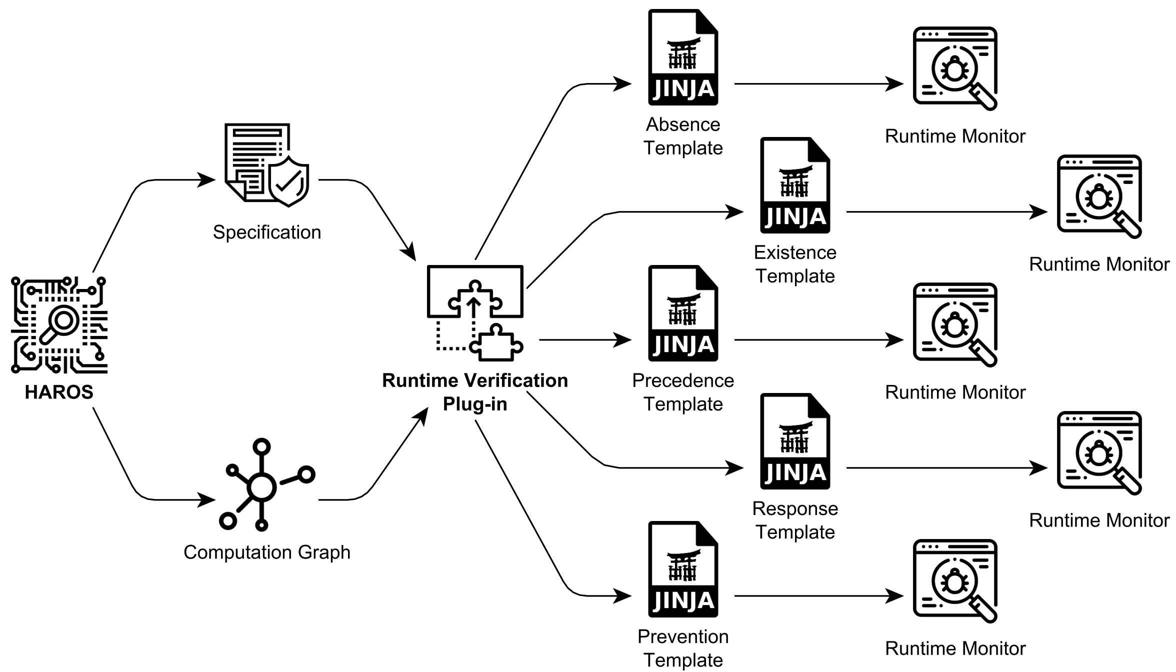


Figure 46: Workflow of the proposed Runtime Verification HAROS plug-in.

few. Figure 46 shows a high-level view of the integrated workflow within HAROS, from specifications to monitor generation using Jinja templates.

6.3.2 *Deployment*

An online monitor is a component that runs alongside a monitored system (or as part of it) for the purposes of the runtime verification process. But there are various methods to integrate the monitor with the system, various ways to observe execution traces, and various degrees of interference with the monitored system. Online monitoring will always have some degree of interference, due to the additional runtime overhead of running the monitors. However, depending on the other choices we make, this interference can be alleviated to an extent.

The most immediate question is where do the generated monitors go, or, rather, where are they placed in the ROS computation graph.

Design Problem 3. *Should the generated monitors run alongside the ROS system (outline monitoring) or as an extension to the existing nodes (i.e., instrumentation, or inline monitoring)?*

In essence, we are deciding whether the monitor generation plug-in produces a standalone (outline) piece of code that executes alongside the ROS system, or whether it inlines the generated monitor in the existing nodes, producing an altered (instrumented) version of the system. There is a grey area in which the monitor runs as a separate entity, but some instrumentation is still applied to the system to report events to the monitor.

Inlining is a common practice, especially in single-process software, as it is the closest a monitor can get to the monitored system. It makes observation of events easier and allows for the earliest possible

detection. It is also a great choice for monitors with an active reaction. The major drawbacks are the increased level of interference, and the requirement of an editable version of the system to instrument – it must alter, e.g., the system's source code or other intermediate representations, such as bytecode. This is not always feasible for various reasons. The target system might be encrypted, might be executing remotely, or it might just be a matter of user permissions.

For ROS, in particular, we deem inlining to be a poor choice for a few reasons.

1. ROS applications are distributed by definition. It is possible for a single property to refer to topics that are advertised by multiple nodes. For inlining to work, in general, we would be forced into a distributed monitor architecture, which raises a number of other questions and obstacles.
2. It is unlikely for users to have access to the source code of all the components that make up the system. After all, one of the core philosophies of ROS is component reuse and, in many cases, reused components are installed as pre-compiled binaries, not as source code.
3. In the context of HAROS, we might be working with Configuration models for which there is neither source code nor pre-compiled binaries to instrument. For instance, we could be using models completely built by hand, or we could set up HAROS to run on a machine other than the machines used for development, such as a remote server.

Outline solutions, i.e., monitors that run as their own entities, are more flexible in the ways they can handle the obstacles above. Distribution of the ROS application, for example, can be tackled both with a distributed monitor architecture and with a centralised monitor architecture, provided there is some interface through which monitors are able to observe all necessary messages. The existence of source code (or lack thereof) might not a problem either, depending on how we approach the observation of events, which leads us to the next question.

Design Problem 4. *Should monitors observe messages directly (i.e., should they subscribe topics on their own, and thus register themselves as ROS nodes), or should the monitored system be instrumented to log events, such as publishing or receiving a message, for external monitors to use?*

This question determines whether we opt for a pure outline solution, or whether we fall into the grey area, using instrumentation to report events, but not to evaluate formulae. The mixed approach is ideal to achieve soundness of the decision procedure, as we will discuss later in this section. However, for the reasons listed previously, regarding inlining, it might not always be feasible. In constrast, outline monitors with no instrumentation, i.e., ROS nodes that passively observe the system, can always be generated and deployed. This, in conjunction with a few other advantages, led us to choose the *outline* approach.

First, outline monitoring nodes can always be generated and deployed because this approach works out-of-the-box with any ROS system. The system is, for all purposes, a black-box with exposed topics, regardless of whether it actually exists or of its current state of development. Monitoring nodes can be generated based on the specification. This decoupled stance is especially appealing in a few circumstances. For example, in the context of HAROS, we want to minimise any restrictions on the systems we are able to analyse, when possible.

Second, the implementation is simpler, since it does not rely on instrumentation. Neither does it rely on dynamic changes to the ROS environment, such as replacing the ROS Master, providing wrappers for the ROS client libraries, or injecting men-in-the-middle nodes (e.g., as seen in ROSRV [85]). The monitored ROS system is left as it is. We have only to put together a template for an independent node, that, like many other ROS tools, uses the standard ROS interfaces to observe messages and perform its job.

Third, and related to the previous point, it is the least intrusive option. Since the implementation of the monitored system is not tampered with, this approach has no effect on the non-functional performance of the monitored nodes. This is an important point; if a node experiences decreased non-functional performance, it can suddenly miss deadlines that it would not otherwise. Missed deadlines in one node mean false assumptions and late (or stale) data for another node, which could ultimately spiral into a system-wide failure. For instance, it is common for sensor nodes (such as laser scanners) to state their publication rate up front in the device documentation. If the overhead of a monitor causes the sensor to skip a message, client nodes (e.g., mapping and localisation) that assume this frequency might misbehave, and cause the robot's model of the world to start diverging from reality. Over time, these small divergences accumulate and cause unexpected behaviour. One could argue that there is overhead in simply running the monitoring system alongside the ROS application, but, since nodes run in independent processes, the risk is smaller. In the worst case, the monitoring nodes could be moved to a separate machine in the ROS network, exchanging performance overhead for late detection – which leads us to another discussion.

6.3.3 *Architecture and Orchestration*

One of the downsides of implementing outline monitors as ROS nodes that subscribe topics on their own is late detection. In this case, monitors have to be *completely asynchronous* (as per the taxonomy in [42]), and completely asynchronous monitors are known to suffer from late detection, in exchange for being the most efficient. Without instrumentation, there is no way to block the system and have it wait for the monitors, so any synchronous or partially synchronous approach is out of the question. These monitors are not ideal to take an active stance (e.g., for system repair), but, as we previously stated, that is not our main goal with this Runtime Verification approach. And, while detection might be late, it is still not as late as with offline monitoring; there is some utility in it to explore.

The degree of latency to which monitors are subject is a variable that we can manipulate to some extent. Our choices regarding monitor architecture, orchestration and distribution may be a contributing (or dampening) factor. But, as noted by Francalanza et al. [68], distributed systems, such as ROS systems, face an increased number of challenges, in particular regarding fault tolerance, correctness and global atomic observations. Thus, lies our next question.

Design Problem 5. *Should monitors be monolithic (i.e., a traditional monitoring approach, with a single central monitor), or should they be distributed over the network (possibly addressing different parts of the specification)?*

Choreographed, *orchestrated* and *decentralised* monitors take advantage of local observations, when a system is spread over multiple network locations. In theory, they can be more efficient than a traditional approach. In a ROS setting, however, nodes tend to live in a reduced number of locations (possibly a single machine), with the notable exception being robot swarms. Even so, spreading monitors over multiple machines in a network would only make a difference for high-frequency publishing between nodes living in the same machine (to decrease the latency of the monitor). As such, we deem the increased complexity of distributed solutions unnecessary and opt for a *traditional monitoring* approach, with a *centralised* architecture.

To further reinforce this position, one of the goals of ROS is to abstract away the network layers and focus on messages, after all. ROS nodes are allowed to subscribe any number of topics, and are able to use a global clock by default (in the form of the `rostime` interface), further discouraging monitor distribution and making the assumption of such a clock (and a total ordering of events) trivial. Furthermore, our specifications are not easily prone to decomposition. There are not many circumstances in which we might be actively looking for concurrent events (for a single property), and the addition of references to past messages adds another layer of complexity to the monitor implementation.

Implementing a generation process that produces a monolithic ROS node (per property) is also convenient to make verdicts available, both to the user and to other components. Once the monitor reaches a verdict, of either $\top$ or $\bot$, regarding the system's compliance with the monitored property, it can simply publish the verdict as a boolean ROS message. Thus, each monitor should advertise a `verdict` topic under its private namespace, so that the verdicts for each property are both easily accessible, as well as easily distinguishable. For instance, if we have two monitor nodes, `/monitor1` and `/monitor2`, we would also have two verdict topics, under the ROS names `/monitor1/verdict` and `/monitor2/verdict`. Components that build on top of this Runtime Verification approach can, then, selectively listen for the verdicts that are relevant to them. With this is mind, there is only one more item to discuss: monitor semantics, or how monitors reach verdicts.

6.3.4 *Semantics*

The final design problem, now closer to the algorithm itself, is related to the monitor's decision procedure.

Design Problem 6. *What kind of decision procedure should monitors adopt?*

Analytical decision procedures, such as querying records, are better suited for an offline monitoring approach, rather than online monitors. Between the different operational procedures, we opt for an *automata-based monitor*. Given that our specification language covers different states of the system and their transitions – e.g., entering a scope or triggering a Response pattern – automata are the most direct implementation.

The semantics of this decision procedure are mostly the same as the semantics we presented in Chapter 4, for the specification language, with one caveat. We defined the semantics of the specification language based on infinite traces. Online monitors can be based on the concept of infinite traces, even

though they always operate on a *finite trace prefix* – the finite number of events that have been observed, up to a certain point in time. The first thing we have to do is to define what is a finite trace prefix. Traces can be defined in terms of temporal first-order structures, as follows.

Definition 5. *Let (D, τ) be a temporal first-order structure over signature S, with $D = (D_0, D_1, \dots)$ a sequence of structures over S and $\tau = (\tau_0, \tau_1, \dots)$ a sequence of natural numbers (time stamps). Given $k \in \mathbb{N}_0$, we define the k-prefix of (D, τ) as the pair $(D_{[0,k)}, \tau_{[0,k)})$, where $D_{[0,k)} = (D_0, D_1, \dots, D_{k-1})$ and $\tau_{[0,k)} = (\tau_0, \tau_1, \dots, \tau_{k-1})$ are finite sequences of size k.*

According to this definition, at the initial instant the monitor is operating on a 0-prefix, i.e., it has not observed any events yet. As soon as it observes the first event, it operates on a 1-prefix trace. Note that this is not a prefix of any particular trace. We know that, theoretically, there is only a single trace of execution. In practice, at any given instant, we have only access to a k-prefix, for some k, which is a valid k-prefix for an infinite number of infinite traces – we cannot predict the future. However, we must be able to query the monitor's verdict at any given time. Monitors do not run for an infinite amount of time, neither does the monitored system. Intuition also says that, in many cases, we should be able to tell that a violation occurred as soon as it is observed.

In order to be able to assign a verdict to each state of the generated automata, we use a three-valued variant of MFOTL, similar to other three-valued variants of temporal logics commonly used in Runtime Verification (see, for instance, [23]). Formally, we define the semantics of the runtime monitors in terms of the semantics given in Chapter 4, as follows.

Definition 6. *Let $k \in \mathbb{N}_0$ and $(D_{[0,k)}, \tau_{[0,k)})$ be a k-prefix of a temporal structure over S. Let v be a valuation, $i \in \mathbb{N}_0$ and Φ be a property over S. We define $(D_{[0,k)}, \tau_{[0,k)}, v, i) \models \Phi$ as*

- $\top$ *iff, for all (D', τ') temporal structures over S, $(D_{[0,k)}, \tau_{[0,k)})$ is a k-prefix of (D', τ') and $(D', \tau', v, i) \models \Phi$;*
- $\bot$ *iff, for all (D', τ') temporal structures over S, $(D_{[0,k)}, \tau_{[0,k)})$ is a k-prefix of (D', τ') and $(D', \tau', v, i) \not\models \Phi$;*
- $?$ *(or unknown) otherwise.*

Informally speaking, a runtime monitor has the following three possible outcomes.

TRUE if and only if, for all possible continuations of the observed trace prefix, the property is always true; i.e., a *good prefix* has been already observed.

FALSE if and only if, for all possible continuations of the observed trace prefix, the property is always false; i.e., a *bad prefix* has been already observed.

UNKNOWN in any other case; i.e., the monitor is still uncertain regarding the validity of future trace suffixes.

The proposed semantics ensure two key properties of the monitor's decision procedure. First, the decision procedure is *complete* – it always produces an output, one of $\top$, $\bot$ or $?$. Second, the decision

procedure is *impartial* – its outputs are not contradictory over time. Monitors only transition from ? to $\top$ or from ? to $\bot$, never backwards, and never from $\top$ to $\bot$ or vice-versa. Unfortunately, another desirable property, *soundness* cannot be guaranteed.

A sound monitor never provides incorrect output. Our monitors, as previously discussed, follow an outline implementation. They are ROS nodes that passively observe the system and the exchanged messages over time. This means that the monitors are subject to the non-deterministic nature of ROS, and it is this non-determinism that puts soundness beyond our reach.

To clarify, in Chapter 4, we define the semantics of our specification language in terms of *"observing a message on a topic"*. For reference:

> Let $\mathbb{T}$ be the set of all ROS topics. For all `/t` $\in \mathbb{T}$ we define a predicate $t \in \mathrm{P}$ such that $t(m)$ is true if and only if a message $m \in \mathbb{M}$ can be observed on topic `/t`.

In practice, we know that topics are abstract entities that represent a network connection between two nodes. There is no such thing as a topic entity, where a message is stored, even temporarily. We must build our implementation around one of two related events: either the *publication* of a message or the *reception* of a message. Ideally, a smart combination of the two would be used. For example, take the property '`globally: /a causes /b within 100 ms`'. If, by analysing the architectural model, we know that the publishers of `/b` also subscribe to `/a`, the monitor should track timestamps for the *reception* of `/a` and the *publication* of `/b` by such nodes.

However, figuring out all the best combinations of publication and reception timestamps, for all kinds of system architectures, is far from trivial. In addition, this is an approach that would require instrumentation. Instrumentation enables monitors to observe events *as they happened* during the execution of a ROS node. There are no out-of-order observations, which is why we previously stated that a *grey approach* (outline with some instrumentation) was ideal.

If instrumentation is not available, we can only refer to the *monitor's reception* of a message. And this is where the non-deterministic nature of ROS enters the scene. Under normal circumstances, ROS only guarantees the ordering of messages within the same topic (but even this can be circumvented, e.g., when using UDP as the transport layer). Messages in different topics can be received in a different order from that in which they were published, for various reasons (e.g., network delays, multithreading, etc.). This means that the system could publish, deterministically, within a single thread, a message on `/a` and then a message on `/b`. On the other end, the monitor could observe the `/b` first, and the message on `/a` afterwards. The same applies to any node in the network, not just the monitors, and, as such, there is nothing we can do regarding this issue. Fortunately, this issue requires such an alignment of factors (timing of the reception of messages, timing of the thread scheduler, etc.) that it should be a relatively rare occurrence in practice. In any case, we advocate that both the system as well as the specified behavioural properties should be designed in such a way that they are resilient to small timing interactions.

Figure 47: Workflow of the proposed Runtime Verification approach.

6.4 Monitor Implementation

In the previous section, we have dicussed how the most feasible approach for online monitoring of ROS applications is to synthesize each specified property into a monolithic, standalone ROS monitoring node. The synthesized node is a passive process – it does not interfere with the system's execution. It should subscribe to each topic the original property refers, and observe the exchanged messages on such topics, until it is able to reach a verdict of either $\top$ or $\bot$. In order to make its verdict easily observable (and usable by other components), the monitoring node should advertise a topic, `~verdict`, where the verdict of the monitored property is to be published. If the monitor reaches a verdict of $\top$ (resp. $\bot$), the monitor should publish a boolean message on `~verdict` containing **`True`** (resp. **`False`**). At that point, the monitor can safely stop operating – verdicts are final. In every other state, the monitor's verdict is ? and nothing should be published. The diagram in Figure 47 illustrates the described workflow.

Regarding the decision procedure, our monitor implementation is based on automata. More specifically, we implement monitors as state machines whose states and transitions reflect observed messages that are relevant for the property. Naturally, different scopes and property patterns lead to different state machines. In spite of these differences, however, from a high-level point of view, the handling of property scopes changes little from pattern to pattern. Thus, we divide the presentation of our approach by pattern, and discuss scopes where appropriate. Furthermore, binary property patterns (Precedence, Response, Prevention) are quite more complex than unary patterns (Absence, Existence), due to the possible cross-references between message contents, e.g., '`/a as A causes /b {x = @A.x}`'. If there are no such references, the resulting state machines are much simpler. So, besides the division by pattern, we also subdivide binary patterns into the simple and the general cases.

We provide state machine diagrams for each discussed pattern, according to the notation in Figure 48, to help illustrate our approach. All diagrams reflect the implementation for the **`after-until`** scope, which

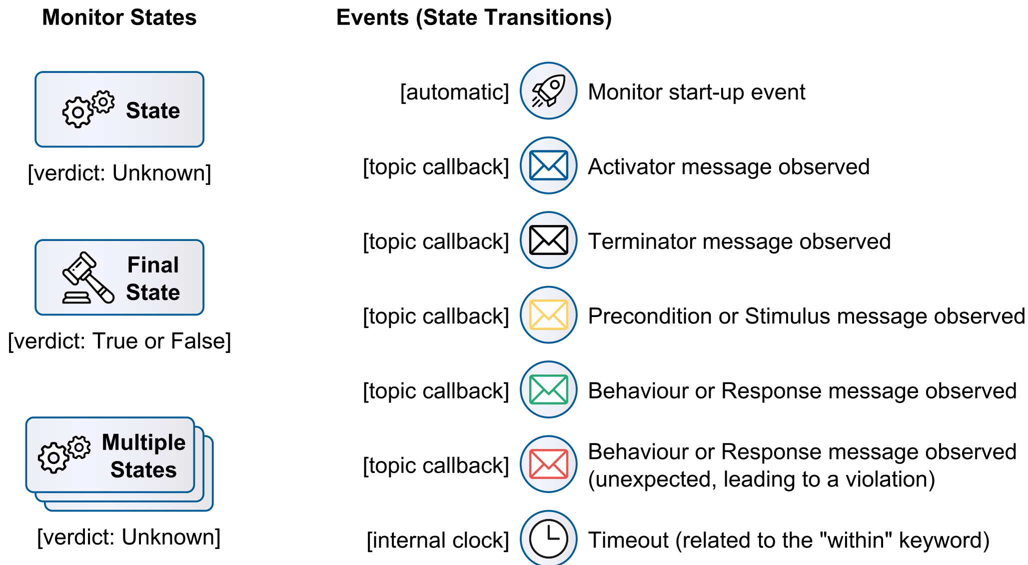


Figure 48: Notation for state machine diagrams.

is the most complex (and least efficient) case. The monitor start-up (or launch) event is an event that is automatically triggered on the monitor, once the ROS node is fully initialised. The state transitions via activator message do not exist for the `globally` and `until` scopes, since these scopes do not have a corresponding activator event (they start at the initial instant). In these cases, this transition is replaced with an implicit event that triggers once, right after the monitor's start-up event. The state transitions via terminator message do not exist for the `globally` and `after` scopes, since these scopes do not have a corresponding terminator event (they never terminate). These differences lead to optimisations that we will discuss further.

6.4.1 *Absence Properties*

The monitors for Absence properties implement the state machine shown in Figure 49.

Absence properties are stated in the negative form, e.g., '`no /bumper {data < 0 or data > 7}`'. Intuitively, to detect a violation, monitors have to look for messages that match exactly the undesired *behaviour* pattern, i.e., what comes after `no` (any message on `/bumper` such that `data < 0` or `data > 7` in this case). As shown in the diagram, this state machine is composed of three non-final states.

Inactive State This is the initial state. In this state, the monitor ignores all messages related to the inner pattern of the property. It is actively looking only for the necessary events to enter the scope of the property. Upon observing the scope's activator event, it transitions to the Active state.

Active State In this state, the monitor assumes that the trace is within a valid scope. It is actively looking for messages that match the behaviour event or messages that terminate the scope. Upon observing any messages that match the behaviour, a verdict of $\perp$ is produced – the property states that there should

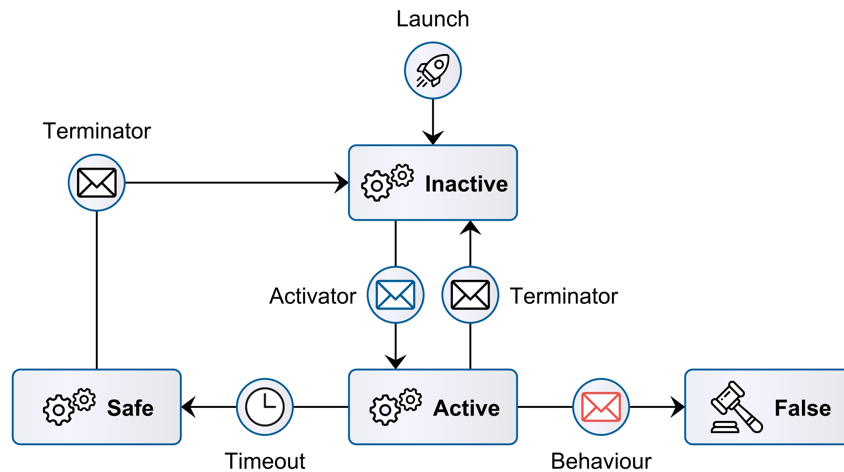


Figure 49: State machine monitoring for the Absence pattern.

be none, after all. Upon observing the scope's terminator event, it simply transitions back to the Inactive state. In the case that the original property includes a timeout (e.g., '`no /bumper within 100 ms`'), once the specified duration elapses (100 milliseconds in), the monitor transitions to the Safe state.

Safe State In this state, the monitor assumes that the trace is within a valid scope. The monitor only reaches this state once the property's timeout elapses. As per the language's semantics, once the timeout elapses, it is no longer possible to violate the property; even if we observe messages matching the behaviour event. Thus, this state is deemed *safe*, i.e., it cannot transition to a verdict of $\bot$ by any means. Upon observing the scope's terminator event, it simply transitions back to the Inactive state. A timeout cannot make the monitor transition directly to the Inactive state because, upon observing an activator message (within the same scope), it would transition back to the Active state, which is incorrect. The monitor must observe, at least, one terminator message before it tracks further scope activators.

Optimisations

An immediate fact about the presented diagram is that there are no final states with a verdict of $\top$, implying that Absence monitors can never deem a property as being true based on the observation of a trace prefix. Considering that the semantics operate on infinite traces, and that the **`after-until`** scope can occur infinitely often (due to being reentrant), it is, indeed, impossible to reach a verdict of $\top$. But this does not apply to all scopes.

The **`globally`** and the **`after`** scopes do not have a terminator. If we remove the transitions via terminator event from the diagram, we are left with no transitions out of the Safe state. Thus, the Safe state degenerates into a final state. In this case, it degenerates into a verdict of $\top$ because, as previously stated, it is impossible to violate the property when in the Safe state.

Furthermore, the **`globally`** and the **`until`** scopes do not have an activator. In this case, the monitor's start-up event can take the monitor directly into the Active state, making it the new initial state. When a terminator is observed (for the **`until`** scope), the monitor transitions from the Active or the Safe states

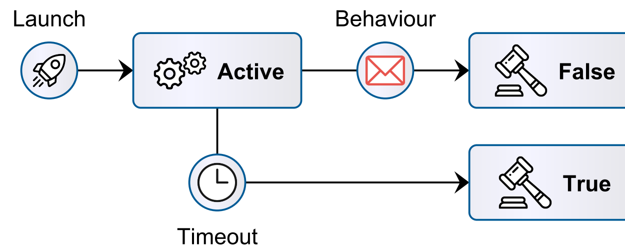


Figure 50: State machine for the Absence pattern with the **globally** scope.

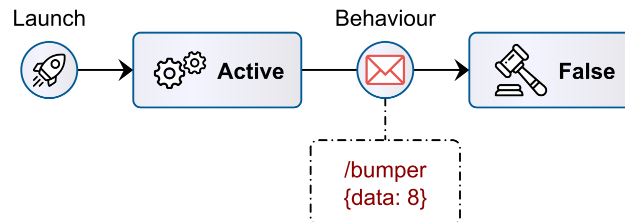


Figure 51: Monitoring example for the Absence pattern.

to the Inactive state. The Inactive state becomes a final state, since there are no activators, and, again, degenerates into a verdict of $\top$ – once out of the scope, it is impossible to violate the property.

With the two aforementioned optimisations in mind, we can now observe the reduced state machine for the **globally** scope, for example, in Figure 50.

Example

As a practical example of an Absence monitor, consider the following property of the Fictibot system,

```
1 globally: no /bumper {data < 0 or data > 7}
```

And the following trace of messages, which constitutes a violation to the property.

```
1 @50ms  /bumper {data: 0}
2 @100ms /bumper {data: 8}
```

Note that we only list messages on the relevant topics (i.e., `/bumper`). Other messages have no effect on the monitor. The resulting sequence of monitor states is shown in Figure 51.

Since the scope is implicit (**globally**), the monitor transitions directly to the Active state, considering the reduced state machine. And it shall remain in this state until a verdict is produced, for there is no scope terminator. The first message matches the behaviour topic, but, for the value 0, the event predicate (‘`data < 0 or data > 7`’) is false; the message is not a match. In this case, non-matching messages represent valid system behaviour. The second message, with a value of 8, matches the predicate (‘`data > 7`’), which, as expected, leads the monitor through a transition to a verdict of $\bot$.

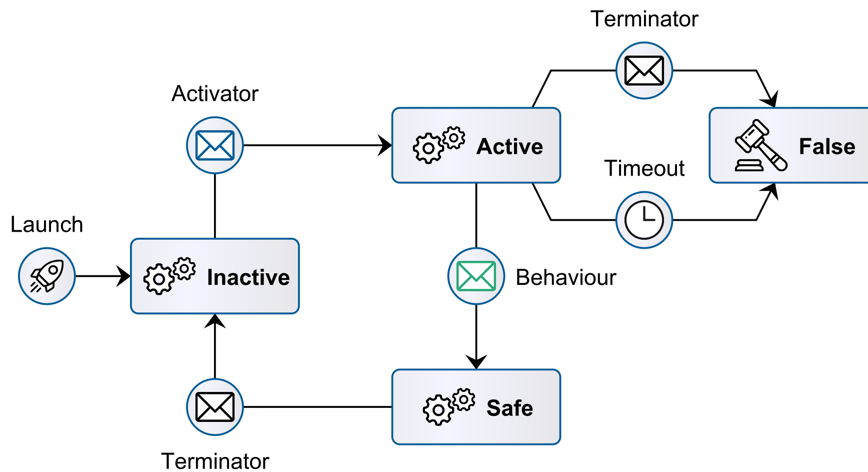


Figure 52: State machine monitoring for the Existence pattern.

6.4.2 *Existence Properties*

The monitors for Existence properties implement the state machine shown in Figure 52.

Existence properties (e.g., '`some /bumper within 1000 ms`') require the observation of at least one message matching the *behaviour* pattern, within the scope and the specified timeout. Intuitively, to detect a violation, there must not be any matching message until the scope terminates or the timeout elapses. As shown in the diagram, this state machine is composed of three non-final states.

Inactive State This is the initial state. In this state, the monitor ignores all messages related to the inner pattern of the property. It is actively looking only for the necessary events to enter the scope of the property. Upon observing the scope's activator event, it transitions to the Active state.

Active State In this state, the monitor assumes that the trace is within a valid scope. It is actively looking for messages that match the behaviour event or messages that terminate the scope. Upon observing a message that matches the behaviour, it transitions to the Safe state – the property can no longer be violated within that scope. Upon observing the scope's terminator event, it produces a verdict of $\perp$, because the monitor expected at least one behaviour message within the scope. The same applies if the specified pattern duration elapses.

Safe State In this state, the monitor assumes that the trace is within a valid scope. The monitor only reaches this state after a behaviour message has been observed. As per the language's semantics, after observing a behaviour message, it is no longer possible to violate the property *within the same scope*. Thus, this state is deemed *safe*, i.e., it cannot transition to a verdict of $\perp$ by any means. Observing further behaviour messages has no effect. Upon observing the scope's terminator event, it simply transitions back to the Inactive state.

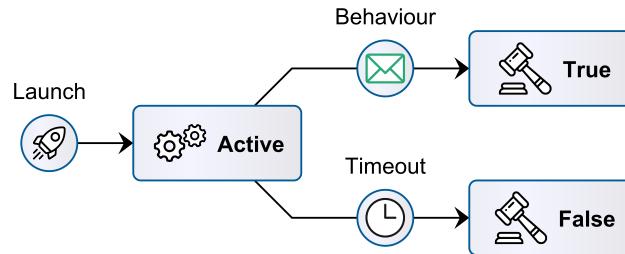


Figure 53: State machine for the Existence pattern with the **globally** scope.

Optimisations

The state machine for Existence properties, like the state machine for Absence properties, features no final states with a verdict of $\top$. The reasoning is the same: the semantics operate on infinite traces, and the **after-until** scope can occur infinitely often. At any given moment, the monitor is either unsure, or it is certain that a violation has occurred. As was the case with the previous pattern, there is room for optimisation, for other scopes.

The **globally** and the **after** scopes do not have a terminator. If we remove the transitions via terminator event from the diagram, we are left with no transitions out of the Safe state. Thus, the Safe state degenerates into a final state, with a verdict of $\top$. There is also one less transition from Active to a verdict of $\bot$, meaning that the only way to falsify the property is via timeout.

The **globally** and the **until** scopes do not have an activator. As we have seen before, the monitor's start-up event can take the monitor directly into the Active state, making it the new initial state. In a similar fashion to the Absence pattern, the Inactive state becomes a final state, since there are no activators. In this case, only the Safe state can transition into the Inactive state, meaning that, again, it degenerates into a verdict of $\top$. We can take this further – the Safe state has only a single transition, into a verdict of $\top$. Thus, we can safely replace the Safe state with the final state, and remove the Inactive state entirely.

With the two aforementioned optimisations in mind, we can now observe the reduced state machine for the **globally** scope, for example, in Figure 53.

Example

As a practical example of an Existence monitor, consider the following property of the Fictibot system:

```
until /stop_cmd: some /bumper within 1000 ms
```

This property states two things. First, that there are no `/stop_cmd` messages before `/bumper` messages. Second, that the publisher of `/bumper` messages takes, at most, one second to initialise and start publishing.

Consider the following trace of messages *(i)*, which constitutes a valid execution,

```
@100ms /bumper {data: 1}
@110ms /stop_cmd {}
```

And the following trace *(ii)*, which constitutes a violation to the property via timeout.

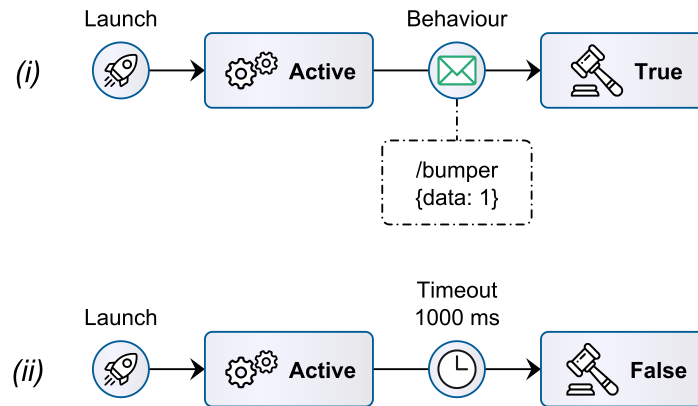


Figure 54: Monitoring examples for the Existence pattern.

```
1 @1050ms /bumper {data: 1}
2 @1100ms /stop_cmd {}
```

The resulting sequence of monitor states is shown in Figure 54.

Since the scope's activator is implicit (**until**), the monitor transitions directly to the Active state. In *(i)*, the first message matches the behaviour topic, within the specified time window, causing a transition to Safe. But, as we have just discussed, for the **until** scope, we can collapse the Inactive and the Safe states into a single final state, with a verdict of $\top$. The terminator that comes afterward, then, is irrelevant, since a decision is already made. In *(ii)*, the first behaviour message only arrives after 1050 milliseconds have elapsed. By that time, the state timer on Active has already ran out, and a verdict of $\bot$ has been produced.

6.4.3 *Precedence Properties (Without Variable References)*

The monitors for Precedence properties (without variable references) implement the state machine shown in Figure 55.

Precedence properties are binary in the sense that a *behaviour* event requires a *precondition* event to have happened before, e.g., '`/stop requires /bumper within 100 ms`'. For this complex pattern, both the behaviour and the precondition are monitored at the same time. Valid precondition messages are logged, and behaviour messages trigger a precondition check. Without variable references, logging boils down to keeping track of timestamps, i.e., *when* was the last precondition message observed. Intuitively, to detect a violation, we must observe a behaviour message without having observed any previous precondition messages (at all, or, at least, within the specified time window). As shown in the diagram, this state machine is also composed of three non-final states. It works similarly to the Absence state machine, with the major difference being its Safe state.

Inactive State This is the initial state, and it works the same way as the Inactive states of other state machines seen so far. The monitor is actively looking for activator events and ignoring all other messages. Upon observing the scope's activator event, it transitions to the Active state.

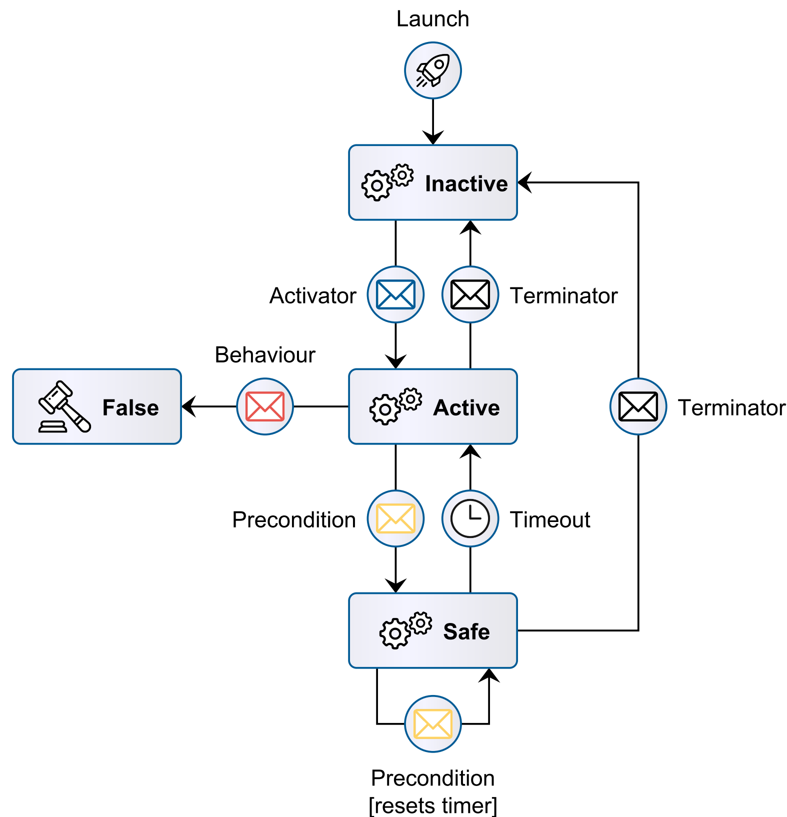


Figure 55: State machine monitoring for the Precedence pattern (without variable references).

Active State In this state, the monitor assumes that the trace is within a valid scope. It is actively looking for messages that match the behaviour event, messages that match the precondition event or messages that terminate the scope. As is the case when monitoring Absence properties, upon observing a message that matches the behaviour, a verdict of $\bot$ is produced – the property states that there should be none without the proper precondition. Upon observing a precondition message, the monitor transitions to the Safe state. Upon observing the scope's terminator event, it simply transitions back to the Inactive state.

Safe State In this state, the monitor assumes that the trace is within a valid scope. The monitor only reaches this state after observing a precondition message. Observing behaviour messages in this state has no effect; a precondition is already in place to validate the behaviour. In the case that the original property includes a timeout, once the specified duration elapses, the monitor transitions back to the Active state, as the observed precondition message is no longer valid. Observing further precondition messages renews the timer on this state (if applicable). Upon observing the scope's terminator event, it simply transitions back to the Inactive state.

Optimisations

The state machine for Precedence properties, like the previous state machines for unary patterns, features no final states with a verdict of $\top$. This is a recurring issue, always for the same reasons: the semantics

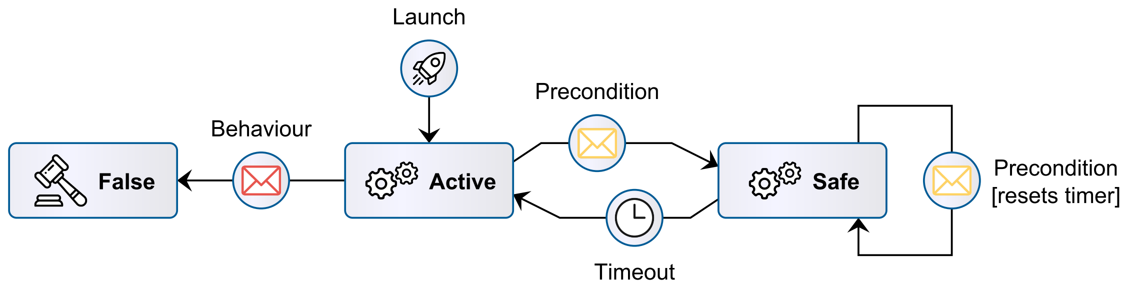


Figure 56: State machine for the Precedence pattern with the **globally** scope.

operate on infinite traces, and the **after-until** scope can occur infinitely often. We can short-circuit some transitions for other scopes, as we have done in the previous cases.

The **globally** and the **after** scopes do not have a terminator. If we remove the transitions via terminator event from the diagram, we are left with a monitor that can only leave the Safe state via timeout. If a timeout is not specified, then there are no further transitions out, and the Safe state degenerates into a final state, with a verdict of $\top$.

The **globally** and the **until** scopes do not have an activator. As we have seen before, the monitor's start-up event can take the monitor directly into the Active state, making it the new initial state. In a similar fashion to the Absence pattern, the Inactive state becomes a final state, since there are no activators. Again, this state degenerates into a verdict of $\top$. If a timeout is not specified, the only transition out of the Safe state is to this verdict. In that case, we can collapse the Safe state directly into the verdict of $\top$.

With the two aforementioned optimisations in mind, we can now observe the reduced state machine for the **globally** scope, for a property with timeouts, in Figure 56.

Example

As a practical example of a Precedence monitor, consider the following property of the Fictibot system,

```
after /state {data > 0} until /state {not data > 0}:
    /stop_cmd requires /high_priority_stop within 100 ms
```

And the following trace of messages, which constitutes a violation to the property via timeout.

```
@100ms /state {data: 1}
@200ms /high_priority_stop {}
@400ms /stop_cmd {}
```

The resulting sequence of monitor states is shown in Figure 57.

By default, the monitor starts in the Inactive state. After observing the activator message, it transitions to the Active state, in which it listens both for precondition and behaviour messages. In this case, a precondition happens to be observed first, causing the transition to the Safe state. However, more than 100 milliseconds pass until the next message. The absence of messages triggers the timer transition on the Safe state, and the monitor reverts to the Active state; the observed precondition is no longer valid. When

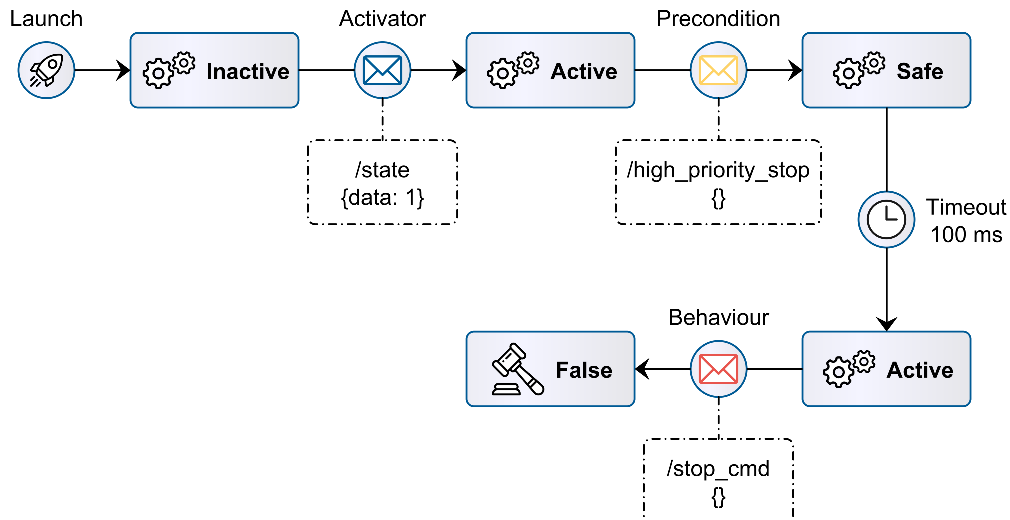


Figure 57: Monitoring example for the Precedence pattern (without variable references).

the behaviour message is finally received, since there are no valid precondition messages, the monitor produces a verdict of ⊥.

6.4.4 *Precedence Properties (General Case)*

The monitors for Precedence properties (in general) implement the state machine shown in Figure 58.

The intuition for Precedence properties with variable references, e.g., '`/b as B requires /a {data = @B.data} within 100 ms`', is the same as in the simple case. To detect a violation, we must observe a behaviour message without having observed any previous *matching* precondition messages (at all or, at least, within the specified time window). But, in this case, we require a more complex logging structure.

Every precondition message has to be recorded individually, along with its timestamp, for the specified duration (e.g., 100 milliseconds). After this duration has elapsed, the log entry can be safely discarded. If the duration is left unspecified, log entries have to be kept until the end of the scope. This is due to the fact that we do not have the entire trace up front, and are unable to predict the future, contrary to offline monitors.

As shown in the diagram, this state machine is composed of multiple states. The number is dynamic, and cannot be determined in advance. In essence, they are all the same, except for the number of entries that is still valid (i.e., within its duration) in the precondition message log.

Inactive State This state works in the same fashion as the Inactive states of the state machines presented so far. Upon observing the scope's activator event, the monitor transitions to the Active-0 state.

Active-0 State In this state, the monitor assumes that the trace is within a valid scope. There are no entries in the precondition message log. The monitor is actively looking for messages that match the behaviour event, messages that match the precondition event or messages that terminate the scope.

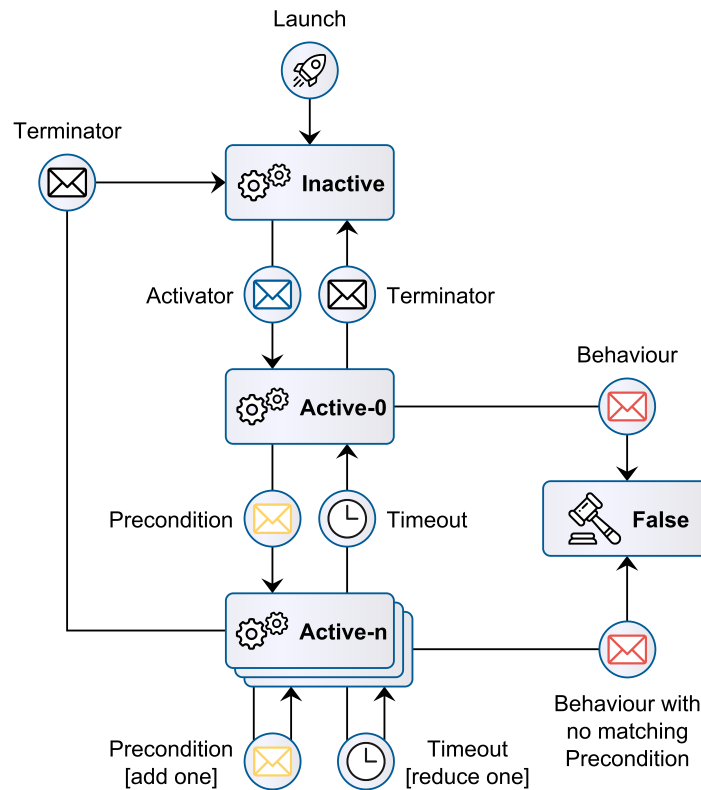


Figure 58: State machine monitoring for the Precedence pattern.

Upon observing a message that matches the behaviour, a verdict of $\bot$ is produced – the property states that there should be none without the proper precondition. Upon observing a precondition message, the monitor adds it to the message log, along with its timestamp, and transitions to the Active-1 state (an instantiation of the Active-n state). Upon observing the scope's terminator event, it simply transitions back to the Inactive state.

Active-n State This is a generic state that is instantiated every time a precondition message is observed and added to the log (Active-1, Active-2, etc.). In this state, the monitor assumes that the trace is within a valid scope. It is actively looking for messages that match the behaviour event, messages that match the precondition event or messages that terminate the scope. Upon observing a message that matches the behaviour, the message log is queried. If there is a matching precondition message for the current behaviour message, the monitor remains in the same state. If there is no matching precondition message, a verdict of $\bot$ is produced. Upon observing a precondition message, the monitor adds it to the message log, along with its timestamp, and transitions to the next Active-n state (Active-2 if it is in Active-1, etc.). Upon observing the scope's terminator event, it simply transitions back to the Inactive state. If a timeout is specified in the property, this state keeps track of the current time. If the oldest message in the log expires, it is removed from the log and the monitor transitions to an $n - 1$ state (Active-1 if it is in Active-2, etc.). If the message log becomes empty, the monitor transitions back to the Active-0 state.

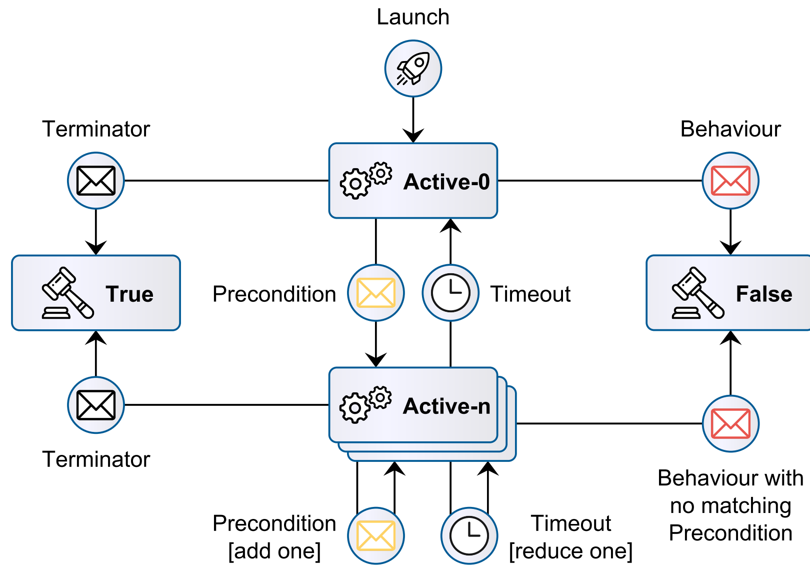


Figure 59: State machine for the Precedence pattern (general case) with the **until** scope.

Optimisations

This is the first state machine seen so far where there is a valid concern regarding space efficiency. Successive precondition messages cause the message log to grow, possibly until the monitor runs out of available memory. One way to alleviate this issue is by specifying timeouts in the properties. Log entries can be safely dropped after their duration expires, as mentioned. Another way to make space management more efficient is by not storing entire messages. With a simple static analysis step over the property, during monitor generation, we can determine which data fields are directly referenced. Those are the only fields we need to store in the log, which could turn out to make a significant difference (e.g., storing a floating-point number, versus storing an entire odometry message).

Contrary to the trend set by the previous patterns, there are not many states and transitions we can remove for the **globally**, **after** and **until** scopes. If we remove the transitions via terminator event from the diagram, we are still left with a monitor that is unable to reach a verdict of $\top$. Removing timeouts, in addition to terminators, makes no difference in this case, because of the Active-n state. This state can always increase its counter (and provide an opportunity for failure) with additional precondition messages. By removing only the activator, however, in the **until** scope, we are able to degenerate the Inactive state into a verdict of $\top$. In practice, this is the same as working with a finite trace that ends with the terminator message. Hence, all possible continuations of the trace after the terminator message are effectively equivalent. This optimisation is depicted in the reduced state machine shown in Figure 59.

Example

As a practical example of a Precedence monitor, with variable references, consider the following property of the Fictibot system,

```
1 after /state {data > 0} until /state {not data > 0}: /controller_cmd as CMD
```

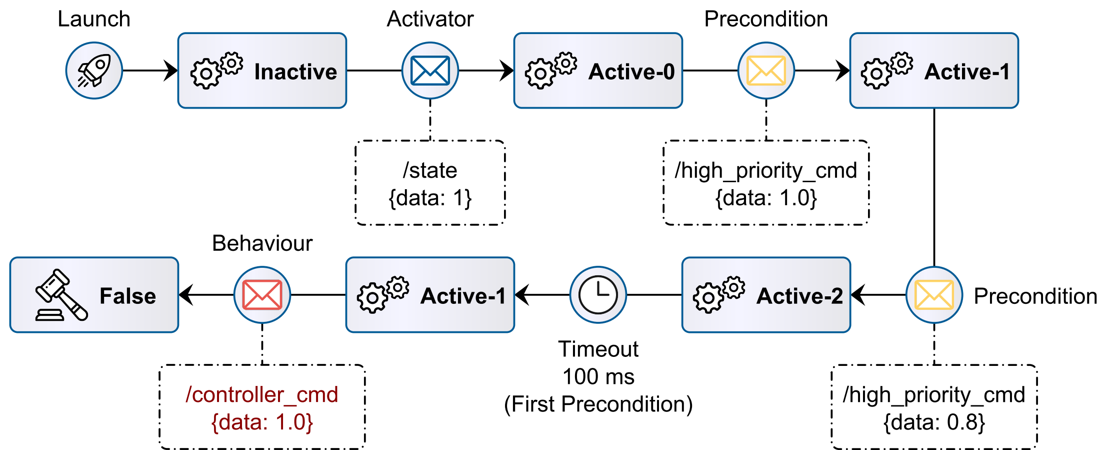


Figure 60: Monitoring example for the Precedence pattern (with variable references).

```
2     requires /high_priority_cmd {data = @CMD.data} within 100 ms
```

And the following trace of messages, which is invalid behaviour.

```
1 @100ms /state {data: 1}
2 @200ms /high_priority_cmd {data: 1.0}
3 @250ms /high_priority_cmd {data: 0.8}
4 @325ms /controller_cmd {data: 1.0}
```

The resulting sequence of monitor states is shown in Figure 60.

By default, the monitor starts in the Inactive state. After observing the activator message, it transitions to the Active-0 state, in which it listens both for precondition and behaviour messages. In this case, a precondition happens to be observed first, causing the transition to the Active-1 state. The monitor now contains '`/high_priority_cmd {data: 1.0}`' in its message log. A second precondition message, '`/high_priority_cmd {data: 0.8}`', is registered, causing a transition to the Active-2 state. However, more than 100 milliseconds pass, since the first precondition, until the next message. The passage of time triggers an update to the message log, causing '`/high_priority_cmd {data: 1.0}`' to be discarded, and the monitor reverts to the Active-1 state. Finally, the monitor observes a behaviour message, which must contain the same `data` as the single entry on the log, i.e., the value 0.8. This is not the case, and a verdict of $\bot$ is produced.

6.4.5 *Response Properties (Without Variable References)*

The monitors for Response properties (without variable references) implement the state machine shown in Figure 61.

By this point, we can see that all monitors, regardless of their pattern, share a number of common states and traits. To avoid repetition, we will skim over the similarities and focus on the differences.

Response properties are binary liveness properties, stating that a *stimulus* (or *precondition*) event leads to an eventual *response* (or *behaviour*) event, e.g., '`/bumper causes /stop within 100 ms`'. This pattern

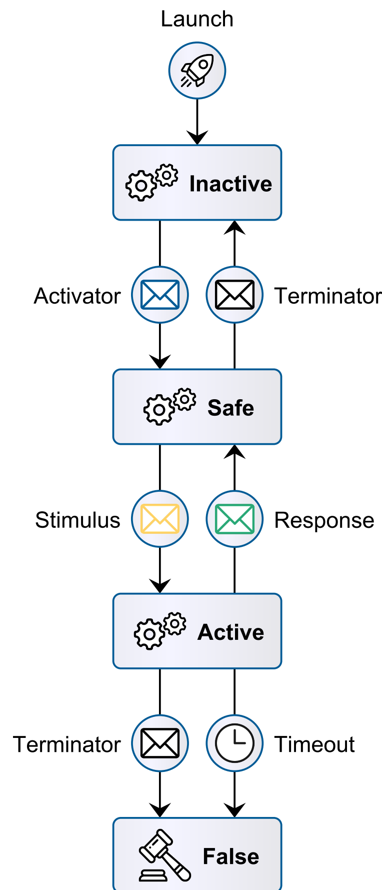


Figure 61: State machine monitoring for the Response pattern (without variable references).

enforces an ordering over messages, which allows monitoring to be sequential. The response event only needs to be monitored after (and if) a stimulus event has been observed. Intuitively, to detect a violation, we must observe a stimulus message for which no response message is observed afterwards (at all, or, at least, within the specified time window). As shown in the diagram, this state machine is composed of three non-final states.

The main novelty in this state machine is that the Inactive state transitions to a Safe state, rather than an Active state. This is due to the aforementioned ordering over messages. We are only at risk of violating a property *after* a stimulus message is observed. In the Active state, observing further stimuli (besides the first one) has no effect because the first stimulus will always be the one with the oldest timestamp, and the one with the shortest time remaining on the associated timer. Upon observing a message that matches the response, the monitor transitions back to the Safe state – one response suffices to match all previous stimuli.

The usual optimisations apply here. Removing the activator turns Inactive into a final state with a verdict of $\top$. Removing only the terminator or only the timeout does not yield much of an improvement. Removing both (`globally` or `after` scope with no specified timeout), however, collapses the whole state machine into a loop between Safe and Active that never reaches any verdict. In other words, monitoring unbounded liveness properties (e.g., '`globally: /a causes /b`'), is pointless, as previously discussed.

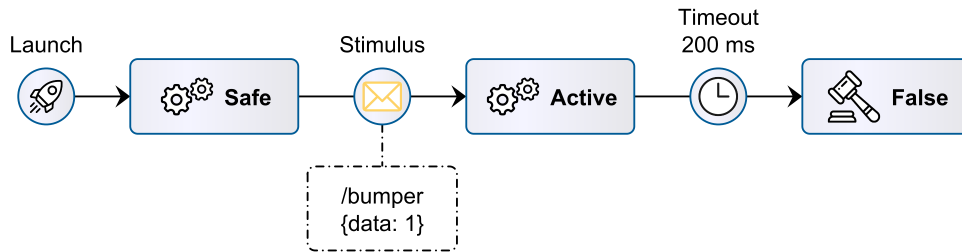


Figure 62: Monitoring example for the Response pattern (without variable references).

As an example of a Response monitor, consider the following property of the Fictibot system,

```
globally: /bumper {not data = 0} causes /normal_priority_stop within 200 ms
```

And the following trace of messages, which constitutes invalid behaviour.

```
@100ms /bumper {data: 1}
@400ms /normal_priority_stop {}
```

The resulting sequence of monitor states is shown in Figure 62.

By default, the monitor starts in the Safe state, since the scope activator is implicit. After observing the stimulus message, it transitions to the Active state, in which it awaits a response message. The response message is not observed within the time window, leading the monitor to a verdict of $\bot$.

6.4.6 *Response Properties (General Case)*

The monitors for Response properties (with variable references) implement the state machine shown in Figure 63.

The intuition for Response properties with variable references, e.g., '`/a as A causes /b {data = @A.data} within 100 ms`', is the same as in the simple case. To detect a violation, we must observe a stimulus message, but not a *matching* response message (at all or, at least, within the specified time window). But, in this case, we must maintain internal state across states, i.e., we must log stimulus messages, just like we had to log precondition messages in the Precedence pattern. Also in similar fashion to the Precedence pattern, this state machine is composed of a dynamic number of states, depending on how many unmatched stimuli have been observed.

Every stimulus message has to be recorded individually, along with its timestamp, for the specified duration. If this duration runs out, we have detected a property violation. If the duration is left unspecified, log entries have to be kept until the end of the scope. Again, there are concerns regarding space efficiency, but we can optimise logging by storing just the data fields that have been directly referenced in the property. In addition, observing a valid response message enables us to remove all matching stimuli entries from the message log.

As an example of a Response monitor, with variable references, consider the following property of the Fictibot system,

```
after /state {data > 0} until /state {not data > 0}: /high_priority_cmd as CMD
```

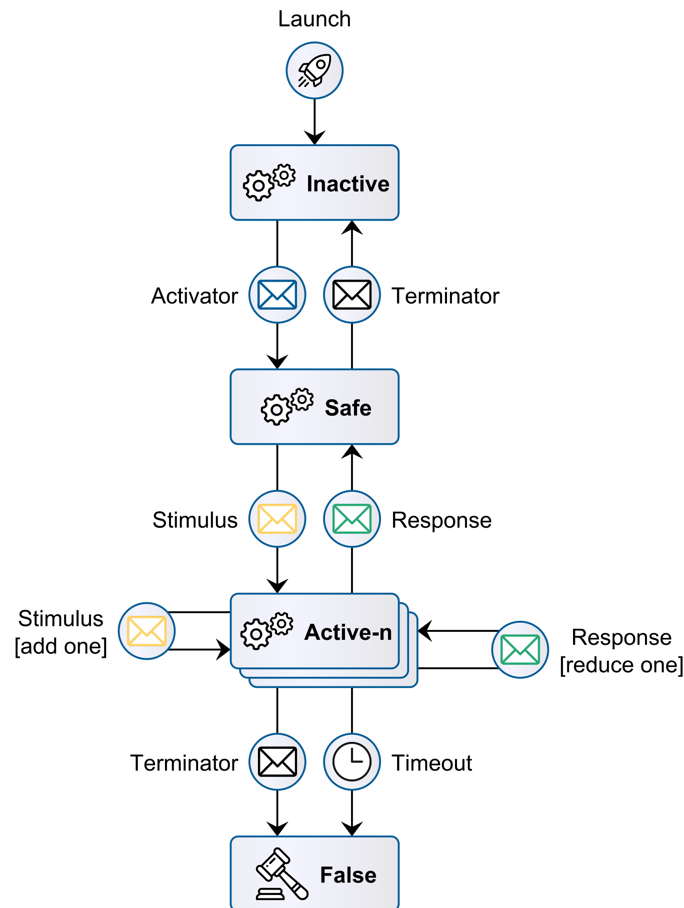


Figure 63: State machine monitoring for the Response pattern (with variable references).

```
2      causes /controller_cmd {data = @CMD.data} within 100 ms
```

And the following trace of messages, which constitutes a violation.

```
1 @100ms /state {data: 1}
2 @200ms /high_priority_cmd {data: 1.0}
3 @210ms /high_priority_cmd {data: 0.8}
4 @250ms /controller_cmd {data: 0.8}
```

The resulting sequence of monitor states is shown in Figure 64. There is nothing surprising in this example. The monitor enters the scope and receives two stimulus messages. However, only the second stimulus, with `data = 0.8`, received a response, causing the monitor to report a violation regarding the first stimulus.

6.4.7 *Prevention Properties*

At last, we present the monitors for Prevention properties. Since there is little novelty at this point, there is little need to separate the cases with and without variable references. The state machine in Figure 65 corresponds to the simple case, whereas the state machine in Figure 66 covers the general case.

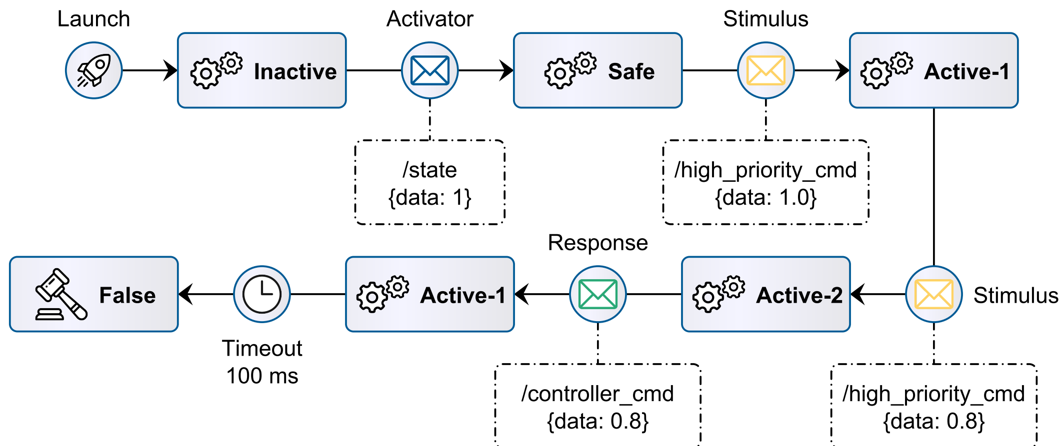


Figure 64: Monitoring example for the Response pattern (with variable references).

The Prevention pattern is a binary safety property that is very similar to the Response pattern, regarding monitoring. The main difference is that a *stimulus* event leads to the absence of a *response* event, e.g., '`/bumper forbids /cmd within 100 ms`'. While in the Active state, observing a response produces a verdict of ⊥; responses must not be observed in this state. In the case that the original property includes a timeout, once the specified duration elapses, the monitor simply transitions back to the Safe state; the observed stimuli up to that point are no longer valid. The timer starts counting from the moment the monitor enters this state, i.e., after the stimulus message that caused the transition. Observing further stimuli restarts the timer.

For Prevention properties with variable references, as in the previous binary patterns, we must log stimulus messages (or their relevant data fields) along with the respective timestamp for the specified duration. If this duration runs out, the entry can be safely dropped from the log.

As an example of a Prevention monitor with variable references consider the following property of the Fictibot system,

```
1 after /state {data > 0} until /state {not data > 0}: /high_priority_cmd as CMD
2     forbids /controller_cmd {not data = @CMD.data} within 100 ms
```

And the following trace of messages, which constitutes a violation.

```
1 @100ms /state {data: 1}
2 @200ms /high_priority_cmd {data: 1.0}
3 @210ms /high_priority_cmd {data: 0.8}
4 @250ms /controller_cmd {data: 0.8}
```

After observing the activator message, a stimulus message eventually arrives, causing a transition to the Active-1 state. The monitor now contains '`/high_priority_cmd {data: 1.0}`' in its message log. A second stimulus message, '`/high_priority_cmd {data: 0.8}`', is registered, causing a transition to the Active-2 state. When the monitor observes a behaviour message, '`/controller_cmd {data: 0.8}`', it matches as a response to the second stimulus message. Thus, a verdict of ⊥ is produced; such a behaviour message would have to wait another 60 milliseconds before it could be published.

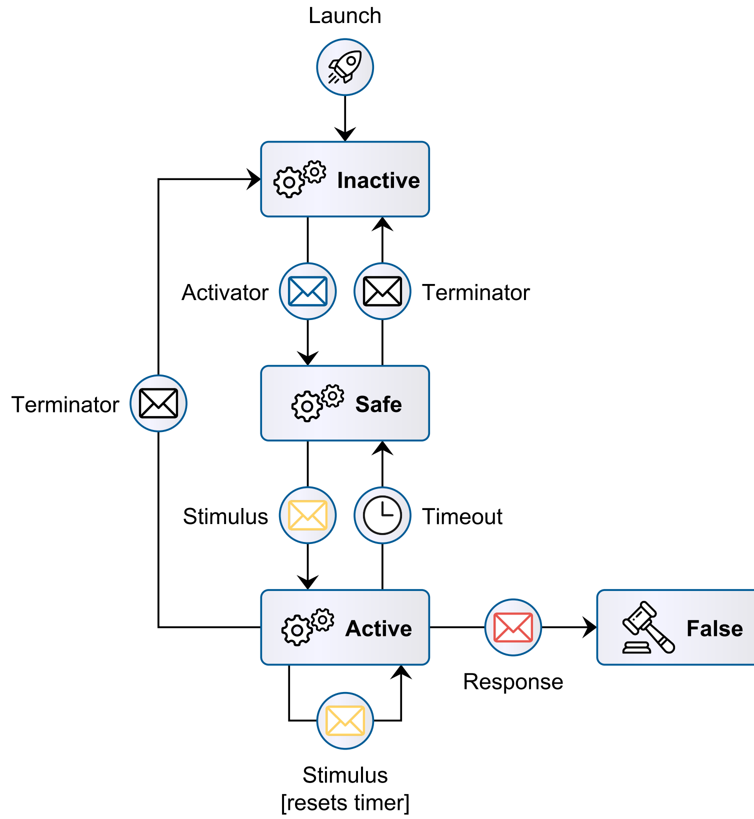


Figure 65: State machine monitoring for the Prevention pattern (without variable references).

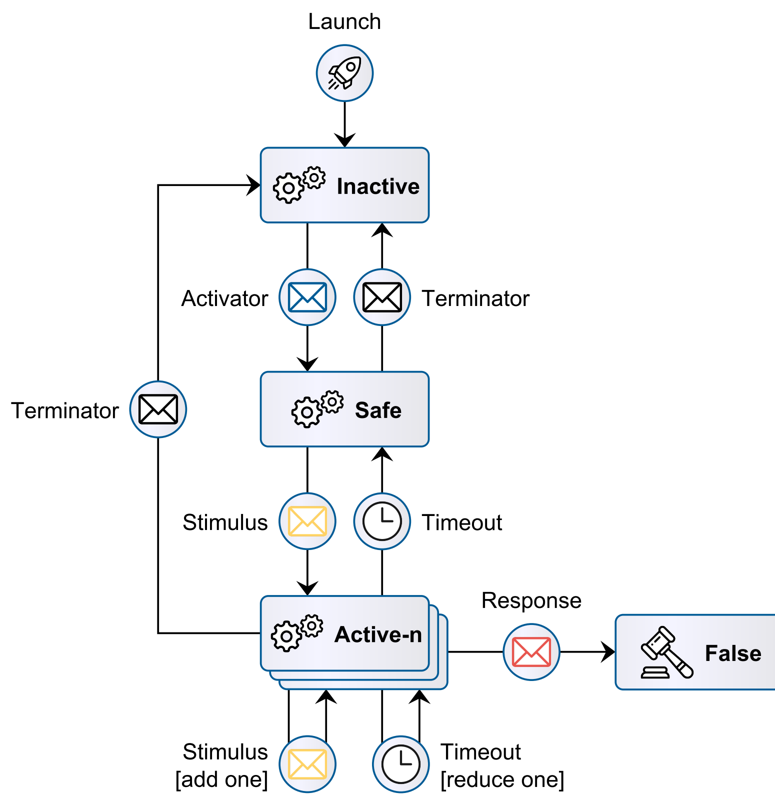


Figure 66: State machine monitoring for the Prevention pattern (with variable references).

6.5 Summary

In this chapter we have presented our first approach to check whether a ROS system complies with its behavioural specification. Contrary to previous chapters, we have deviated from static analyses. We proposed an approach based on Runtime Verification, in which runtime monitors are automatically generated from a specification and an architectural model. Monitors are then deployed as ROS nodes that passively observe exchanged messages and produce a message of their own, containing a verdict regarding the property.

Even though this approach is not one of the core contributions of this thesis, it is an utility on its own, and a building block for other approaches, such as the one we present in Chapter 7. Besides, this is the first Runtime Verification approach for ROS that is based on a high-level, pattern-oriented specification language, that is able to capture both real-time constraints as well as cross-references between messages.

7

PROPERTY-BASED TESTING FOR ROS APPLICATIONS

Showing that a system behaves as expected, with concrete evidence, is not an easy problem to solve. One method that stands out in the software industry for its efficiency and typically shallow learning curve, to the point of being almost mandatory, is software testing [136]. Testing is capable of improving the rate of early failure detection, and the overall quality assurance of a system, with relatively little effort. It does not suffer from being an undecidable problem, as some static analysis techniques do, but it only proves the presence of faults, not their absence.

There is a long tradition of building tests by hand, by choosing test cases that seem interesting to the eye of the developer. However, manual software testing is hard to maintain for any non-trivial project. The number and complexity of tests quickly becomes unmanageable. Automated test case generation, on the other hand, is able to explore a large space of the input domain and detect subtle bugs, at the cost of some kind of specification – cf. Specification-based Testing [25], Model-based Testing [53] or Property-based Testing [64] (PBT).

Specification-based Testing and Property-based Testing are of particular interest to us. The former deals with the generation of test strategies from formal specifications of a system in terms of its input and output. The latter handles input space exploration and test execution for general properties (often in the form of assertions), rather than testing by example. The appeal of these testing techniques is evident, given our context.

In Chapter 4 we propose a property specification language with formal semantics that *(i)* can serve as a basis to generate tests, and *(ii)* alleviates the main burden of Property-based Testing – i.e., to write properties of the system's behaviour. Furthermore, both techniques favour a black-box testing approach, which pairs well with HAROS, as it builds architectural models (Chapters 3 and 5) that define the system's inputs and outputs (ROS topics, in this case).

In this chapter, we propose a new approach to provide evidence that a system's behaviour complies with its behavioural specification. Inspired in Specification-based Testing, we will identify various test strategies for each property type. Each test strategy boils down to an *input trace schema* – an abstract trace of messages with constraints, intended to stimulate the system under test (SUT). Then, we use a PBT tool to automatically instantiate the schema, exercise the SUT with a variety of examples and report counterexamples. To detect whether a property is violated with the instantiated input trace, we rely on our Runtime Verification approach, described in Chapter 6.

This mixed approach, combining elements of Specification-based Testing, Property-based Testing and Runtime Verification, is implemented as a HAROS plug-in that we also present in this chapter. The plug-in, which is essentially a test generator, takes an architectural model and a behavioural specification as inputs. For each specified property, it determines the appropriate input trace schemas and generates a ROS PBT node, using metaprogramming and source code templates. Each node, in turn, contains data generators for the PBT tool to instantiate schemas, and is able to deploy the SUT and the appropriate runtime monitors.

An early prototype of this approach, focusing just on the Property-based Testing and test execution part, has been published in [150]. The specification language had not yet been designed at that stage, meaning that the tests did not target any property in particular right out-of-the-box, nor did they follow a specific test strategy. The current approach, in comparison, is much more mature. An evaluation of its current form in presented in Chapter 8.

7.1 State of the Art

The domain of Software Testing is known to be vast, with a multitude of approaches and techniques being proposed since the early times of software development. They range in scope (e.g., Unit Testing versus Integration Testing and System Testing), in generation method (manual versus automated) and assumptions about the system under test (white-box testing versus black-box testing), for example. In this section we can only skim the surface of this research topic, by discussing the work that relates the most to our own approach. Namely, we focus on Specification-based Testing and Property-based Testing, two testing techniques that, so far, have not yet been explored in the domain of ROS.

7.1.1 *Specification-based Testing*

Specification-based Testing, also known as Behaviour-based Testing, is a testing technique that is based on a formal specification of the system's behaviour in terms of its inputs and outputs (i.e., requirements). The specifications are based on mathematical semantics such as temporal logic. The main goal of this technique is to provide a systematic way of deriving test cases (a testing strategy), based on the characteristics of the inputs – for instance, by partitioning the specified inputs in different categories. Describing a system in terms of its inputs and outputs means that it is a black-box testing technique, i.e., implementation details are left out of the specification.

> **Note:** The term Specification-based Testing is sometimes used interchangeably with the term Property-based Testing. In this thesis, we assume different definitions, as seen, for instance, in [11]. Both techniques are based on desired properties of a program, but, as we will see, there is a subtle difference between the two. Specification-based Testing has a notion of identifying relevant input or test classes from the specification. Property-based Testing works closer to the execution level; properties are more akin to program assertions, and input exploration does not necessarily follow a strategy.

The testing approach that we take draws significant inspiration from Specification-based Testing. Like other Specification-based Testing approaches, we also automatically generate tests from a specification with mathematical semantics (see Chapter 4), treating systems as black-boxes with inputs and outputs. Only, in this case, inputs and outputs are ROS messages sent over topics, rather than, e.g., function calls. We are not aware of existing work in Specification-based Testing for Metric First-Order Temporal Logic specifications, but some approaches based on the simpler Linear Temporal Logic [12, 31, 125, 128, 161] provide the foundations of our test generation method.

Tan et al. [161] propose a property-coverage metric that measures how well a given LTL property is tested by the test suite. This metric, which is based on requirement mutation, drives their approach, as it is used to convert the original formula into a finite set of formulae characterising non-trivial tests. Formulae are fed to a model checker whose counterexamples are then used to generate concrete test cases. Although the number of tests is finite, witnesses from the model checker may have infinite length, if they contain loops. Their second contribution is a method of exploiting the infinite-length test structures, and boil them down to equivalent, finite tests.

Michlmayr et al. apply Specification-based Unit Testing to Java publisher-subscriber applications [125]. In the specification phase, the Java programs are annotated with LTL pre-conditions, post-conditions and invariants. Aiming to test components in isolation (the Unit Testing aspect of their approach), they replace the actual publisher-subscriber infrastructure with a mock. The execution traces are sequences of state snapshots of the tested component, rather than sequences of published/received messages. Such snapshots are obtained at runtime, by the mock infrastructure, by using Java's reflection capabilities – which, arguably, goes against the black-box testing principles. There are no timed updates in this approach, snapshots are only built after publish or notify events. In addition, their approach is limited in terms of specification; it evaluates safety properties but not liveness properties. Despite being an interesting approach, it is not applicable to our context for three reasons:

1. we do not focus on testing components in isolation, but rather on a testing approach that handles black-box systems of any size;
2. ROS applications are composed of multiple processes and not amenable to introspection via reflection, as opposed to Java in-memory objects;
3. it is straightforward to capture exchanged messages in a ROS system without replacing the whole publisher-subscriber infrastructure with a mock.

The usefulness of Specification-based Testing in the context of black-box reactive systems has been noted multiple times [12, 31, 128], but Bloem et al. deem requirements to be too abstract to fully predict the behaviour of these systems [31]. They argue that generated test cases have to adapt to the system behaviour at runtime, and should follow fault-based testing techniques (such as mutation testing). The argument in favour of fault-based testing is their ability to catch all faults of a given class. Thus, their approach asks the user for a system specification and a fault model defining coverage goals (rather than a behaviour model), both specified in LTL. These artefacts make the basis for their reactive synthesis process, which works with partial information while providing strong guarantees. One such guarantee is

that if the synthesis process is successful and tests run long enough, they reveal all faults defined in the given fault model.

As in our approach, Bloem et al. resort to Runtime Verification to build a test oracle, and the adaptive test strategies are based on state machines, where the goal is to explore different paths to a target state. A distinctive characteristic of their contribution is that, by default, the generated test cases are able to detect and categorise faults along four categories:

1. **Once** for faults that happen at least once;
2. **Repeatedly** for faults that happen multiple times during test execution;
3. **From some point on** for faults that happen after a certain triggering point, possibly multiple times;
4. **Permanently** for faults that happen in all cases.

The generated tests aim for the weakest category first, i.e., they try to falsify the requirements at least once. Then, they progressively move towards the stronger categories, by trying to replicate the faults multiple times in the same trace and to find their scope, until no further progress is achieved and the final fault category is determined. In addition to the four listed categories, users are also allowed to define new categories using LTL formulae.

The trend of online testing for reactive systems, i.e., the execution of test cases as they are generated, continues in the work of Arcaini et al. [12]. This approach is more similar in concept to the one we propose in this chapter. From a given LTL specification over sequences of method calls in Java programs, they start by generating an online monitor based on Büchi automata. The generated monitor serves two purposes:

- to detect violations in runtime, i.e., to play the role of an oracle;
- to provide an automaton whose states are used to generate test cases.

The online tests dynamically generate sequences of inputs that visit the different monitor states using either a random walk or a guided walk. In addition, they propose the following coverage criteria based on the monitor automata.

- **State Coverage**: each state of the monitor is visited.
- **Method Coverage**: each Java method that exits a given state is visited.
- **Transition Coverage**: each state transition is visited; this is not the same as Method Coverage because transitions may involve various methods, and a single method may be part of multiple transitions.
- **Atomic Proposition Coverage**: each atomic proposition of a transition's formula is visited.
- **n-Transition Coverage**: Transition Coverage, done n times in the same trace.
- **n-Atomic Proposition Coverage**: Atomic Proposition Coverage, done n times in the same trace.

Note that the coverage criteria are hierarchical. Method and Transition coverage subsume State coverage; Atomic Proposition coverage subsumes both Method and Transition coverage; and the n variants naturally subsume their singleton counterparts.

The goal of this procedure is to generate test sequences until full coverage is achieved for a given criterion. Using a random walk, as implied by the name, the criterion is the stopping rule of the process, while the steps of the automaton visit are chosen at random. Using a guided walk the criterion itself is used to select the visit paths. In the event that a goal is too difficult (or impossible) to achieve, a limit of retries is also fed to the generator after which it gives up. Some weaknesses of this approach include its disregard for method parameters, its inability to report which LTL property has been violated once a failure is detected and its focus on safety properties.

Lastly, Narizzano et al. build upon the approach of Arcaini et al [12] but target a more general class of properties [128]. Namely, co-safety properties (*"something good will happen"*), a proper subclass of liveness properties is also considered. This approach takes in a LTL specification and from it builds corresponding Büchi automata. As in the previous approach, the automata are visited to generate the concrete sequences of input and output pairs that constitute test cases. The proposed methodology consists in the following steps.

1. Start the visit from the Büchi automaton's initial state.
2. Generate a sequence of inputs to feed the black-box reactive system and obtain the corresponding sequence of outputs from the system.
3. Compare the input and output pairs with what the sequences predicted by the automaton; the absence of a corresponding state for an input-output pair produced by the system constitutes a violation.

These three steps are iterated, for different sequences of inputs and outputs, until either:

- an acceptance state is reached with a sequence of length equal to or greater than a given minimum; or
- an acceptance state cannot be reached with sequences of length equal to, or lesser than, a given maximum.

The whole process is then iterated to obtain multiple tests that visit each reachable transition at least once, thus achieving coverage.

7.1.2 *Property-based Testing*

Property-based Testing [64] (PBT) is the construction of test cases that check whether a piece of code (function, program, etc.) conforms with a certain property. This technique feeds the tested code data from a large corpus, possibly dynamically generated, possibly dependent on the results of execution on previous data, in an attempt to make it fail and reveal underlying problems. For instance, given any list xs, it should be true that reversing the list twice yields the original list, i.e., $\mathit{reverse}(\mathit{reverse}(xs)) = xs$. This general property is the test case, for which a PBT library would generate a number of input lists, of varying sizes and contents, in an attempt to find a counterexample. We know in advance that this property should be

true, so the expected outcome is for the PBT library to give up after a reasonable number of examples that are unable to falsify the property.

> **Note:** To further distinguish Specification-based Testing and Property-based Testing, we can take a closer look at the previous property example. From the requirements point of view alone, we know that the property *"reversing a list twice yields the original list"* should be valid for any list. As such, there is only one input category, the category containing all lists. A naïve Specification-based Testing approach would take this fact and generate only one test, for a random list, since all lists are treated equally. A naïve Property-based Testing approach, on the other hand, would disregard this fact entirely, and operate on a generator of lists. It would test a (possibly large) number of lists, in an attempt to find corner cases of the implementation (e.g., empty lists) that violate the property.

Property-based Testing was made popular with the QuickCheck tool [48] for the Haskell programming language. In fact, this technique is sometimes informally defined as *"what QuickCheck does"* [113]. A number of other implementations, heavily inspired in QuickCheck, has emerged since for other programming languages, e.g., [14, 112]. The Hypothesis tool [112, 114] is of particular interest in this chapter. Section 7.3 contains a brief introduction to it.

A common feature among Property-based Testing tools and libraries is called *shrinking*. Once a counterexample is found, the testing tool attempts to shrink it, in order to present the user something that is as close to a *minimal counterexample* as possible – an invaluable aid in debugging. For instance, by default, in Hypothesis:

- Boolean parameters shrink towards `False` (i.e. shrinking will replace `True` with `False` where possible);
- integers shrink towards zero, and negative values also shrink towards positive (i.e., $-n$ may be replaced by $+n$);
- lists shrink by trying to remove elements from the list, and by shrinking each individual element of the list (in no particular order, possibly both at the same time); and
- dictionaries shrink by trying to remove keys from the generated dictionary, and by shrinking each generated key and value (also in no particular order, and possibly all at the same time).

PBT has been applied with great success in various contexts. An interesting example is as a teaching tool [69], where it was used to evaluate student projects in concurrency. Concurrency is notorious for being hard to get right, especially without the right high-level primitives. In this study, students were being taught such high-level concurrency primitives, and were asked to implement shared resource specifications for an automatic warehouse with robots. Simply put, the idea is that a number of robots move with cargo from warehouse to warehouse using shared corridors. Each corridor only has space for one robot at a time, and each warehouse has a cargo weight limit. The final project submissions were automatically evaluated, both with a hand-crafted test suite and with the PBT tool Quviq QuickCheck[1] [14]. Quviq QuickCheck was

[1] http://www.quviq.com/products/erlang-quickcheck/

used to model a state machine that, in turn, is able to generate test cases. The PBT specifications aimed to exercise tests to find:

1. a warehouse that admits a new robot and goes beyond the weight limit, or a corridor that admits a robot even when already occupied by another robot;
2. function calls that the model permits but the implementation blocks;
3. multiple concurrent calls, to find concurrency bugs (a combination of tests 1 and 2).

While many project submissions passed the hand-crafted test suite without an issue, the authors found that more than half failed with Quviq QuickCheck.

Property-based Testing has seen successful industrial application as well [14, 86]. In a paper introducing Quviq QuickCheck [14], Arts et al. used PBT to test an implementation of a telecommunications protocol. They took a black-box testing approach, using Quviq QuickCheck to send messages to the telecommunications system and assert that its replies were valid, according to the protocol. Message generators were manually put together, and then validated, over multiple iterations, using Quviq QuickCheck itself. The system was tested both with a positive testing strategy – to check that it behaves as expected for valid input – and with a negative testing strategy – to assert that it responds with an error for invalid input. They were able to find errors that the original test team had also found, manually, but in a fraction of the time the original team spent. In addition, an older version of the system was also tested, under the same circumstances, to estimate how much Property-based Testing would have helped, in terms of bug detection, had it been used earlier. With an investment of just six hours, they were able to uncover nine faults, some of which corresponded to trouble reports in the developer's records. The authors estimate that PBT could detect at least half of the faults associated with all trouble reports, as well as additional faults that had not yet shown as trouble reports. In an interesting concluding remark, the authors state that not only is PBT beneficial for its earlier bug detection, it also forces users to formalise restrictions that would otherwise go undocumented.

John Hughes, one of the authors of QuickCheck, has also published a report [86] of their experiences and lessons learned with QuickCheck and Quviq QuickCheck, applied to industrial case studies over 15 years. In particular, they go over two industrial case studies in which QuickCheck was used to detect a large number of bugs. The first case study tested the AUTOSAR Basic Software that runs in every processor of Volvo Cars vehicles. With about 20 000 lines of QuickCheck specifications, they were able to test a million lines of C code and find more than 200 problems, of which more than half were inconsistencies or ambiguities in the AUTOSAR standard itself. The second case study helped detect a bug in Mnesia[2], the databse system that is distributed with the Erlang environment. This bug had been notoriously hard to find, as it is related to a race condition when accessing records. Where experts spent more than six weeks hunting for the bug and ending up with no clue as to what the cause might be, QuickCheck was able to find a minimal counterexample (in about 10 minutes of runtime) that, then, lead to a fix within a single day – demonstrating the value of shrinking in PBT. Some of the valuable learned lessons are as follows.

1. The same property can find many different bugs.

2 https://erlang.org/doc/man/mnesia.html

2. QuickCheck finds discrepancies – most errors are in the model, not the implementation.
3. Having minimal failing examples makes debugging easier.
4. Formulating specifications is hard and many developers struggle with it.
5. Working with concrete examples is far simpler than to work with general properties, but it is impracticable to test everything by example.

Duregard [57] considers Property-based Testing to be easy to learn, but hard to master. PBT requires little more than boolean functions to get started. One of the main challenges of PBT is to define adequate data generators for user-defined data types. To address this, they propose a few algorithms (implemented in Haskell) to derive generators automatically. Another issue is to ensure good test quality – the absence of a counterexample does not mean that there is none to be found. To address this significant problem, they introduce a black-box variant of mutation testing. By using higher-order functions (easily achievable in Haskell and functional programming), they are able to mutate the system's functions without access to the source code. Then, by checking whether a test suite can detect the mutated bug, they are able to estimate property strength – i.e., how accurate and precise a property is, or how close the specification is to the implementation. Properties can range from tautologies (weak properties) to full specifications (strong properties), although they find that most properties are somewhere in between; they specify part of the behaviour. While a combination of black-box Mutation Testing and Property-based Testing is less rigorous than a white-box variant, it is found to be much more efficient.

Another drawback of Property-based Testing is that input generation is mostly random. An enhanced variant of PBT, called Targeted Property-based Testing (tPBT) [109] overcomes this problem by using a guided input search strategy, rather than the traditional random generators. In essence, each input is attributed a score (based on the respective output), which states how close it was to falsifying a property. Then, the search strategy dynamically generates inputs that aim to maximise this score, using a *neighbouring function* over the input domain (a function that produces the next input in a desired direction). But this trades one problem for another, as for tPBT to work the user needs to specify the search strategy and also supply all ingredients that the search strategy requires. This is laborious and makes tPBT less attractive. However, Löscher and Sagonas propose a tool with an automated search strategy [110], aiming to overcome this problem. The Hypothesis tool also provides experimental support for this approach.

Perez and Nilsson apply Property-based Testing in the context of Functional Reactive Programming (FRP) [138] – a functional programming paradigm that is often used in interactive applications, such as games and graphical user interfaces. Interactive systems are challenging to test. They lack reproducibility, and generating suitable test data might be hard. To alleviate this issue, a variant of this paradigm, called Pure Arrowized FRP, separates data processing from all side effects and time sampling code, in an attempt to remain true to the principles of pure functional programming. This approach makes runs deterministic and, thus, easier to test and debug. The authors use a causal subset of LTL to extend FRP programs with temporal assertions, an evaluation function to express properties, and the ability to record and reproduce input traces. QuickCheck is then used as a generator – starting from a real user trace, QuickCheck takes a prefix of the trace and produces the subsequent input randomly. In other words, given an interactive

application, such as a game, and a prefix for an input trace (so that the application is lead to a specific state), QuickCheck is essentially able to *play* the game.

Alzahrani et al. try a combination of Property-based Testing and Model Checking for property verification [7]. Their proposed workflow starts with an informal specification of the system, which is properly encoded in a formal modelling language, such as TLA+ [100][3]. Temporal properties are verified on this model, using standard model checkers or theorem provers. The formal temporal models are, then, translated into a Scala extension, so that they obtain an executable model of the system. PBT is applied not directly to the system, but rather to the executable model, in order to find defects.

Closer to the domain of robotics and cyber-physical systems, Löscher et al. apply Property-based Testing to sensor networks [111]. Specifications are operational, written in Erlang. To avoid dealing with hardware for the purposes of testing a software system (an issue that is also meaningful in robotics), they rather test the network within a simulated environment. With this approach, they are able to test both properties of individual functions (unit testing) as well as network-global properties (a sort of integration testing).

7.2 Preliminaries

Our approach is based on Specification-based Testing and Property-based Testing principles, as previously mentioned. We use the former to convert formal properties into specialised test strategies, here called *input trace schemas*. We use the latter to automatically explore the input space and test the system with a variety of valid inputs. Combining the two, we are able to nudge the PBT input generators in a direction that is more likely to reveal counterexamples. In this section we define some core concepts to our approach that we will reference throughout the remainder of the chapter.

7.2.1 *Open Topics and Open Subscriptions*

An *open subscription* (or *open subscribed topic*) is any ROS topic in the architectural model that the system subscribes, but for which there are no corresponding publishers. Consider the Computation Graph in Figure 67 as an example. It contains three subscriptions: `/planner` subscribes `/position`, while `/control` subscribes `/intent` and `/sensors`. Only `/position` and `/sensors` (depicted in blue) are open subscriptions; there are no publishers on these topics. The other topics have corresponding publishers, and, thus, do not fit into this definition.

Open subscribed topics represent the inputs of the systems we test. They are the intended interfaces to integrate with the system and to stimulate it, since there is no other purpose for the existence of subscribers that lack their respective publishers. Topics for which there are already some corresponding publishers within the system are *closed* (from the point of view of our approach). That is, even though they are technically usable by external components (such as test drivers or runtime monitors), they are treated as

3 https://lamport.azurewebsites.net/tla/tla.html

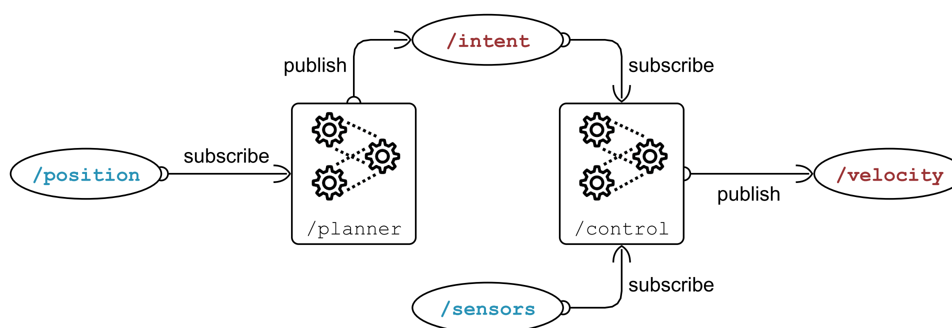


Figure 67: Example ROS Configuration with open subscribers.

internal communication channels of the system. We do not tamper with them by publishing additional messages, because the system is likely not designed to expect the additional publishers.

7.2.2 *Input Traces and Schemas*

An *input trace* is a finite sequence of generated messages, annotated with timestamps, that the test driver should replay on open subscribed topics to stimulate the system under test. Note that this is distinct from the *observed trace*, the actual sequence of events that makes a particular run of the system. All events of the input trace are part of the observed trace, but not the other way around. Whereas the input trace only lists inputs, the observed trace lists all observed messages regardless of their origin. It includes the messages sent by the test driver as well as those generated by the system under test.

An *input trace schema* is an abstraction for a certain category of input traces. Input traces contain concrete messages and timestamps, with concrete values assigned to all data fields. For instance, considering the architecture in Figure 67, the following sequence of messages and timestamps is a valid input trace.

```
1 @50ms  /sensors  {bumper: 0, laser: 255}
2 @100ms /position {x: 0.0, y: 0.0}
3 @150ms /sensors  {bumper: 0, laser: 255}
4 @200ms /position {x: 2.5, y: 0.5}
5 @250ms /sensors  {bumper: 0, laser: 128}
```

A schema, on the other hand, describes a general sequence of events using abstract messages and durations, annotated with their respective constraints. It can contain constraints over message fields or constraints over time. Input traces are, thus, *instances* of a schema. For instance, the following would be a valid schema containing the previous trace.

```
1            forbid /position
2 +50..150: publish /position {x >= 0.0 and y >= 0.0}
3            forbid /position
4            forbid /sensors  {bumper > 0 or laser < 200}
5 +50..:    publish /position {x > 1.0 and y >= 0.0}
6            forbid /position
7            forbid /sensors  {bumper > 0 or laser > 200}
```

The notation is simple; a schema is interpreted line by line, each line representing general constraints over topics (**forbid**) or mandatory events of interest (**publish**). The syntax following either keyword is the syntax used in our specification language, i.e., it expects a topic name, followed by predicates over the message's fields.

The **forbid** statement imposes restrictions over the (zero or more) random messages an input trace is allowed to instantiate within a segment of the trace. It applies from the instant immediately following the previous mandatory event until (but not including) the next mandatory event. If there are no previous events, as is the case in the first line of this example, the restrictions apply from the system's start-up event until the next event. If there are no follow-up events, as is the case in the last two lines, the restrictions apply until the end of the trace. In the first line, the schema states that there should be no messages on `/position` (one of the two open subscribed topics) at first. There are no restrictions on `/sensors` before the first **publish** statement, meaning that any number of messages is valid in this interval, regardless of their contents. The second and third **forbid** statements apply in between the two **publish** statements. The last two **forbid** statements apply indefinitely, after the second **publish** event.

The **publish** statement is composed of an interval and of constraints over a topic, with a colon symbol separating the two. It denotes a single mandatory message, published exactly at the time of the event. The event itself can happen at any of the timestamps contained in the interval. Each interval is denoted by the plus sign followed by the interval limits (in milliseconds). The plus sign is used to denote that intervals are relative to the previous event. In this case, '`+50..150`' denotes $[50, 150)$ counting from the previous **publish** event, or from the system's start-up event if there are no previous **publish** events. A possibly infinite interval is denoted simply with its lower bound, e.g., '`+50..`' denotes $[50, \infty)$. Intervals with an open lower bound, i.e., `+..t`, are not accepted. The first **publish** event happens sometime between the 50 millisecond mark of runtime and the 150 millisecond mark; there are no previous events, so it counts from the system's start-up event. At the time of the event, the test driver should publish a message on `/position` such that $x \geq 0$ and $y \geq 0$. In addition, this event lifts the previous **forbid** restrictions (and is not affected by them). The second **publish** event happens at least 50 milliseconds after the first one, with no imposed upper bound. It lifts the restrictions of the second and third **forbid** statements, and enables the following two, right after its message is published.

We can see how the previous trace is an instance of this schema. The trace contains only two messages on `/position` at the 100 and 200 millisecond marks. These correspond to the two mandatory events in the schema. The relative delays for both events were instantiated as 100. The first event counts from the system's start-up event, so the final timestamp is exactly 100 milliseconds. The second event happens 100

milliseconds after, resulting in the `@200ms` timestamp. The `x` and `y` fields uphold their respective restrictions too. There are no further messages on this topic, satisfying the three `forbid /position` constraints (before the first message, after the second message and in between messages). The input trace contains additional messages on `/sensors`, at 50, 150 and 250 milliseconds, which are not prohibited by the schema. There are no constraints on `/sensors` before the first `publish`, so anything is allowed. In between `publish` statements, the input trace cannot contain `/sensors` messages such that '`bumper > 0 or laser < 200`'. There is a message on this topic, but it is not a match for the predicate, so it is allowed. The same logic applies for the final message, after the second `publish` statement.

Formally speaking, schemas could be interpreted as a set of Metric First-Order Temporal Logic constraints over temporal first-order structures (see Chapter 4).

Definition 7. *An input trace schema is a set of formulae over a signature S.*

Definition 8. *Let S be a signature, v be a valuation and Γ be an input trace schema over S. A temporal first-order structure (D, τ) over S is said to be an instance of Γ if and only if, for all $\varphi \in \Gamma$, it is true that $(D, \tau, v, 0) \models \varphi$.*

The previous definition allows us to determine whether a temporal first-order structure (an infinite trace) is a instance of a schema. But the input traces we generate are finite. We need to define in which cases can an input trace be considered an instance of a schema.

Definition 9. *An input trace of length k is a k-prefix of a temporal structure over a signature S, for some $k \in \mathbb{N}_0$.*

Definition 10. *Let S be a signature, v be a valuation and Γ be an input trace schema over S. Let $k \in \mathbb{N}_0$ and $(D_{[0,k)}, \tau_{[0,k)})$ be a k-prefix of a temporal structure over S. The k-prefix $(D_{[0,k)}, \tau_{[0,k)})$ is said to be an instance of Γ if and only if, for all (D', τ') temporal structures over S, $(D_{[0,k)}, \tau_{[0,k)})$ is a k-prefix of (D', τ'), and it is true that $(D', \tau', v, 0) \models \varphi$, for all $\varphi \in \Gamma$.*

As to how to convert schemas into MFOTL formulae, let us start by defining an atomic variable e_0 for the initial instant of the trace and atomic variables e_i for each `publish` event in a schema. Each atomic variable e_i is true only at the instant when the ith `publish` event happens, i.e., at the exact instant the specified message is published. In MFOTL terms, after normalising open intervals $+\delta$`..` to $[\delta, \infty)$, given statements of the form

```
1 +a_i..b_i: publish topic {φ}
```

We have that, for all $i \in \mathbb{N}$:

- $\Box(e_i \rightarrow \bigcirc(\Box\neg e_i))$
  i.e., once e_i happens, it will not happen again;
- $\Box(e_i \rightarrow (\exists x : \mathit{topic}(x) \land \varphi))$
  i.e., at the instant e_i happens, there is a message on `topic` such that φ.

The first step is to schedule all mandatory events. Let $n \in \mathbb{N}_0$ be the number of mandatory events in a schema. Then, for all $i \in \mathbb{N}$ such that $i \leq n$:

$$\Box(e_{i-1} \rightarrow \Diamond_{[a_i,b_i)} e_i)$$

After ensuring that all mandatory events happen, we translate the constraints given by **forbid** statements. For each statement,

```
1 forbid topic {φ}
```

If the statement is placed in between events e_i and e_{i+1}, it is true that:

$$\Box(e_i \rightarrow \bigcirc((\neg\exists x : \mathit{topic}(x) \land \varphi) \,\mathcal{W}\, e_{i+1}))$$

The $\bigcirc$ operator is necessary to ensure that the restrictions do not apply at the instant of the **publish** event corresponding to e_i. Rather, they start immediately afterwards. The weak-until operator, $\mathcal{W}$, ensures that the restrictions do not apply at the instant of the following **publish** event corresponding to e_{i+1}. Also, note that consecutive logical instants can be attributed the same timestamp. Imposing restrictions immediately after a **publish** statement does not imply different timestamps.

If there are no events preceding a **forbid** statement, the resulting formula is as follows.

$$(\neg\exists x : \mathit{topic}(x) \land \varphi) \,\mathcal{W}\, e_1$$

If there are no events succeding a **forbid** statement, the resulting formula is as follows.

$$\Box(e_i \rightarrow \bigcirc(\Box(\neg\exists x : \mathit{topic}(x) \land \varphi)))$$

If there are no mandatory events in the schema, a **forbid** statement is translated as:

$$\Box(\neg\exists x : \mathit{topic}(x) \land \varphi)$$

Thus, given our previous example, with two **publish** statements and five **forbid** statements, we would have the following set of formulae. For readability, consider φ_1 and φ_2 as the message predicates of each **publish** statement, while ψ_1 and ψ_2 represent the **forbid** restrictions on `/sensors`.

- $\Box(e_1 \rightarrow \bigcirc(\Box\neg e_1))$, i.e., e_1 only happens once.
- $\Box(e_2 \rightarrow \bigcirc(\Box\neg e_2))$, i.e., e_2 only happens once.
- $\Box(e_1 \rightarrow (\exists x : \mathit{position}(x) \land \varphi_1))$, i.e., at e_1 the specified message is published.
- $\Box(e_2 \rightarrow (\exists x : \mathit{position}(x) \land \varphi_2))$, i.e., at e_2 the specified message is published.
- $\Box(e_0 \rightarrow \Diamond_{[50,150)} e_1)$, i.e., e_1 happens within the first 50 to 150 milliseconds.
- $\Box(e_1 \rightarrow \Diamond_{[50,\infty)} e_2)$, i.e., e_2 happens at least 50 milliseconds after e_1.
- $(\neg\exists x : \mathit{position}(x)) \,\mathcal{W}\, e_1$, i.e., the first **forbid** statement.
- $\Box(e_1 \rightarrow \bigcirc((\neg\exists x : \mathit{position}(x)) \,\mathcal{W}\, e_2))$, i.e., the second **forbid** statement.
- $\Box(e_1 \rightarrow \bigcirc((\neg\exists x : \mathit{sensors}(x) \land \psi_1) \,\mathcal{W}\, e_2))$, i.e., the third **forbid** statement.

- $\square(e_2 \rightarrow \bigcirc(\square(\neg\exists x : \mathit{position}(x))))$, i.e., the fourth **forbid** statement.
- $\square(e_2 \rightarrow \bigcirc(\square(\neg\exists x : \mathit{sensors}(x) \wedge \psi_2)))$, i.e., the fifth **forbid** statement.

7.2.3 *Testable Properties*

Every property defined with our specification language is monitorable, as previously established in Chapter 6. However, for our current approach, only a subset of the language is deemed *testable*. In this context, and for our current approach, we consider a *testable property* to be any property for which we can have complete control of its scope and preconditions. In practice, this translates to a property such that:

- its activator event is either unspecified or a message on an open subscribed topic;
- its terminator event is either unspecified or a message on an open subscribed topic;
- its behaviour event is a message published by the system, i.e., a message *not on* open subscribed topics;
- for Precedence properties, its precondition event is a message on open subscribed topics;
- for Response and Prevention properties, its stimulus event is a message on open subscribed topics;
- predicates over the fields of activator, terminator, precondition or stimulus messages do not contain arithmetic or quantified expressions.

The rationale behind these limitations is simple to understand. The only messages produced by the system (i.e., the only outputs) are behaviour events. If all other events are inputs of the system, rather than outputs, the test driver can dictate the pace of the test. More importantly, tests are active and reproducible. By controlling the activator (resp. terminator) event, the input trace schema dictates when the scope should start (resp. end). By controlling also stimuli events (if applicable), the input trace schema dictates when behaviours should be triggered.

It is always possible to close more topics in a property (or replace open subscribers with open publishers) and still perform tests. In fact, it is a planned future improvement for our approach. But, in doing so, tests become passive and rather uninteresting. The more we depend on messages produced by the system (e.g., to open scopes or to trigger behaviours), the more linear test schemas become; there are less events under the control of the test driver. At the limit, all topics referenced in a property are controlled by the system. In that case, *testing* a property is, essentially, just *monitoring* the system. The main difference is that the test driver can still produce messages on open subscribed topics *not* referenced by the property (if there are any). As such, we opt not to support test generation for such properties. We focus, instead, on the most interesting problem, using the aforementioned restrictions over topics, which involves a variety of input trace schemas and carefully designed data generators.

The restriction on predicates containing quantifiers or arithmetic expressions is more of a technical one. Simply put, arithmetic makes data generation much harder to implement. In general, it is possible to specify a system of equations (or inequations) such that message fields are cross-referenced in multiple places and hold various relations with other fields of the same message, or even with fields of past messages. We could *brute force* data generation. For instance, we could generate numbers randomly until one is

generated such that the predicates hold. For obvious reasons, this could turn out to be very slow, and even unfeasible, especially if the given predicate is unsatisfiable. Instead, we (automatically) produce data generators that select inputs from an appropriate domain right from the start, as we will see later in this chapter. Coming up with *smart* generators for general arithmetic expressions would require the use of additional tools, such as SMT solvers (Satisfiability Modulo Theories, e.g., Z3 [54]) or symbolic computing libraries (e.g., SymPy [123]) to figure out appropriate domains (i.e., to solve equations and inequations). A similar reasoning applies to quantified expressions. We leave the exploration of these problems for future work.

7.2.4 *Axioms*

Axioms are properties such that *all* events refer to open subscribed topics, i.e., all events are system inputs. By definition, these properties are not testable. We use axioms as global constraints over the input trace schemas for testable properties, and to invalidate tests whose actual input traces do not conform with the axioms. That is, given axioms $\Phi_1, \ldots, \Phi_n$, in order to test a property Ψ, the generated input trace schemas should conform with $\Phi_1, \ldots, \Phi_n$, in addition to any relevant constraints of Ψ. As an example, consider again the architecture of Figure 67, with its open subscribed topic `/position`. The property '`globally: no /position {x < -10.0 or x > 10.0}`' is a valid axiom. It is a property solely over open subscribed topics, and is, thus not testable. This axiom can be used to force all `/position` messages, in all input trace schemas for other properties, to contain x ∈ [-10, 10]. In the same fashion, axioms can also be used to express causality (`causes`), to set messages apart from each other for a given period (e.g., '`/position forbids /position within 100 ms`'), or to express any other property. As we will explain later, axioms will be used to refine general input trace schemas, narrowing down the domain of valid inputs by construction.

7.3 Approach Overview

Our approach to automatically test whether a ROS application violates a given property combines Specification-based Testing, Property-based Testing and Runtime Verification. Inspired in Specification-based Testing techniques, especially [12, 128], we are able to convert formal properties into state machines, and then develop input trace schemas based on the state machines. Using the schemas, we are able to restrict the semi-random inputs produced by the PBT library to a domain that is more relevant to the tested property.

In this section, we discuss the overall workflow of this approach, the semantics of the testing procedure, and the technologies behind it. A more in-depth discussion about the schemas for each type of property is given in Section 7.4. Concrete details of the implementation are given in Section 7.5.

7.3.1 *Workflow*

Our approach to automatically test whether a ROS application violates a given property involves a few steps, as we previously stated: *(i)* deriving input trace schemas from a specification; *(ii)* instantiating concrete input traces from the schemas; and *(iii)* replaying input traces to the SUT while it is monitored for violations. Point *(i)* is based on Specification-based Testing principles. For each type of property, we will identify the different categories of traces that could potentially lead the SUT into erroneous behaviour. Points *(ii)* and *(iii)* are based on Property-based Testing principles. For the former, we must provide adequate data generators. In this case, we require generators of input traces that comply with constraints over time and data. The final point is more straightforward. It involves putting together a typical PBT test driver that, in addition to setting up the SUT, also deploys the required runtime monitors alongside the system (see Chapter 6 for the Runtime Verification approach). Then, the test driver must ensure that the generated input trace is dispatched to the SUT at the right times, based on the timestamps of each message.

In practice, point *(i)* can be implemented with a test generator. It only needs to be done once (for each version of the system and the specification), not every time the system is tested. Given a ROS Configuration and a specification, we start by identifying all testable properties. The Configuration is necessary for this step in order to be able to tell which topics are open subscriptions. Open subscribed topics, as stated previously, determine which properties are axioms, which are testable properties and which are non-testable properties. Once all properties are labelled, for each testable property, the generator should produce a number of schemas, based on the target property itself as well as on the applicable axioms. Some types of axioms are used to refine the default schemas automatically, as we will see in Section 7.4. Axioms that we cannot integrate directly into the schemas will be monitored in runtime.

To instantiate schemas as concrete traces of ROS messages, and to address point *(ii)*, each computed schema should be translated into a number of adequate data generators for a Property-based Testing library. To accomplish it, we require information about the message types associated with each topic, so that the data generators reference the correct data types and populate the messages with the expected data fields. Using metaprogramming and a template engine (much like we did to generate runtime monitors in Chapter 6), the generator should plug the data generators into a test script template.

The generated test scripts constitute the PBT drivers that will, repeatedly, test the SUT with semi-randomly generated traces, addressing point *(iii)*. Each testable property yields one test script. Each test script tests a single property, but can (optionally) monitor additional properties to take further advantage of the time spent testing. In addition, each test driver is itself a ROS node. This is not surprising, given that the driver must be able to publish messages to the system. At the same time, the driver must deploy runtime monitors to observe the system, and be able to gather a verdict from the monitors. Monitors are themselves ROS nodes, so the test driver must also subscribe to the verdict topics. Unlike runtime monitors, however, tests must terminate and define what to do in the event that a monitor verdict has not been reached. In general, as we will see, inconclusive monitor verdicts translate into passing tests.

In Figure 68 we illustrate the actual Computation Graph running for each test, including testing and monitoring nodes, using the example architecture of Figure 67. Note that, for the purposes of this approach,

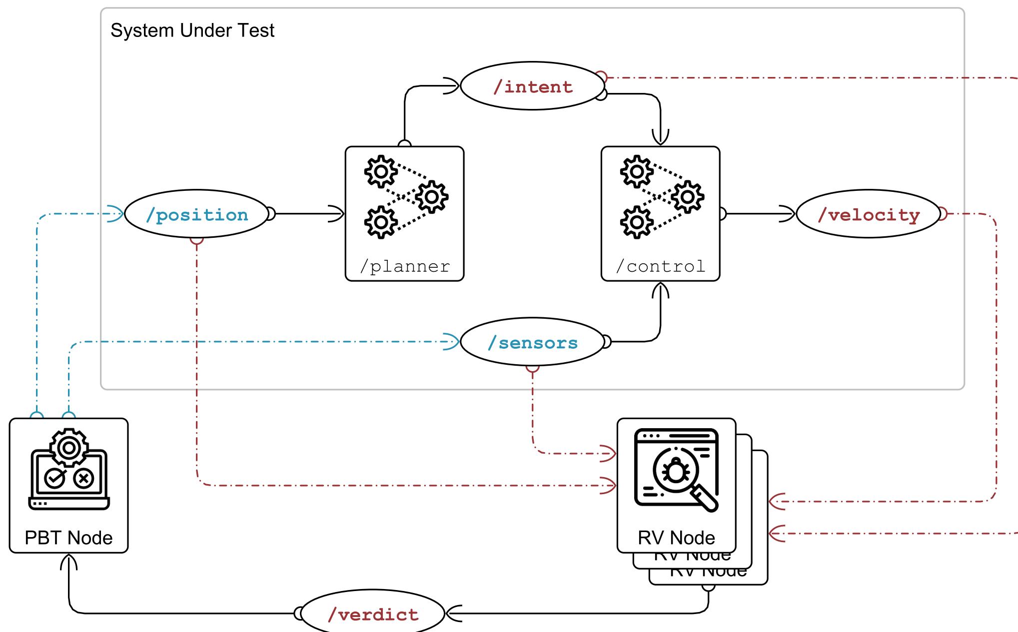


Figure 68: Runtime perspective of the proposed testing approach.

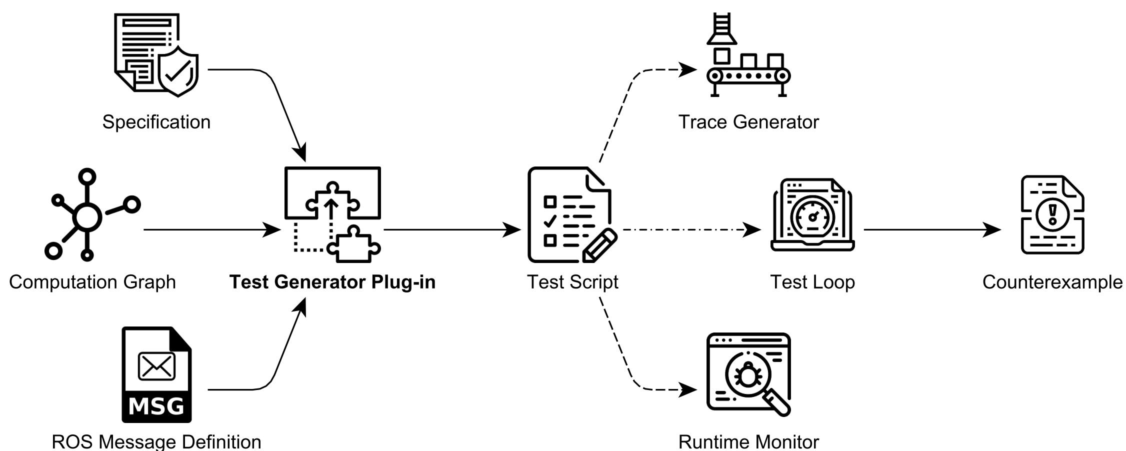


Figure 69: Workflow of the proposed Property-based Testing HAROS plug-in.

it makes no difference whether we test a single node at a time, or a full system. The SUT is a black-box with topics, for all intents and purposes.

Given our requirement on Configurations, specifications and message type information to achieve test generation, we have opted to implement the proposed test generator as a HAROS plug-in. HAROS is capable of providing all the necessary input, allowing us to focus the implementation effort on the generator logic. The workflow we just described, from properties to tests, is depicted in Figure 69.

In terms of technologies, for our template engine we rely on Jinja. For the Property-based Testing infrastructure we rely on Hypothesis [112, 114], given that it is among the most powerful free, industrial-strength and open source PBT libraries.

7.3.2 *Semantics*

Software Testing is only useful to demonstrate the presence of faults, not to prove their absence. Property-based Testing, in particular, embraces this philosophy well. Given a property of the system under test, it tries a variety of inputs with the goal of falsifying the property, i.e., aiming to find a counterexample. Typically, the user sets an upper limit on the number of examples that the PBT tool is allowed to try, before giving up. If no counterexamples are found, then the property is probably true.

We have already defined semantics for the specification language (Chapter 4) and for the Runtime Verification aspect of this approach (Chapter 6). However, both work with a notion of infinite traces. Tests are inherently finite and often establish a deadline for each test case. After this deadline, the SUT and the test driver are brought down, and a verdict is produced. This means that, in practice, we are only testing bounded properties (even if the specification is unbounded).

To be able to produce a verdict for all tests, we will adopt a *weak* version of the semantics for traces. That is:

- if the runtime monitors produce a verdict of $\bot$ before the test reaches its deadline, the verdict of the test is $\bot$;
- if the runtime monitors produce a verdict of $\top$ before the test reaches its deadline, the verdict of the test is $\top$;
- otherwise (i.e., the monitor did not produce a verdict before the deadline), the verdict of the test is $\top$.

An immediate result of these semantics is that tests will never report a property as false, unless the monitors do so as well. Another consequence is that it is useless to test unbounded liveness properties, in the same sense that it is useless to monitor them. As an example, suppose we want to test '`globally: some /velocity`'. There are two possible scenarios; either a message on `/velocity` is observed within the test's duration or it is not. If a message is observed every time (for every input Hypothesis tries), then the verdict is $\top$, without a doubt. If, for at least one of the examples, the message is not observed, the monitor's verdict would have been inconclusive. By our semantics, this also translates to a test result of $\top$. In other words, it is impossible for the test infrastructure to come up with a counterexample for this property.

This limitation is intended. As previously stated, the goal of testing, and PBT in particular, is to falsify properties and come up with counterexamples. By choosing a weak semantics, we avoid a number of false positives and bogus counterexamples that would send developers in a hunt for faults that might not exist. On the other hand, we are more open to false negatives and a false sense of reliability. How accurate the test results are depends on the given parameters for the test deadlines, and the maximum number of examples Hypothesis tries for a single property, but such is the nature of testing.

One of the advantages of Property-based Testing (and of Hypothesis) is that, by default, it attempts to minimise both false positives as well as false negatives. False positives and false negatives can be introduced by the non-deterministic and non-real-time nature of ROS. Nodes can publish messages in a

given order, and subscribers (such as monitors) can observe messages in a different order. The network layer might also cause sporadic delays that go beyond the specified time bounds (e.g., '`within 100 ms`'). However, when a specific input causes the test to fail, Hypothesis repeats it multiple times, to ensure (with some degree of confidence) that it is, indeed, a counterexample. In a similar fashion, repeating tests over and over, for a large enough number of times, will cause the likelihood of a false negative going unnoticed to reduce drastically.

7.3.3 *A Short Introduction to Hypothesis*

Hypothesis is a multilingual library, but its Python variant (the one we use) is the most mature and developed one. Overall, writing Hypothesis tests is relatively straightforward. Its syntax is not very intrusive or different from regular Python programs. Tests are, in fact, regular Python functions decorated with a special annotation, `@given`, as seen in Listing 7.1. When the `test_addition_is_commutative` or the `test_reversing_twice_gives_same_list` functions are called, Hypothesis handles the PBT logic in the background. That is, a call to the annotated function will not run the function once, as expected, but rather call it multiple times (the exact number is configurable) with various generated arguments, until a counterexample is found or Hypothesis exhausts its examples. The `@given` annotation also assigns data generators to each of the input parameters of the annotated function (`x` and `y` in the first case, `xs` in the second). In Hypothesis jargon, data generators are called *strategies*, a term that stems from the more general term *input search strategy*.

```
1  from hypothesis import given
2  from hypothesis.strategies import integers, lists
3
4  @given(x=integers(), y=integers())
5  def test_addition_is_commutative(x, y):
6      assert x + y == y + x
7
8  @given(xs=lists(integers()))
9  def test_reversing_twice_gives_same_list(xs):
10     # This will generate lists of arbitrary length (usually between 0 and
11     # 100 elements) whose elements are integers.
12     ys = list(xs)
13     ys.reverse()
14     ys.reverse()
15     assert xs == ys
```

Listing 7.1: Trivial example of using Hypothesis.

Hypothesis provides strategies for most built-in types with parameters to constrain or adjust the generated values, as well as higher-order strategies that can be composed to generate more complex types. The basic example of Listing 7.1 features only the strategy to generate integers, with the default arguments, and the strategy to generate lists, given only its first positional argument which is a strategy for the elements of the list. The `integers` strategy accepts optional arguments to define a minimum value and a maximum value. Examples from this strategy will shrink towards zero, and negative values will shrink towards positive

(i.e., $-n$ may be replaced by $+n$). The `lists` strategy can be customised in terms of its minimum length, maximum length and whether all elements must be unique. Examples from this strategy shrink by trying to remove elements from the list, and by shrinking each individual element of the list, in no particular order, and possibly both at the same time.

Overall, Hypothesis provides strategies for byte sequences, booleans, integers, floating-point numbers, text strings, complex numbers, fractions, dates, lists, tuples, dictionaries, sets, email addresses, among others. It provides strategies to choose elements from a limited (and known) input domain, to emulate enumerations. It provides ways to build strategies from other strategies, recursively or given functions to transform data, e.g., via mapping or filtering. When none of the above suffices, or to generate custom data types (as is our case), Hypothesis provides the `@composite` strategy.

```
1  from collections import namedtuple
2  from hypothesis.strategies import composite, floats, text
3
4  Location = namedtuple('Location', ['name', 'latitude', 'longitude'])
5
6  @composite
7  def locations(draw):
8      name = draw(text())
9      latitude = draw(floats(min_value=-90.0, max_value=90.0))
10     longitude = draw(floats(min_value=-180.0, max_value=180.0))
11     return Location(name, latitude, longitude)
```

Listing 7.2: Using Hypothesis strategies to build ROS messages.

As seen in Listing 7.2, `@composite` allows the user to define strategies using arbitrary code and core strategies. We use it, in this example, to create a strategy for geographic locations containing a textual name and floating-point latitude and longitude coordinates. For names we `draw` a value (an example) from the `text` strategy, without any of its optional arguments. By default, `text` can generate strings of any length from the full unicode range, excluding surrogate characters. Examples shrink towards shorter strings, and with the characters in the text shrinking as per the provided alphabet strategy (if any, not in this case). For coordinates, we use the `float` strategy, and we delimit the generated values using the self-explanatory parameters `min_value` and `max_value`. As for how floating-point numbers shrink, we should quote Hypothesis' documentation[4]:

> Examples from this strategy have a complicated and hard to explain shrinking behaviour, but it tries to improve "human readability". Finite numbers will be preferred to infinity and infinity will be preferred to `NaN`.

Putting it all together, with the `locations` strategy we are able to build `Location` objects, properly populated with a textual name, and floating-point latitude and longitude. Since `@composite` is used to build arbitrary objects with an arbitrary number of other strategies (possibly also composite strategies), there is only one sensible shrinking behaviour. Examples from this strategy shrink by shrinking the output of each `draw` call.

4 https://hypothesis.readthedocs.io/en/latest/data.html#hypothesis.strategies.floats

In this case, `locations` shrink by shrinking `name`, `latitude` and `longitude`. Again, Hypothesis is free to decide order and combinations. It might try to shrink one field at a time, or a combination of some. Following this logic, the (most likely) *minimal example* Hypothesis might try would be `Location('', 0.0, 0.0)`.

There are only two more concepts worth discussing regarding Hypothesis. The first is the `assume` function, that works similarly to **`assert`**. When the condition given to `assume` does not hold, Hypothesis marks the generated example as invalid, rather than failing the test, and tries to avoid similar examples in the future. This confirms that the input space exploration of Hypothesis is not truly random. The second concept is the `@settings` annotation. This annotation is used to wrap a test function (like `@given`) and define a number of settings for that test case, such as the maximum number of examples, the test deadline or whether to use a fixed seed with random number generators, so that tests are reproducible.

7.4 Pattern-based Trace Schemas

Based on Specification-based Testing techniques, especially [12, 128], we transform testable properties into state machines, and then use the state machines to identify different test schemas. We will refer to the state machines we presented in Chapter 6 as our basis. Naturally, considering that the main goal of Property-based Testing is to falsify properties, we will explore all paths of the various state machines that lead to a verdict of $\bot$. Our goal is to construct schemas that exercise such paths.

In some cases, state machines contain loops, such as entering and exiting a scope multiple times (with **`after-until`**). When they do, there is an infinite number of paths that we must consider, in theory. But testing is inherently finite, and so should be the schemas we identify. To illustrate our approach, we only go up to a limit of two loops, i.e., we will unroll the state machine paths and consider entering the scope once, twice or thrice within a single execution.

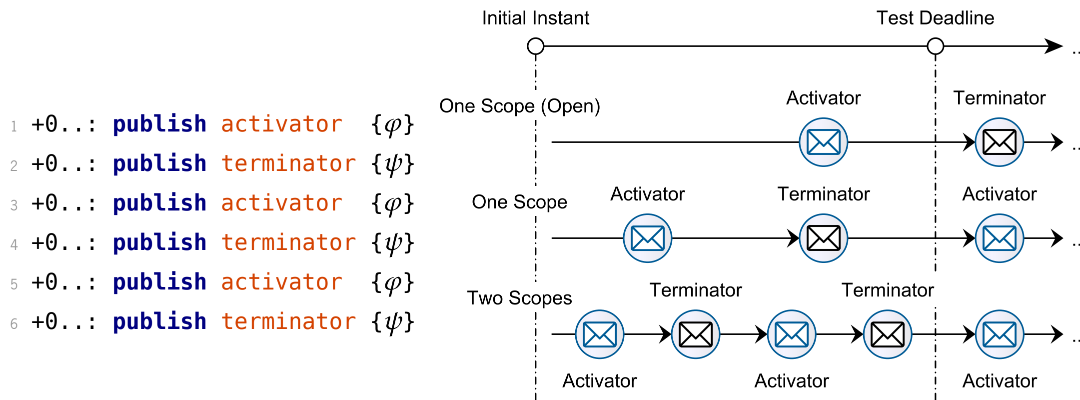
With a limit of up to three scopes for a single test, it stands to reason that:

- tests with a single scope will focus on making the SUT fail on that scope;
- tests with two or more scopes will focus on making the SUT fail on the last scope, and assume other tests of shorter length to focus on the initial scopes.

But note that we can simply focus on schemas for *exactly* three scopes. With careful manipulation of the allowed messages and durations for each scope, there is a *subsumption* relation in place. By schema subsumption, we mean that every fault a schema with a single scope can reveal, can also be captured by a schema with two scopes, and so on.

A schema with a single scope is a schema that publishes an activator message, and may or may not publish a terminator message, but never publishes a second activator message within the test deadline. If we allow for the intervals between activators and terminators to exceed the test deadline, any schema of two or three scopes can also be instantiated with a prefix such that only a single scope can be reproduced within the test deadline. In practice, the longer schemas achieve the same effect as the shorter schema. A schema for three scopes can, thus, subsume schemas of one or two scopes. To illustrate this concept, consider the following simple schema with three scopes. The timelines on the right side show three possible

instances of the schema; in the first two only the first scope starts within the test's deadline, while in the third example two full scopes are included (by choosing lower values for the message timestamps). Also, note that, even if the generated input trace (for three scopes) could be fully reproduced within the deadline, it is possible for the SUT to fail early, within the first or second scopes.



Despite focusing on the relatively low number of three scopes, it should suffice to reveal a large number of faults, according to the small scope hypothesis [10, 90]. Besides, it would be trivial to parameterise this value and repeat the same approach for a higher number of loop iterations. With three scopes, however, we can show some degree of complexity, while striving for readable schemas. Our presentation will focus on the **`after-until`** scope, since it is the most interesting of the four, overall. Other scopes are not reentrant, limiting schemas to a single scope. As a bonus, we can simply reuse the state machines already presented in Chapter 6, for the same scope type; the diagrams had no transitions to $\top$, so we can focus just on the transitions to $\bot$.

At the end of this section, after presenting the general schemas for each property pattern, we discuss an orthogonal problem – how to refine schemas using axioms.

7.4.1 *Absence Properties*

In order to devise schemas for Absence properties, we can take the same state machine that we used for Runtime Verification (Figure 70) as our starting point.

There is only one transition to $\bot$, via a Behaviour message in the Active state. Behaviour messages are produced by the SUT, so our test strategy boils down to finding paths, with two loop unrolls, that lead to Active. Omitting the system's start-up event for readability, we can identify the following paths:

1. Activator → Terminator → Activator → Terminator → Activator;
2. Activator → Terminator → Activator → Timeout → Terminator → Activator;
3. Activator → Timeout → Terminator → Activator → Terminator → Activator;
4. Activator → Timeout → Terminator → Activator → Timeout → Terminator → Activator.

Timeout transitions cannot be translated to schema events in the same way that message-based transitions can. Instead, they are converted into time constraints, affecting the intervals of the following

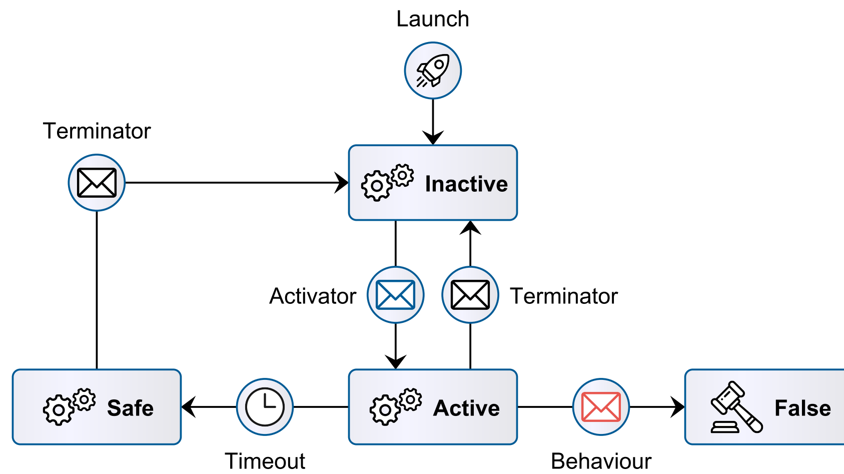


Figure 70: State machine monitoring for the Absence pattern.

messages. In this case, timeouts are always followed by terminator transitions. If a path contains 'Timeout → Terminator', we should produce a schema such that the relative interval pertaining to the terminator has a lower bound greater than the specified timeout, i.e.:

```
1 ...
2 +t..: publish terminator {ψ}
3        forbid activator   {φ}
4 ...
```

Considering the four alternative paths in the state machine, we have the following four schemas.

Schema 1

```
1           forbid activator   {φ}
2  +0..:   publish activator   {φ}
3           forbid terminator {ψ}
4  +0..t:  publish terminator {ψ}
5           forbid activator   {φ}
6  +0..:   publish activator   {φ}
7           forbid terminator {ψ}
8  +0..t:  publish terminator {ψ}
9           forbid activator   {φ}
10 +0..:   publish activator   {φ}
11          forbid terminator {ψ}
```

Schema 2

```
1           forbid activator   {φ}
2  +0..:   publish activator   {φ}
3           forbid terminator {ψ}
4  +0..t:  publish terminator {ψ}
5           forbid activator   {φ}
6  +0..:   publish activator   {φ}
7           forbid terminator {ψ}
8  +t..:   publish terminator {ψ}
9           forbid activator   {φ}
10 +0..:   publish activator   {φ}
11          forbid terminator {ψ}
```

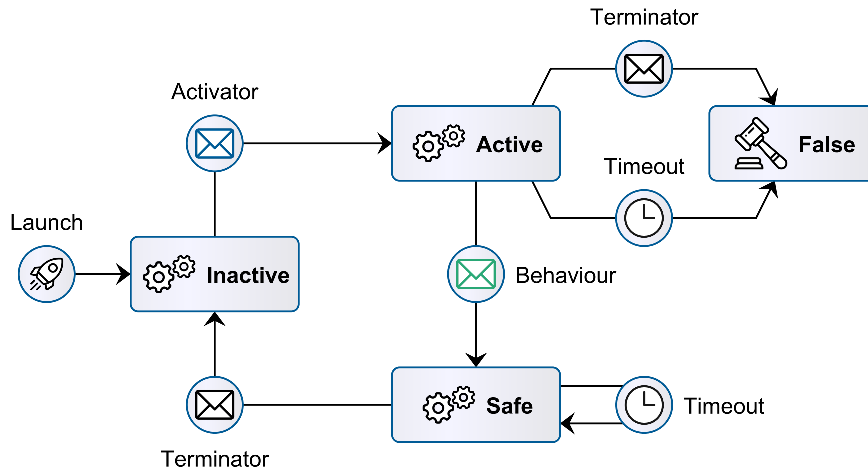


Figure 71: State machine monitoring for the Existence pattern.

Schema 3

```
1          forbid activator  {φ}
2 +0..:    publish activator  {φ}
3          forbid terminator {ψ}
4 +t..:    publish terminator {ψ}
5          forbid activator  {φ}
6 +0..:    publish activator  {φ}
7          forbid terminator {ψ}
8 +0..t:   publish terminator {ψ}
9          forbid activator  {φ}
10 +0..:   publish activator  {φ}
11         forbid terminator {ψ}
```

Schema 4

```
1          forbid activator  {φ}
2 +0..:    publish activator  {φ}
3          forbid terminator {ψ}
4 +t..:    publish terminator {ψ}
5          forbid activator  {φ}
6 +0..:    publish activator  {φ}
7          forbid terminator {ψ}
8 +t..:    publish terminator {ψ}
9          forbid activator  {φ}
10 +0..:   publish activator  {φ}
11         forbid terminator {ψ}
```

The schemas alternate between activators and terminators, switching the permissions on each topic as either is published. Initially, activators are not allowed, until the designated time for the first activator, but other messages can be sent on this topic (e.g., `activator {not φ}`). After the first activator is published the permissions are switched. The activator topic is now allowed any random messages, while the terminator topic must avoid matching the terminator's predicate. In the third block, everything is mirrored – we publish a terminator message and allow anything on the terminator topic, while the activator topic must avoid matching activator messages. This back and forth goes on for as many scopes as we require. What changes is the intervals regarding the publication of terminator messages, in accordance with the specific path we are pursuing.

7.4.2 *Existence Properties*

Runtime monitoring for Existence properties is given by the state machine in Figure 71.

There are two transitions to a verdict of $\perp$, one via a terminator message and another via timeout, when the state machine is in the Active state. More specifically, disregarding loops for the moment, schemas should focus on the following two path suffixes.

1. Activator → Terminator;
2. Activator → Timeout.

We can see how liveness properties (such as Existence) work in contrast with safety properties (such as Absence). Whereas, previously, a behaviour message led to a violation of the property, now such messages are expected from the SUT. The listed path suffixes translate into the following schema suffixes.

Path 1 (via Terminator)

```
1 ...
2 +0..t: publish terminator {ψ}
3        forbid activator   {φ}
```

Path 2 (via Timeout)

```
1 ...
2 +t..:  publish terminator {ψ}
3        forbid activator   {φ}
```

One can argue that the path via Terminator might be unfair for the system; it gives the SUT no time to react to the activator, and almost trivially falsifies the property. But this is one of the strengths of our approach. The system developers might not have been foreseen such a possibility, even though it could happen, in theory.

We must now discuss what happens with the initial scopes leading to the test's focus, the loops in the state machine. This is another stark contrast between safety and liveness properties. While, for Absence properties, the test driver had control over scope loops, for Existence properties the loop only happens if the system publishes a behaviour message. However, we cannot build schemas that include SUT messages; we can only specify inputs.

The best solution for this problem is not to address it at all. We will construct schemas assuming that the system behaves well (i.e., publishes the behaviour message) in the first round of scopes, and that any terminator timestamp is acceptable. Note that these are safe assumptions to make. If the assumptions are confirmed in practice, then we will have successfully prepared the SUT with a certain prefix, to proceed into the most interesting part of the test – the last scope. If, on the other hand, the assumptions turn out to be false, it means that the system will have failed early. This would reveal a problem and falsify the property, aligning perfectly with our end goal.

Having gone through the major considerations for Existence properties, we are now able to look at the schemas for the general case. We have, essentially, the same four alternatives we had for initial scopes in the Absence pattern, times the two possible suffixes we have just discussed, totaling $4 \times 2 = 8$ schemas. We will provide only two as an example, each following one of the two suffixes.

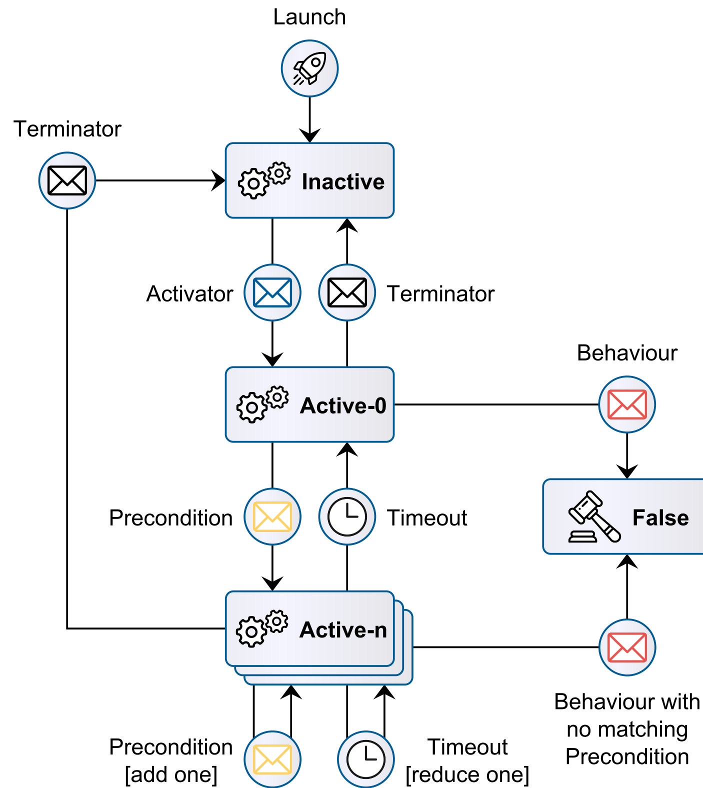


Figure 72: State machine monitoring for the Precedence pattern.

Schema 1 (via Terminator)

```
1          forbid activator   {φ}
2 +0..:   publish activator   {φ}
3          forbid terminator  {ψ}
4 +0..t:  publish terminator  {ψ}
5          forbid activator   {φ}
6 +0..:   publish activator   {φ}
7          forbid terminator  {ψ}
8 +t..:   publish terminator  {ψ}
9          forbid activator   {φ}
10 +0..:  publish activator   {φ}
11         forbid terminator  {ψ}
12 +0..t: publish terminator  {ψ}
13         forbid activator   {φ}
```

Schema 2 (via Timeout)

```
1          forbid activator   {φ}
2 +0..:   publish activator   {φ}
3          forbid terminator  {ψ}
4 +0..t:  publish terminator  {ψ}
5          forbid activator   {φ}
6 +0..:   publish activator   {φ}
7          forbid terminator  {ψ}
8 +t..:   publish terminator  {ψ}
9          forbid activator   {φ}
10 +0..:  publish activator   {φ}
11         forbid terminator  {ψ}
12 +t..:  publish terminator  {ψ}
13         forbid activator   {φ}
```

7.4.3 *Precedence Properties*

Runtime monitoring for Precedence properties, in general, is given by the state machine in Figure 72.

This type of property is similar to the Absence case; they are both safety properties, and they both fail when the SUT publishes an unexpected behaviour message. Given that this is a binary property pattern, we have to deal with additional paths (and internal loops) caused by precondition messages. To avoid a

scenario in which we would have to generate an infinite number of schemas, we collapse all Active-n states into one and unroll loops up to one (mandatory) precondition message. In effect, we consider only two alternative paths, each targeting one of the Behaviour transitions in the diagram.

1. Active-0 $\xrightarrow{Behaviour} \bot$
2. Active-n $\xrightarrow{Behaviour} \bot$

The looping transitions from Active-n to itself, via precondition messages and timeouts, will be delegated to the random messages the input trace generator is allowed to instantiate. That is, our scopes will contain either zero precondition messages, or at least one precondition message (not `forbid`-ing further messages).

Path 1 (zero preconditions)

```
...
+0..: publish activator    {φ}
       forbid precondition {α}
       forbid terminator   {ψ}
...
```

Path 2 (multiple preconditions)

```
...
+0..: publish activator    {φ}
       forbid terminator   {ψ}
+0..: publish precondition {α}
       forbid terminator   {ψ}
...
```

This branching pattern multiplies over the number of scopes, so that we have all possible combinations:

1. 0 Preconditions → 0 Preconditions → 0 Preconditions;
2. >0 Preconditions → 0 Preconditions → 0 Preconditions;
3. 0 Preconditions → >0 Preconditions → 0 Preconditions;
4. 0 Preconditions → 0 Preconditions → >0 Preconditions;
5. >0 Preconditions → >0 Preconditions → 0 Preconditions;
6. >0 Preconditions → 0 Preconditions → >0 Preconditions;
7. 0 Preconditions → >0 Preconditions → >0 Preconditions;
8. >0 Preconditions → >0 Preconditions → >0 Preconditions.

On top of this, there are also the alternatives between short and long initial scopes, when visiting the Active-n state, as we have seen previously. For the sake of illustration, the following is a schema for the case with preconditions in the first and last scopes.

```
1          forbid activator    {φ}
2 +0..:   publish activator    {φ}
3          forbid terminator   {ψ}
4 +0..:   publish precondition {α}     (at least one precondition)
5          forbid terminator   {ψ}
6 +0..t:  publish terminator   {ψ}     (short scope)
7          forbid activator    {φ}
8 +0..:   publish activator    {φ}
9          forbid precondition {α}     (zero preconditions)
10         forbid terminator   {ψ}
11 +0..:  publish terminator   {ψ}     (no timeout available in Active-0)
12         forbid activator    {φ}
13 +0..:  publish activator    {φ}
14         forbid terminator   {ψ}
15 +0..:  publish precondition {α}     (at least one precondition)
16         forbid terminator   {ψ}
```

7.4.4 *Response Properties*

Response properties are monitored with the state machine shown in Figure 73.

This is another liveness pattern, similar to the Existence pattern. From the diagram, we can see that, besides the activator message, at least one stimulus message is required to enable the transition to $\bot$. This rules out any schemas that do not contain any stimulus messages, overall, but note that only the last scope is required to contain at least one stimulus.

In contrast with the previous Precedence pattern, we must **forbid** further stimuli beyond the **publish** event. This is because in Precedence properties we do not control the transition to $\bot$. In Response properties we do, and we want to have a fixed point in the schemas from which no further stimuli are published, so that we can start the timer to exercise either the Terminator transition or the Timeout transition.

To illustrate one of the possibilities, the following schema shows how we would approach a case of zero stimulus messages, followed by two scopes with at least one message.

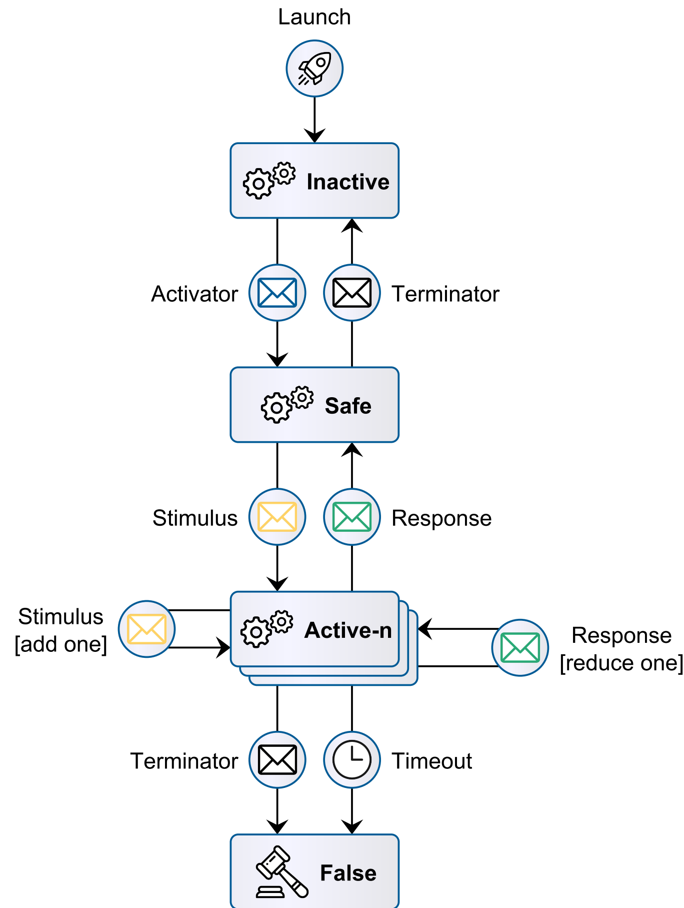


Figure 73: State machine monitoring for the Response pattern.

```
1          forbid activator  {φ}
2  +0..:  publish activator  {φ}
3          forbid stimulus   {α}
4          forbid terminator {ψ}
5  +0..:  publish terminator {ψ}      (zero stimuli)
6          forbid activator  {φ}
7  +0..:  publish activator  {φ}      (stimuli not forbidden)
8          forbid terminator {ψ}
9  +0..:  publish stimulus   {α}      (mandatory stimulus)
10         forbid stimulus   {α}
11         forbid terminator {ψ}
12 +t..:  publish terminator {ψ}      (long scope)
13         forbid activator  {φ}
14 +0..:  publish activator  {φ}
15         forbid terminator {ψ}
16 +0..:  publish stimulus   {α}      (mandatory stimulus)
17         forbid stimulus   {α}
18         forbid terminator {ψ}
19 +0..t: publish terminator {ψ}      (short scope)
20         forbid activator  {φ}
```

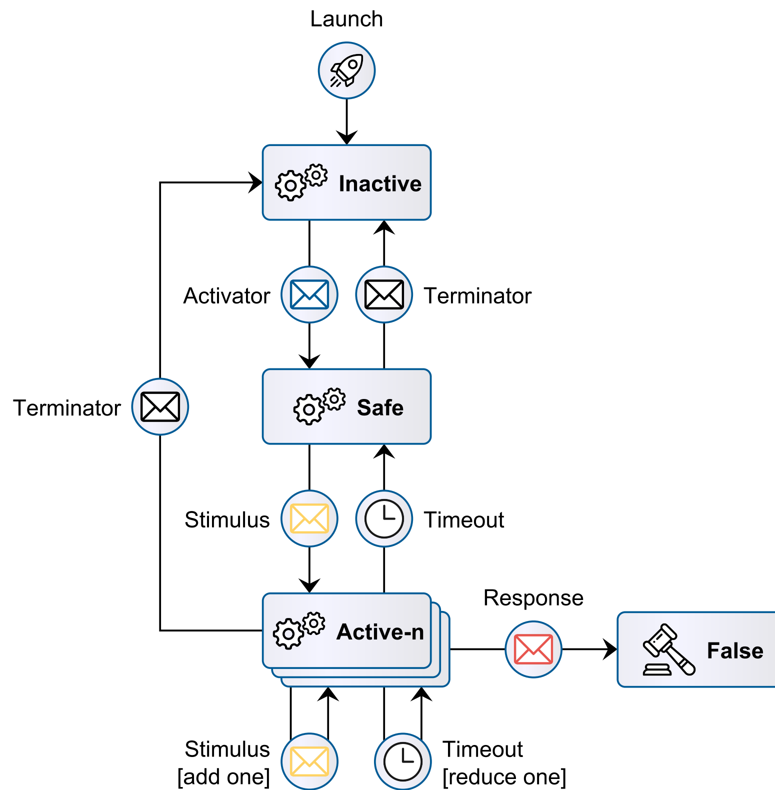


Figure 74: State machine monitoring for the Prevention pattern.

7.4.5 *Prevention Properties*

Runtime monitoring for the Prevention pattern is given by the state machine in Figure 74.

There is not much to discuss at this point. The Prevention pattern is a safety property that has a mixed structure resembling the Response and the Absence patterns. As in the Absence pattern, detecting a violation depends on the system producing a specific message. The messages that cause a violation, however, are sent in response to previous stimuli. Therefore, following suit with the Response pattern, we will be interested in the cases with at least one stimulus in the final scope.

To illustrate one of the possible schemas for this type of property, we will consider an initial scope containing zero stimulus messages, and the last two scopes containing at least one, just like in the Response case, for easier comparison. The only difference is the lack of a terminator at the end.

```
1          forbid activator  {φ}
2 +0..:   publish activator  {φ}
3          forbid stimulus   {α}
4          forbid terminator {ψ}
5 +0..:   publish terminator {ψ}      (zero stimuli)
6          forbid activator  {φ}
7 +0..:   publish activator  {φ}      (stimuli not forbidden)
8          forbid terminator {ψ}
9 +0..:   publish stimulus   {α}      (mandatory stimulus)
10         forbid stimulus   {α}
11         forbid terminator {ψ}
12 +t..:  publish terminator {ψ}      (long scope)
13         forbid activator  {φ}
14 +0..:  publish activator  {φ}
15         forbid terminator {ψ}
16 +0..:  publish stimulus   {α}      (mandatory stimulus)
17         forbid stimulus   {α}
18         forbid terminator {ψ}
```

7.4.6 *Schema Refinement*

We have seen how to build schemas for the various property patterns. However, as mentioned in the overview and workflow of our testing approach, properties do not exist in isolation. Axioms play an important role in the whole process, notably in terms of establishing valid data and valid trace structure. The simple approach to axioms is to let the data generators of the PBT library produce inputs blindly, according to the general schemas, and then discard invalid examples (not in accordance with axioms) via runtime monitoring. And this is what we do in most instances. But, in some cases, we can analyse the axioms statically to refine our trace schemas before the test generation step takes place. This way, we narrow down the examples that the test script is capable of producing to a set that contains (ideally) only the intended inputs or, at least, considerably less invalid examples.

To see how axioms can have an effect on the input trace schemas, let us consider the following example Existence property.

```
1 after /bumper {data = 0} until /bumper {data > 0}:
2     some /controller_cmd within 100 ms
```

Recall that we have to consider eight alternative paths in the Existence state machine, alternating between short and long scopes. With two loop iterations and a short final scope, the minimal input trace for this type of property gives us an obvious (and almost trivial) counterexample, in which the SUT has no time to react and publish its expected message.

```
1 @0ms /bumper {data: 0}     (scope activator)
2 @0ms /bumper {data: 1}     (scope terminator)
3 @0ms /bumper {data: 0}     (scope activator)
4 @0ms /bumper {data: 1}     (scope terminator)
5 @0ms /bumper {data: 0}     (scope activator)
6 @0ms /bumper {data: 1}     (scope terminator)
```

Obvious counterexamples such as this are telltale signs of a problem with the specification. If a rapid sequence of activator and terminator messages is possible in the real system, then the property is not true to begin with (i.e., the specification is incorrect). If such a sequence should not happen, users can augment the specification with axioms to rule out this scenario (i.e., the system is underspecified). Ultimate judgement is left for the user, as is always the case with Property-based Testing. Let us assume an underspecified system. The following pair of axioms forbids scopes starting before the first 100 milliseconds of runtime and imposes a minimum scope duration of 10 milliseconds.

```
1 globally: no /bumper within 100 ms
2 globally: /bumper forbids /bumper within 10 ms
```

The conjunction of these restrictions gives a new shape to the schema. The axioms effectively provide a lower bound for the activator and terminator messages, while the property's timeout provides the upper bounds (where applicable).

```
1             forbid /bumper {data = 0}
2  +100..:   publish /bumper {data = 0}
3             forbid /bumper {data > 0}
4  +10..:    publish /bumper {data > 0}
5             forbid /bumper {data = 0}
6  +10..:    publish /bumper {data = 0}
7             forbid /bumper {data > 0}
8  +10..:    publish /bumper {data > 0}
9             forbid /bumper {data = 0}
10 +10..:    publish /bumper {data = 0}
11            forbid /bumper {data > 0}
12 +10..100: publish /bumper {data > 0}
13            forbid /bumper {data = 0}
```

In the general case, though, axioms are not always easy to integrate. Note that even the refined schema is not foolproof, relative to the second axiom. The input trace generator is free to include additional *spam* `/bumper` messages, so long as they do not clash with the schema's **`forbid`** constraints (i.e., so long and the spam messages do not match the forbidden predicates over message fields). If it does include such messages, the interval lower bound of 10 milliseconds is no longer enough. Consider, for example, the following sequence of messages.

```
1 @100ms /bumper {data: 0}      (activator)
2 @110ms /bumper {data: 0}      (spam)
3 @115ms /bumper {data: 1}      (terminator)
4 ...
```

In this example, the timestamp for the terminator message, which has to be at least 10 milliseconds greater than the activator's, was instantiated as 15 milliseconds. However, in between the two messages, a random message was produced. This is no longer valid input, since it violates the minimal 10 millisecond delay between `/bumper` messages. In short, even with some refinement taking place, we would have to deploy runtime monitors.

Our implementation only takes into account two types of axioms for automatic schema refinement at static time, namely the following Absence properties.

```
1 globally: no topic within 100 ms
2 globally: no topic {predicate}
```

Other types of axioms are currently unhandled; i.e., the generators behave as in the general case, and delegate validation to runtime monitors. Regardless, here we discuss potential approaches to handle each axiom pattern.

Absence Axioms We automatically integrate global-scoped Absence properties that impose *either* temporal constraints or data constraints. The former can be integrated as we have just shown, by increasing the lower bound of the first relevant interval. The latter is automatically built into the data generators, which is part of the reason why we do not handle expressions containing arithmetic or quantifiers. Mixing both types of constraints requires more complex analyses. We would have to determine, at compile time, whether there is overlap between property and axiom predicates, to decide whether the intervals should be affected. This path is set aside in our current approach, delegated to runtime monitoring. We also set aside scopes other than **`globally`**, which require similarly complex analyses.

Existence Axioms This type of axiom requires that the input trace must contain a certain type of message. Depending on the scope, the specified message may be placed anywhere in the trace or it might be confined to a specified scope. A best-effort approach (not yet implemented) would likely generate the input trace in two passes: the first following the general schema, and the second identifying the appropriate scopes to generate (and insert) the axiom-related messages.

Precedence Axioms A best-effort approach for this type of axiom would have to ensure that precondition messages occur within a certain time window (to the past) of behaviour messages. This could be done again in, at least, two passes. The first pass generates messages as per the general schema. The second pass evaluates whether the generated messages satisfy the axiom and, if not, whether some messages could be moved closer together (or reordered) to satisfy time constraints. If there are no available

messages, new precondition messages would have to be generated for each behaviour in the preliminary trace, or behaviours would have to be removed.

Response Axioms As in the case of Existence axioms, these require the insertion of messages into the preliminary trace, but only if the trace already contains the stimulus message. In that regard, these might be easier to satisfy, since, if there are no stimuli, nothing needs to be done. In their simplest form, '`globally: /a causes /b within T ms`', these properties can be integrated in the schemas by setting an upper limit of `T` on some intervals (or possibly decreasing an upper bound already in place).

Prevention Axioms These axioms work in contrast with the previous. In the presence of stimuli, messages have to be removed, or set apart from each other for a minimum delay, which might end up interfering with the remaining constraints. The simplest forms of these axioms, though, such as '`globally: /a forbids /b within T ms`', are relatively easy to integrate, as we have shown, even though they are not currently implemented. They just increase the lower bounds of some intervals to `T` in the schema.

7.5 Implementation

In this section we will go over the implementation details behind our approach, starting with the test generation plug-in, moving into the Hypothesis data generators for input traces, and down to the test driver function. As we have previously mentioned, our implementation is not capable of handling everything the specification language is able to express. Arithmetic and quantified expressions are not supported for open subscribed topics, for instance. We will identify current limitations at appropriate points of the discussion.

7.5.1 *Test Generation Plug-in*

The plug-in itself does not warrant much discussion. Even though it performs a number of computations, most of its steps are relatively straightforward. The overall process of the plug-in is summarised in Listing 7.3.

```
1 def make_tests(configuration):
2     open_subscribed_topics = get_open_subscribed_topics(configuration)
3     all_properties = configuration.hpl_properties
4     testable_properties = get_testable_properties(all_properties,
5                                                   open_subscribed_topics)
6     axioms = get_axioms(all_properties, open_subscribed_topics)
7     rendered_templates = make_test_templates(open_subscribed_topics,
8                                              testable_properties, axioms)
9     write_test_files(rendered_templates)
```

Listing 7.3: Top-level function to generate test scripts from annotated Configurations.

One of its first steps is to identify open subscribed topics, given a HAROS Configuration, as shown in Listing 7.4. Determining this set of topics will, in turn, determine which properties are axioms and which properties are testable, according to our previous definitions.

```
1 def get_open_subscribed_topics(configuration):
2     ignored = plugin_settings.get("ignored", ())
3     topics = {} # topic -> TypeToken
4     for topic in configuration.topics.enabled:
5         if topic.subscribers and not topic.publishers:
6             if topic.unresolved or topic.type == "?":
7                 log_warning("Skipping unresolved topic {} ({}).".format(
8                     topic.rosname.full, configuration.name))
9             elif topic.rosname.full in ignored:
10                log_warning("Skipping ignored topic {} ({}).".format(
11                    topic.rosname.full, configuration.name))
12            else:
13                topics[topic.rosname.full] = get_type_token(topic.type)
14    return topics
```

Listing 7.4: Function to identify open subscribed topics in a Configuration.

The most complex step of the overall process is the `make_test_templates` function, which is broken down into a number of steps itself, as shown in Listing 7.5. For each testable property, it starts by building data generators for Hypothesis (lines 5-7), so that input traces can be instantiated, according to a number of schemas. The schemas are determined based on the type of property and its state machine, as we have seen in the previous section. Then, the plug-in builds other necessary pieces of information for the general test script template, and gathers everything under a `data` dictionary (line 11). For instance, it collects the names of runtime monitors to deploy, `roslaunch` and `rosrun` commands to deploy the SUT, necessary Python package dependencies, and general test settings, such as the test deadline. Lastly, the collected information is passed on to the Jinja template engine, to produce the source code for an executable ROS PBT node (line 12).

```
1 def make_test_templates(open_topics, properties, axioms):
2     tests = []
3     for p in properties:
4         try:
5             strategies = build_input_trace_strategies(p)
6             py_input_trace = render_template(
7                 "input_trace_strategies.python.jinja", strategies)
8         except SchemaError:
9             log_warning("Cannot produce a test for:\n'{}'\n\n{}".format(p, e))
10            continue
11        data = {} # py_input_trace, monitors, ROS launch commands, etc.
12        py_test_case = render_template("test_case.python.jinja", data)
13        tests.append(py_test_case)
14    return tests
```

Listing 7.5: Function to build test scripts using schemas and code templates.

The remaining parts of the plug-in are mostly related to the implementation of the `build_input_trace_-strategies` function. This makes for some quite extensive blocks of source code that we will omit, for

readability purposes. The plug-in starts by performing a number of (very simple) static analysis checks over the properties and axioms. First, with static analysis, we are able to discard obvious contradictions on the predicates over message fields (e.g., '`x > 0 and x < 0`') and still report them at *compile-time*, rather than failing at runtime. Second, we are able to narrow down some of the possible values for Hypothesis strategies, e.g., given an integer '`x > 0`', we are able to use Hypothesis' `integers(min_value=1)`, as we will see. Once this is done, the plug-in determines appropriate schemas for the given property and generates all the necessary Hypothesis strategies. This includes strategies for mandatory events (e.g., activator messages), strategies for spam messages over the various phases of an input trace, and also a strategy for the input trace itself.

With the details about the plug-in itself out of the way, next we will delve into the details of Hypothesis strategies for ROS messages and input traces.

7.5.2 *Input Trace Data Generators*

Basic Messages and Data Types

We start off by creating strategies for all basic data types a ROS message can contain, such as 8-bit, 16-bit, 32-bit and 64-bit integers (signed and unsigned), 32-bit and 64-bit floating-point numbers, strings and fixed-length or variable-length arrays of such values. Listing 7.6 illustrates this with the resulting strategy for 8-bit signed integers.

```
1  from hypothesis.strategies import integers
2
3  INT8_MIN_VALUE = -(2 ** 7)
4  INT8_MAX_VALUE = (2 ** 7) - 1
5
6  def ros_int8(min_value=INT8_MIN_VALUE, max_value=INT8_MAX_VALUE):
7      if (min_value < INT8_MIN_VALUE or min_value > INT8_MAX_VALUE
8              or max_value < INT8_MIN_VALUE or max_value > INT8_MAX_VALUE
9              or min_value > max_value):
10         raise ValueError('values out of bounds: {}, {}'.format(min_value, max_value))
11     return integers(min_value=max(min_value, INT8_MIN_VALUE),
12                     max_value=min(max_value, INT8_MAX_VALUE))
```

Listing 7.6: Hypothesis strategy for 8-bit signed integers.

On top of these strategies for basic data types, we are able to build default strategies for all message types the test requires. Since ROS messages are custom data types, that Hypothesis knows nothing about, we will use the previously presented `@composite` strategy. To build default strategies for all message types, we follow three steps.

1. Identify all open subscribed topics in the architectural model.
2. Create *type tokens* for the message types of all open subscriptions and their transitive dependencies (`get_type_token` function, line 13 in Listing 7.4). A type token is a data structure that lists all the fields the message type is supposed to contain, and their respective types.

3. Create a strategy for each message type, using the default strategies for the data types of each field. Listing 7.7 shows an example for the `geometry_msgs/Twist` message, a standard message to express velocities in free space, broken into their linear and angular parts.

```
@composite
def ros_geometry_msgs_Twist(draw):
    msg = geometry_msgs.Twist()
    msg.linear = draw(ros_geometry_msgs_Vector3())
    msg.angular = draw(ros_geometry_msgs_Vector3())
    return msg

@composite
def ros_geometry_msgs_Vector3(draw):
    msg = geometry_msgs.Vector3()
    msg.x = draw(ros_float64())
    msg.y = draw(ros_float64())
    msg.z = draw(ros_float64())
    return msg
```

Listing 7.7: Strategy for `geometry_msgs/Twist` messages.

Messages With Constraints

The following step is to generate message strategies with constraints. This is where we must take into consideration any provided axioms, as well as the specific property pattern we are testing. We must ensure that specific messages such as activators and preconditions have their own strategies, complying both with user-defined predicates and with axioms regarding message contents. Even random messages must have several strategies, because, depending on where the message is to be positioned in the trace, in some cases we have to avoid matching activators, while in other cases we avoid matching terminators or stimulus.

As an example of message strategies conforming with predicates, consider the following Response property.

```
globally: /laser {not data > 50} causes /stop_cmd within 200 ms
```

From the previous state machine and schema analysis, we know that the resulting trace must contain a `/laser` message such that '`not data > 50`'. When possible, our approach builds these predicates directly into the message generators, via static analysis, so that we only generate valid values from the start. In part, this is why we impose so many restrictions over testable properties, such as avoiding arithmetic. Naturally, some predicates are beyond the capabilities of static analysis (or beyond reasonable computation effort), in which case we must resort to random values and validation at runtime. Listing 7.8 shows the generated strategy for the example property.

At this point, we should also make a short note regarding axioms (and message predicates, in general). Powerful as they are, specifying axioms comes at the cost of degraded test performance. Axioms are, in practice, additional constraints that must hold for the generated inputs. While we are able to optimise for

```
1 @composite
2 def cms1_std_msgs_Int8(draw):
3     msg = std_msgs.Int8()
4     msg.data = draw(ros_int8(max_value=50))
5     return msg
```

Listing 7.8: Strategy for `std_msgs/Int8` messages such that `data` $\leq$ 50.

some cases, via static analysis of the properties themselves, for most expressible properties we can only evaluate the generated inputs after generation. Thus, if the axioms are hard to satisfy, we are at risk of having to generate inputs repeatedly until one is finally valid.

Our current implementation only optimises for global Absence properties. Axioms such as '`globally: no /laser {data < 0 or data > 7}`' can be automatically built into the data generators, so that only satisfying inputs are generated. All other axioms are evaluated dynamically before running the test example. If an axiom is violated, the test example is discarded and marked as invalid (using the `assume` function from Hypothesis).

Input Traces

The last step is to group messages together under a single trace, and this is where we must take schemas and timing constraints into account. How the strategy for an input trace looks depends largely on the selected schema. The exact amount of (optional) messages that the generated traces contain, or the delays between messages, are some examples of variables that we leave for Hypothesis to explore. The order in which spam messages are allocated throughout the trace is also variable, to an extent. For illustrative purposes, we show in Listing 7.9 a (simplified) generated trace strategy for the previous Response property, based on the following schema. Note the absence of scope activators and terminators, a consequence of the `globally` keyword.

```
1          forbid /laser {not data > 50}
2 +0..:   publish /laser {not data > 50}     (single stimulus)
3          forbid /laser {not data > 50}
```

As we can see, the trace strategy starts by generating the timestamp for the first mandatory event. Then, it generates random messages (lines 8-16) for irrelevant topics (`/bumper` and `/wheel`). By default, the trace generator aims for relatively short traces. Whenever random messages are allowed in a trace, it adds up to three random messages per topic, possibly none. This decision is based on the *small scope hypothesis* [10, 90], which states that testing a program for all inputs within a relatively small scope is enough to make a high number of faults manifest.

In lines 17-21, the messages are sorted randomly (`shuffle`) and then, for each message, we draw a random relative delay of zero or more milliseconds (`relative_delay`), up to the value of the initial timestamp. The relative delays set the waiting time between one message and the next. Once we have all messages, they can be easily converted into absolute timestamps. A delay of zero means that the respective message is sent at the same time as the previous message in the trace.

```python
@composite
def traces(draw):
    trace = Trace()
    r = draw(randoms())
    delay_strategy = integers(min_value=0, TEST_DEADLINE)
    first_delay = draw(delay_strategy)
    # Spam Messages #######################################
    random_msgs = []
    elems = ros_std_msgs_Int8()
    msgs = draw(lists(elems, min_size=0, max_size=3))
    for msg in msgs:
        random_msgs.append((msg, "/bumper"))
    elems = ros_std_msgs_Int8()
    msgs = draw(lists(elems, min_size=0, max_size=3))
    for msg in msgs:
        random_msgs.append((msg, "/wheel"))
    r.shuffle(random_msgs)
    delay_strategy = integers(min_value=0, max_value=first_delay+1)
    for msg in random_msgs:
        relative_delay = draw(delay_strategy)
        trace.append(msg, relative_delay)
    assume(trace.duration < first_delay)
    # Stimulus Message ####################################
    msg = (draw(cms1_std_msgs_Int8()), "/laser")
    trace.append(msg, first_delay)
    return trace
```

Listing 7.9: Example strategy to generate message traces.

This first segment of code (lines 7-22) handles the initial phase of the schema, from the SUT's launch until the stimulus event; there is no scope activator for the `globally` scope. The second segment of the presented function (lines 23-25) generates the mandatory `/laser` message such that `data` $\leq$ 50, for the stimulus event. Again, due to being a `globally` scope, there are no terminators, nor further scopes besides the one.

Shrinking

In our current approach, as we could see from the various strategy examples shown so far, shrinking is left for Hypothesis to handle. We use the default roster of strategies provided by Hypothesis whenever possible (`integers`, `lists`, etc.), which implies the default shrinking behaviour. By default, Hypothesis will shrink traces along three axes:

- reduce the number of messages in the trace (lists of random messages tending towards zero elements);
- reduce the time it takes to replay the trace (delays between messages tending towards the minimum value);
- shrink the contents of each individual message (shrinking data fields as per `@composite` rules).

As we have seen, it could try any of the three approaches, or a combination of the above. This implementation makes no effort to override Hypothesis' shrinking heuristics. In part, this is because it is

not clear what the best shrinking behaviour should be. It is not clear whether we should prioritise traces with a smaller number of messages, traces with shorter delays between messages, or traces with smaller values in the various message fields. The first and third options are better for debugging purposes, to help users understand reported failures. The second option (and the first, to a lesser extent) helps in reducing the total time needed to replay an input trace.

Following one of the alternatives above the others might actually increase the time it takes to test a property. Suppose a complex counterexample is found. The shrinking phase begins, opting, for instance, to reduce the number of messages in the trace. It would try to remove all optional messages, starting with one at a time, then two, and so on, until it exhausts all possibilities. If the actual counterexample requires all messages, but can be shrunk in the individual message fields, all the investment in prioritisation would have gone to waste. A mixed approach might be the most performant, on average.

We opt to leave this problem (which is likely more complex than it appears to be) for future work, and trust in the years of research and experience behind Hypothesis.

7.5.3 *Hypothesis Test Driver*

Given a trace generator that satisifes all the previous requirements, the test driver itself is relatively straightforward. Some low-level details, such as deploying the SUT between tests will be left out, as they are mostly boilerplate code. Listing 7.10 shows a simplified version of the code for the main test function.

```
1  @given(trace=traces())
2  def test_properties(self, trace):
3      self.eval_trace(trace)
4      self.setup_test()
5      self.setup_sut()
6      try:
7          for msg, topic, delay in trace:
8              rospy.sleep(delay)
9              for monitor_verdict in self.verdicts:
10                 assert monitor_verdict
11             self.publishers[topic].publish(msg)
12         # after replaying all messages in the input trace:
13         rate = rospy.Rate(100) # hz
14         while not self.safe_to_terminate:
15             for monitor_verdict in self.verdicts:
16                 assert monitor_verdict
17             rate.sleep()
18     finally:
19         self.teardown_sut()
```

Listing 7.10: Main Hypothesis test function.

The input for the test function is a concrete input trace. It includes messages up front for however many scopes we will be testing within a single test case. In other words, traces are *not* generated online; we do trace generation beforehand. The main reason for this is a concern for non-functional performance. Sometimes, traces might include very small delays between messages, and possibly a considerable number

of simultaneous messages. Some ROS messages are quite large and complex, and it takes time to produce them. Considering also that we have to uphold a number of constraints, if we opted for online generation, it would be likely that we would miss deadlines, and our messages would no longer be published at the intended timestamps.

The first thing we do within the test function is trace validation. The `eval_trace` function takes in an input trace to evaluate, dynamically, whether the given trace satisfies all axioms. Evaluating a trace is faster than waiting for its replay and monitoring, which contributes to an overall shorter testing period. This is another point in favour of offline generation of traces.

As we move on, the first and last few lines handle the test infrastructure. The `setup_test` function initialises internal variables, publishers, subscribers, timers and runtime monitors. The `setup_sut` function launches the ROS system under test, using functions directly from the ROS tools. It then blocks until all required nodes and topics are detected. Monitors with an implicit activator are notified of the system's launch at this point. The `teardown_sut` is the dual function that shuts down the system under test. Recall that the `test_properties` function is executed multiple times by Hypothesis. As such, we can see that the system is launched and shut down with every example. This takes a significant toll on execution time, as we will see in Chapter 8, but it is the only way to ensure that the system is started from a clean state with every test.

Regarding the core of the function's body, we can see a simple loop, using `rospy.sleep()` to ensure that we meet the desired publishing frequency. The simplified code omits optimisations that can take place at this point, to skip the sleep instruction entirely if the delay between messages is zero. Within this loop, we also check whether any monitor has detected a property violation so far, and abort the test early if so.

Finally, after the trace is fully replayed, there is a new check to ensure that no property has been violated. It is repeated periodically, until it is safe to deem the test case complete. By default, it runs until a given deadline. In some cases, we can terminate early – for instance, if the input trace ends with a terminator message.

7.6 Summary

This chapter presents our Property-based Testing approach based on Runtime Verification for ROS systems. The main goal of this work is to leverage the benefits of automatic testing techniques, such as Specification-based Testing and Property-based Testing, in this case, while lowering the specification effort. The specification language we presented in Chapter 4 is used as an entry point for the automatic generation of test scripts specifically designed for a given system architecture. We realise the proposed approach as a HAROS plug-in, since the framework is already capable of automatic extraction of the required architectural models. One of the advantages of this approach is that users get feedback as to whether a concrete system implementation complies with a specification, as well as a counterexample in the case that it does not. In addition, the use of Property-based Testing tools ensures that there is an effort to shrink counterexamples. Rather than presenting the first encountered counterexample for a property, this approach strives to provide the user a minimal counterexample; an invaluable aid in the debugging effort.

8

CASE STUDIES AND EVALUATION

In the previous chapters we have proposed the two major components of our general approach to ROS-specific analyses of software systems. The first component is an architecture model extraction technique that captures both static artefacts, such as packages and files, and the runtime entities that compose the system's ROS environment, such as nodes and topics. Relying on this system model, and on a domain-specific property specification language, the second component is able to generate automated property-based tests that explore a vast range of the input space and are able to detect non-compliant behaviour. Both components have their respective implementation integrated within the HAROS analysis framework, and, in this chapter we evaluate their performance on two distinct case studies. For the whole evaluation process, we use a virtual machine running 64-bit Linux Xubuntu 18.04.1 and ROS Melodic. This virtual machine was given 2 cores of a Intel(R) Core(TM) i7 2630QM (2.00 GHz) and a base memory of 2 GB DDR3 RAM (665 MHz).

8.1 Case Studies

8.1.1 *TurtleBot2*

TurtleBot is one of the most iconic ROS robots. It is a low-cost, personal robot kit with open-source software that was designed in collaboration with the original makers of ROS, Willow Garage. Its main purpose is to provide an entry-level platform for roboticists to build applications with. It is a platform to learn, explore and research with ROS, and, thus, it is more of an academic case study.

TurtleBot2[1] (Figure 75) is the second generation of the TurtleBot, following within the ROS Enhancement Proposal (REP) 119 specification[2]. This REP – a document that describes a ROS design issue, or provides general guidelines or information to the ROS community – outlines compatibility definitions for TurtleBot compatible platforms, such as hardware and software requirements. This second generation personal robot is equipped with a Kobuki robot base (essentially a repurposed robot vacuum cleaner), and various off-the-shelf consumer electronics, such as a standard dual-core netbook, a RGBD sensor and a gyroscope,

1 http://www.turtlebot.com/turtlebot2/
2 https://www.ros.org/reps/rep-0119.html

Figure 75: The TurtleBot2 robot.

in order to keep costs low. The TurtleBot is a specific product, but it is an Open Hardware Design released under the FreeBSD Documentation License.

TurtleBot2 provides for an interesting case study, since its source code has been relatively well maintained, and it includes some uncommon features, such as conditional topic subscriptions, that our tools should be able to handle. Besides, it has been implemented with extensibility and customisation in mind, featuring tens of different launch files and configurations out-of-the-box. Another compelling factor is the richness of online tutorials, videos, sample code and other documentation that is easily available. The existence of these artifacts makes the evaluation of our tools easier, since we know what is the expected architecture and behaviour of the system without relying too much on reading the robot's source code. For the purposes of our evaluation, we will focus on two configurations of the TurtleBot2.

The first configuration is called the Random Walker Controller configuration. This configuration launches all the core components of the Kobuki mobile base, and then adds a Random Walker Controller node, that, as implied by its name, moves the robot around in randomly chosen directions. In addition, two other key nodes of the TurtleBot2 design are launched, the Safety Controller node, responsible for reacting to dangerous sensor readings, and the Multiplexer node, that allows only the highest priority velocity commands to be processed. A diagram of this architecture is shown in Figure 76.

The second configuration is called the AMCL[3] Navigation configuration. While the core components are the same as in the previous configuration, the Random Walker Controller is replaced by a proper navigation stack. Given a previously known map, an initial location estimate and a destination point, the robot is capable of identifying its location and move towards its goal, avoiding both static and dynamic obstacles along the way. Besides sending destination goals, users are also allowed to integrate teleoperation in the

3 AMCL stands for Adaptive (or KLD-sampling) Monte Carlo localisation, the robot localisation algorithm used in this configuration.

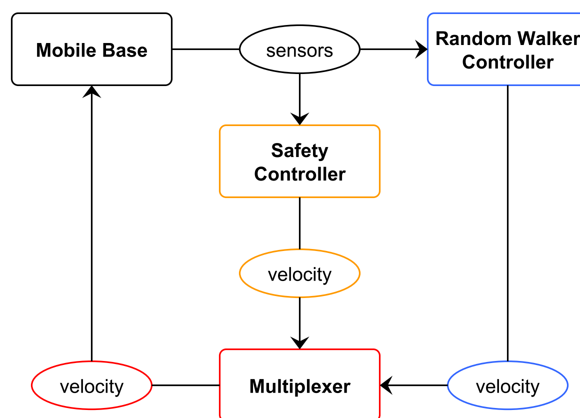


Figure 76: The Random Walker Controller configuration.

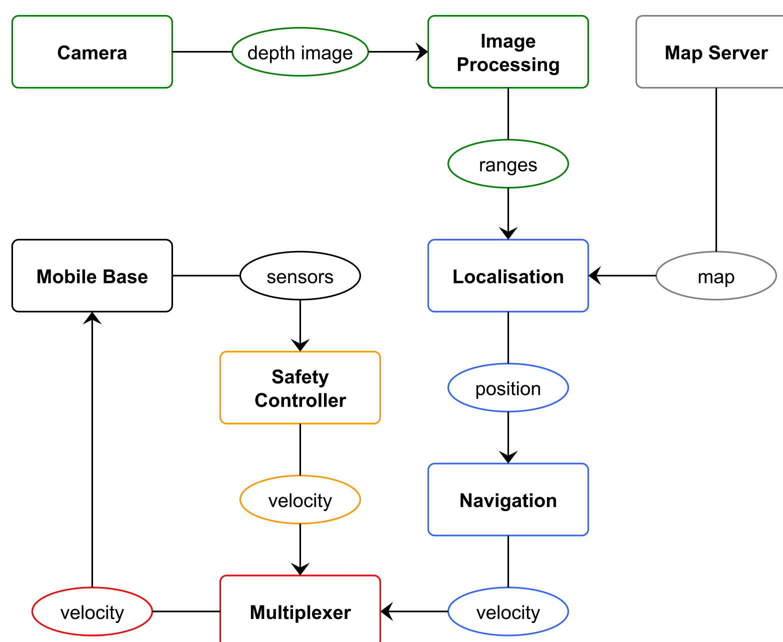


Figure 77: The AMCL Navigation configuration.

configuration. For the purposes of navigation, additional nodes to process camera signals are required. Figure 77 shows a diagram of the architecture.

8.1.2 *AgRob V16*

AgRob V16[4] is a modular robotic platform for hillside agriculture, adapted to sloping and uneven terrains (see Fig. 78). It is designed for monitoring, precision spraying, pruning and selective harvesting, particularly in steep slope vineyards. Among other innovative features, this robotic platform is equipped with an advanced navigation system that allows it to estimate its location even when GPS is not fully available. Its modularity makes it able to integrate various types of payload, both in the form of sensors and actuators. The main driver behind this project is to provide commercial solutions for hillside vineyards capable of

4 http://agrob.inesctec.pt/

Figure 78: The AgRob V16 agriculture robot monitoring a slope vineyard.

autonomously executing operations of Monitoring and Logistics. Such features will help winemakers make important decisions, increasing wine quality and production.

There are a few appealing factors in AgRob V16 that make it an interesting case study for us.

- It is a real industrial robot that heavily relies on ROS, a sharp contrast with TurtleBot2, which is more of an academic case study.
- It is a more complex system in every sense, from the hardware components to the dimension of its software.
- A large part of its software comes from integrated third-party packages that provide mapping and localisation, among other features, while TurtleBot2 was mostly implemented from the ground up in earlier versions of ROS.
- The coding styles of both robots are vastly different, despite both being programmed in C++.
- TurtleBot2 was made with open source in mind, and it has received contributions and reviews from the community over the years, while AgRob V16 is mostly a closed source system (at the time of writing).
- The developers and maintainers of AgRob V16 belong to a robotics research group from our research institution, allowing for personal meetings, e-mail exchanges and, overall, an easier understanding of the system's inner workings and intended behaviour.

For the purposes of our evaluation, we consider two very similar configurations. The first is the Basic configuration, which allows a user to alternate between teleoperation and autonomous navigation using a joystick. The second is the Path Planning configuration, which is mostly the same as the Basic configuration, except for adding a specialised path planning node that is adapted to sloping terrain. The diagram of Figure 79 shows the overall architecture for both configurations. The Path Planning component is shown in dashed lines.

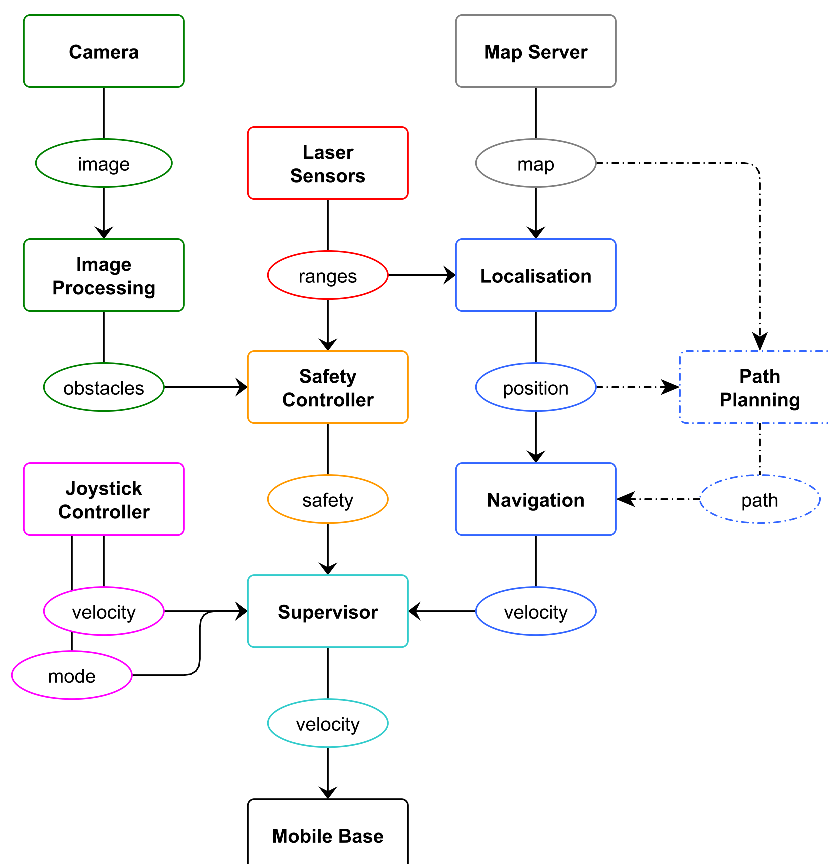


Figure 79: The AgRob V16 Basic and Path Planning configurations, the latter in dashed.

Both configurations share a relatively large set of nodes, including mobile base controllers, laser sensors, cameras, map servers, and, most importantly, custom Joystick Controller, Supervisor and Safety Controller nodes. The Safety Controller node, in a similar fashion to the TurtleBot2 case, is responsible for reading the various sensors (lasers, IMU, cameras) and determining the current state of the robot (e.g., *safe*, *caution* or *danger*). The Supervisor node is (in concept) a glorified variant of TurtleBot2's Multiplexer. It receives velocity commands from multiple sources (e.g., joystick, autonomous navigation) and selects the appropriate command to pass down to the mobile base, according to the state reported by the Safety Controller node. In addition, it also ensures that all velocity commands are within dynamic maximum and minimum velocity limits (that also depend on the safety state). The Joystick Controller converts joystick commands into velocity commands and changes the robot's mode of operation (via a message sent to the Supervisor) depending on whether certain buttons are pressed. Lastly, the Path Planning component implements an extension of the A* algorithm, specifically designed to navigate in steep slope vineyards, that is aware of the robot's distance to vine trees. The algorithm behind this component has been published in [152].

8.2 Evaluation of the Model Extractor

The process of evaluating a model extraction (or reverse engineering) tool is not standard, and is very dependent on the application area and main goal of the model. In our case, we have to evaluate how accurate are the architectural models of a ROS system extracted with HAROS. These models include several types of complex entities, both from the file system layer (e.g., Packages and Files) as well as the system's runtime (i.e., the ROS Computation Graph). Our objective is to assess how many entities of the actual system are captured by the model, and to what level of detail.

Intuitively, we know that capturing all system entities but providing no details about them (such as types, full names, etc.) is not the same thing as capturing just a few, but with all attributes completely determined. Since the extraction of file system entities is relatively straightforward, requiring little more than a traversal of directories, we decided to leave these out of the evaluation, in order to avoid bias. Thus, we focus this evaluation on the ROS Computation Graph, i.e., in HAROS's ability to correctly parse and interpret launch files as well as C++ source code.

In quantitative approaches, the selected metrics vary, depending on the application area and main purpose of the model. However, the metrics of Precision, Recall and the F-score (in particular, the F_1-score) are among the most common for performance measurement. We use these metrics, in conjunction with the number of required extraction hints, as our evaluation goals. In this section we define each of the metrics and explain our measurement process.

8.2.1 *Core Performance Metrics*

In binary classification problems, such as testing whether a person has contracted a disease, whether a piece of software is faulty, or, more generally, whether a datum is correctly labelled, observations fall under the following four categories.

TRUE POSITIVE (TP) The test reports positive for a truly positive case; e.g., a faulty piece of software is deemed faulty.

TRUE NEGATIVE (TN) The test reports negative for a truly negative case; e.g., a correct piece of software is deemed correct.

FALSE POSITIVE (FP) The test wrongly reports positive for a negative case; e.g., a correct piece of software is deemed faulty.

FALSE NEGATIVE (FN) The test wrongly reports negative for a positive case; e.g., a faulty piece of software is deemed correct.

It is clear that we want to maximise true positives and true negatives, while minimising false positives and false negatives. In the example of detecting software bugs, false positives are an annoyance – they set the maintainer on a search for a problem that does not exist, lowering trust on the detection mechanism.

False negatives, on the other hand, are a more serious problem – bugs slip by undetected, bolstering trust at first, but possibly manifesting under unfavourable circumstances at a later point.

These four metrics are often combined, so as to achieve a single metric that reflects perceived performance. The naïve (and most intuitive) approach is to calculate the Accuracy (*ACC*) metric [158]. It is the ratio of correct predictions over the total number of observations, and is defined as:

$$ACC = \frac{TP + TN}{TP + TN + FP + FN}$$

The main problem with this metric is that it favours trivial implementations for asymmetrical datasets. For instance, when detecting software bugs in safety-critical systems, a detector can assume that most of the software is correct, and thus report every observation as correct, i.e., everything is a negative. Naturally, this yields a low number of false negatives, and thus a high accuracy metric, but the cost of a single false negative is high. So, optimising for Accuracy is not always the best choice, and the metrics of Precision and Recall tend to be favoured instead.

Precision (*PRE*) can be informally defined as the ratio of *how many of the positive reports are actually positive cases*, and Recall (*REC*) can be informally defined as the ratio of *how many of the actual positives are reported as positive*. They are defined as follows.

$$PRE = \frac{TP}{TP + FP} \qquad REC = \frac{TP}{TP + FN}$$

Depending on the application, one metric might be preferable over the other. For instance, in e-mail spam detection, optimising for precision is more desirable because the cost of a false positive is high. A false positive means that a regular e-mail is incorrectly flagged as spam, which might lead to a user losing important e-mails. In software bug detection, we witness the converse situation, in which recall is the optimal metric. The cost of a false negative, i.e., letting a bug go undetected, is higher than the cost of reporting correct software as faulty. Note that, for relatively complex classification problems, these metrics tend to be inversely proportional. To achieve maximum precision, the safest course of action is to refrain from making any judgement – if there are no reported positives, there are no false positive reports. However, to achieve maximum recall, one wants to label everything as positive, to ensure that no actual positives are disregarded.

The F-score (or F-measure) is the harmonic mean between Precision and Recall [43, 55, 159]. Its main purpose is to achieve a single performance metric, in the range of $[0, 1]$, that strikes a balance between the two metrics, yet is more useful for uneven data distributions than Accuracy; the F-score does not focus on true negatives, and works better than Accuracy if the costs of false positives and false negatives are not equal. The most common variant of this metric is the F_1-score (F_1), a function that weighs precision and recall evenly. Other common variants are the F_2-score and the $F_{0.5}$-score, which emphasize recall over precision and precision over recall, respectively. The definition of F_1 is as follows.

$$F_1 = 2 \times \frac{PRE \times REC}{PRE + REC}$$

8.2.2 *Performance Metrics for Complex Entities*

In our case, we are not dealing with a binary classification problem, so the notions of *positive* and *negative* become blurred. We are extracting entities that, when compared with a ground truth, can sit anywhere in the spectrum of completely different to completely correct, with various levels of partial correctness in between. How to address such complex comparisons and convert them into Precision and Recall values is not straightforward. We found the modern approaches used in the Named Entity Recognition and Classification (NERC) sub-domain of Natural Language Processing [44, 154] to be among the best fits for this purpose, as we explain here.

In NERC applications, the intent is to process a source text, written in natural languages such as English, extract relevant *named entities* and classify them (e.g., people, organisations, locations). Entities are often characterised by their string boundaries (start and end index of the substring in the main text) and the type of the entity. Our problem shares a few traits with NERC, but with added complexity. In our case, the source text is source code, split among various files and written in different programming languages. Our entities do not have string boundaries, but they do have traceability information, i.e., a location in the source text that marks the start of the entity's occurrence. Lastly, besides a type, our entities have a number of other attributes, such as the ROS name. Contrary to NERC, though, our extraction process does not always assign a value to each entity attribute; we allow data fields to be labelled as *unknown*, resulting in more opportunities for partial correctness.

A number of ranking and evaluation systems emerged over the years to provide a fair score to an evaluated system at the entity level (as opposed to evaluating at the token level), when comparing its output to that of a human linguist (i.e., comparing automated output to a ground truth). For the evaluation of our model extractor, we adapted the process used in the 2013 International Workshop on Semantic Evaluation (SemEval-2013) [154], which is in turn based on other processes, such as the one used in the Conference on Message Understanding (MUC) [44]. This process is illustrated in the diagram of Figure 80.

In the SemEval-2013 evaluation process, there is not a single performance metric, but rather four sets of performance metrics (with Precision, Recall and F_1 for each), associated with different measurement modes.

STRICT The string boundaries and type of the compared entities must be equal.

EXACT The string boundaries of the compared entities must be equal, but not their types.

PARTIAL The string boundaries of the compared entities must overlap, if they are not the same.

TYPE The types of the entities must match, even if the string boundaries are wrong.

For each of the previous measurement modes, extracted entities are matched against the ground truth entities (called the *golden standard* in this context) by string boundaries. Each entity pair is counted under one of five categories, depending on the measurement mode.

CORRECT (COR) The extracted entity and the golden entity are considered equal.

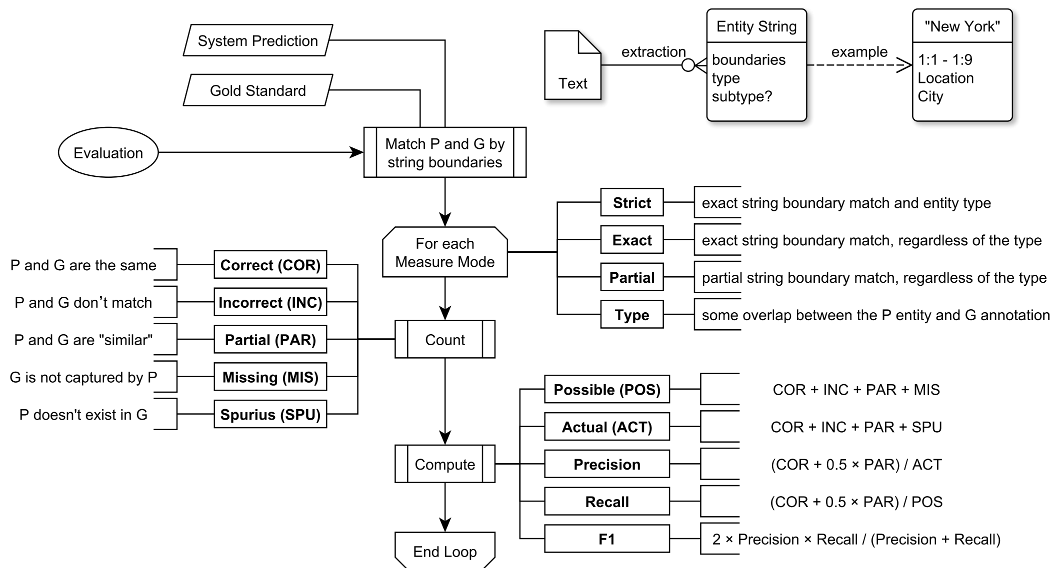


Figure 80: The evaluation system used in SemEval-2013.

INCORRECT (INC) The extracted entity and the golden entity are different.

PARTIAL (PAR) The extracted entity and the golden entity match for some attributes.

MISSING (MIS) The golden entity does not have a match in the set of extracted entities.

SPURIOUS (SPU) The extracted entity does not have a match in the golden standard.

These measurements are then converted into Precision and Recall according to the following formulae. The F_1-score is calculated as per the traditional formula.

$$PRE = \frac{COR + 0.5 \times PAR}{COR + INC + PAR + SPU} \qquad REC = \frac{COR + 0.5 \times PAR}{COR + INC + PAR + MIS}$$

8.2.3 *Methodology*

Our evaluation process is similar to the SemEval-2013, but adapted to our specific context. The first step is to annotate each selected Configuration of the case studies with a ground truth model. This model is built by hand, by reading and interpreting the code as the ROS environment would, and then using runtime inspection tools for confirmation. To write the ground truth model, out of convenience and to strive for consistency, we use the same YAML syntax that we previously defined in Chapter 3 for the model extraction hints. As is the case with the extraction hints, this model contains the top-level entities that are spawned by launch files (Nodes and Parameters), with the Links produced via C++ or Python code as attributes of the Nodes. However, for evaluation purposes, we consider Links as independent entities because they are also resources (first-class citizens) of the ROS computation graph. We treat them as if they were declared at top-level, with an additional attribute defining their Node owner. Listing 8.1 shows a small example of this syntax.

```yaml
nodes:
  "/mobile_base":
    node_type: kobuki_node/KobukiNodelet
    args: "load kobuki_node/KobukiNodelet mobile_base_nodelet_manager"
    remaps:
      "/mobile_base/odom": "/odom"
      "/mobile_base/joint_states": "/joint_states"
    traceability:
      {package: kobuki_node, file: launch/minimal.launch, line: 8, column: 3}
    publishers:
      - topic: "/odom"
        original_name: "/mobile_base/odom"
        msg_type: nav_msgs/Odometry
        queue_size: 50
        traceability:
          {package: kobuki_node, file: src/library/odometry.cpp, line: 76, column: 20}
parameters:
    ...
```

Listing 8.1: Excerpt of a ground truth model, illustrating a node with a publisher.

Note that, for a typical ROS system, much of its functionality comes from nodes and libraries that are not built from scratch; reuse of freely distributed components made by other members of the community is greatly encouraged. A large set of these components are what makes today the standard distributions of ROS, and, in practice, these would not even be installed from source, but rather from pre-packaged binaries. As mentioned in Chapter 5, when possible, HAROS resorts to a database of pre-parsed Nodes for such standard components. To avoid this massive source of bias for the evaluation, we decided to drop out of both the extracted models and the ground truth all the Links that such nodes create. Instead, we evaluate just the extractor's ability to detect that such Nodes are present in launch files, and evaluate the Node's specific attributes – e.g., in which namespace is the node launched, and with which name remappings. In short, we evaluate our performance only for the entities that are truly unique to each Configuration.

Once the laborious process of building a ground truth is completed, we submit the extracted model and the corresponding ground truth to an evaluation script that compares both and calculates Precision, Recall and the F_1-score. Again, out of convenience, the evaluation script is actually implemented as a HAROS plug-in[5]; this way it has direct access to the models, and is able to use HAROS's reporting mechanisms to display the results. As to how the actual performance measurement is done, we illustrate the process in the diagram of Figure 81.

The first step is to match the extracted entities with their counterparts from the ground truth. In this case, it is not as simple as comparing string boundaries, and must take partial information into account as well, such as unresolved names that stem from dynamic values. Thus, we try to mimic what a human would likely do. When looking at the two models, especially if looking at their visual representation in the HAROS visualiser, we assume that a human would try to match the models according to the following criteria.

1. Match by Resource type; i.e., match Nodes with Nodes, Publishers with Publishers, etc.

5 https://github.com/git-afsantos/haros-plugin-model-ged

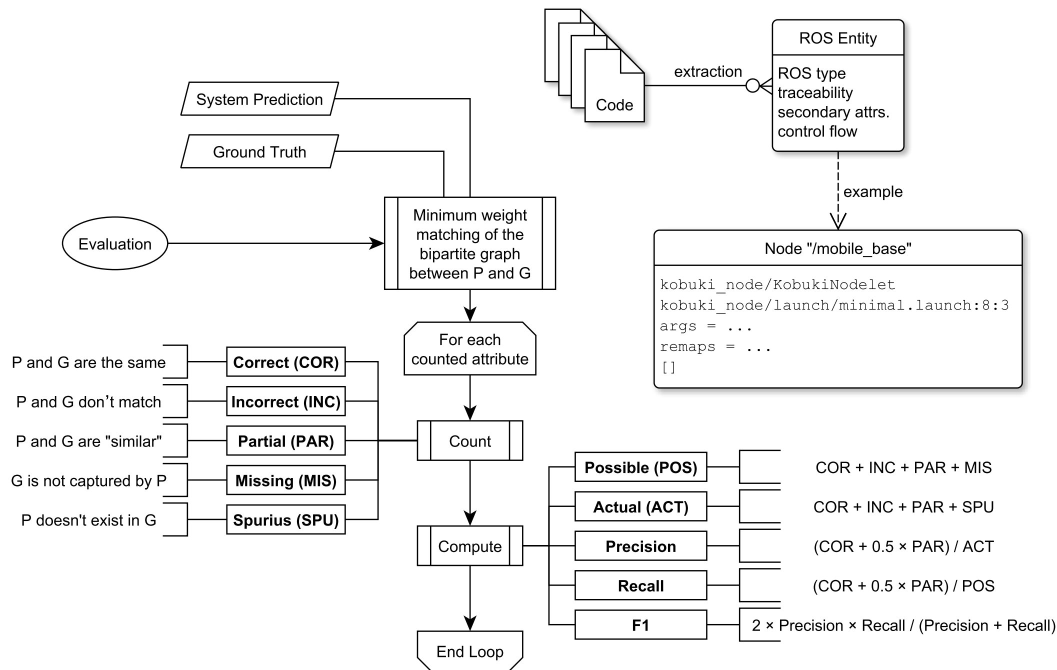


Figure 81: The evaluation algorithm we use to assess the model extractor's performance.

2. Nodes and Parameters are independent resources, but, for Links, match only resources of the same type that belong to the same Node; i.e., do not match publishers from node `/robot` with publishers from node `/laser`.
3. Within the resources of the same type, match by ROS name; i.e., `/robot` and `/robot` are a match, `/?` and `/robot` *might be* a match, and `/laser` and `/robot` are not a match.
4. If the match by ROS name is unclear, try to match by ROS type; i.e., a publisher of type `geometry_msgs/Twist` is likely not a match to a publisher of type `sensor_msgs/LaserScan`.
5. If the match by ROS name and type is still unclear, try to match the traceability information; i.e., the source code location at which the entity occurs.

To implement the resource matching algorithm, we first build a full bipartite graph between resources of the same type. Each edge of the graph is assigned a *weight*, based on the aforementioned heuristics (lesser is better). Then, we use Karp's algorithm [95] to produce a *minimum weight matching* of the graph. A matching is a selection of edges from the graph such that no node appears twice, and all nodes are included if possible. A minimum weight matching is a matching such that the combined weight of the edges is the lowest possible, when compared to other possible matchings. In the case that one side of the bipartite graph includes more nodes than the other, there will be leftover nodes, not included in the matching (meaning that they have no suitable pair).

In Figure 82 we can see an example matching. On one side of the graph we have nodes **A**, **B** and **C**, while on the other side we have nodes **1** and **2**. A greedy and naïve strategy would likely match nodes **A-1**, since the weight of this edge is 0, but that would leave no choices other than **B-2** or **C-2**, which would amount to a total weight of 10. If we select edges **A-2** and **B-1** we have a minimum weight of 6, with node

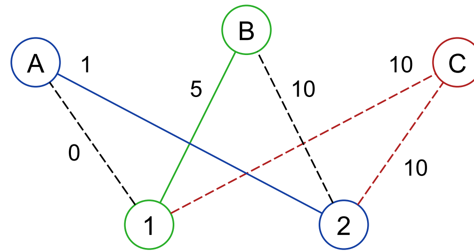


Figure 82: Example of a minimum weight matching.

C as a leftover. Depending on whether **C** is a part of the extracted model or the ground truth, we consider all its attributes as *spurious* or *missing* (respectively).

Once the entities are paired with the ground truth counterparts, we evaluate performance separately for each attribute of a resource pair. This way, we can assess, for instance, HAROS's performance regarding the extraction of ROS names, or its performance regarding queue sizes for publishers and subscribers. Another benefit of this approach, is that we can aggregate the metrics at higher levels of abstraction, to have an idea of the overall performance for all Node attributes, or for all attributes of all entities in general (achieving a single F_1-score relative to a Configuration).

As is the case with NERC, each attribute pair is deemed Correct, Incorrect, Partial, Missing or Spurious. For Missing entities (respectively, Spurious), all their attributes are automatically counted as Missing (respectively, Spurious). We try to refrain from counting attributes as Partial, except for ROS names. With ROS names, having partial data is a feature of our proposed approach, so we first convert the unresolved name to a regular expression pattern. Only if the regular expression is compatible with the fully resolved name of the ground truth do we count it as Partial. Otherwise, it is counted as Incorrect. With this setting, there is little distinction between the measurement modes of the SemEval-2013 approach, so we conducted our evaluation with this single mixed measurement mode.

For each of the case studies, and for each of the studied configurations, we present tables that show: *(i)* how many attributes were measured, and under which categories (COR, INC, PAR, MIS, SPU); *(ii)* Precision, Recall and F_1-score according to the measurements of *(i)*; and *(iii)* the measurements for each entity type, as well as the overall figures. For readability purposes, we provide only tables for all attributes combined. In addition, we detail how many attributes had to be specified via extraction hints afterwards, in order to achieve perfect Precision and Recall.

8.2.4 *Results for the TurtleBot2 Case Study*

Random Walker Controller Configuration

For the Random Walker Controller configuration, we were able to achieve an F_1-score of 93.5%, as shown in the following table.

All Attributes	*COR*	*INC*	*PAR*	*MIS*	*SPU*	*Precision*	*Recall*	*F1-score*
Node	36	0	0	0	0	1.000	1.000	1.000
Parameter	185	0	0	0	0	1.000	1.000	1.000
Topic Publisher	165	3	0	28	0	0.982	0.846	0.909
Topic Subscriber	123	2	1	6	0	0.980	0.943	0.961
Service Client	0	0	0	0	0	1.000	1.000	1.000
Service Server	0	0	0	5	0	1.000	0.000	0.000
Parameter Setter	0	0	0	12	0	1.000	0.000	0.000
Parameter Getter	104	4	0	18	0	0.963	0.832	0.893
Overall	613	9	1	69	0	0.985	0.890	0.935

In this case, we missed the extraction of Links that come from composite primitives, provided by packages other than the base ROS client libraries (not yet included in the parsing capabilities of HAROS). For instance, we missed Links (full entities, with multiple attributes) created by the `tf`, `diagnostics` and `dynamic_reconfigure` packages. The `dynamic_reconfigure` case, in particular, is a recurrent issue that lowers the metrics considerably, as the use of this package entails at the very least 5 different Links, with a single line of C++ code. We also missed subscriptions of the multiplexer node, which are dynamically created, in a loop, after reading subscriber data from a parameter YAML file. Lastly, we missed the default values (attributes) for three parameters that are defined in constants of a C++ implementation file (not a header) that is not part of the analysed source code (i.e., we only consider the binaries of that package). All issues are related to C++ parsing; launch file parsing for this configuration was perfect.

In terms of extraction hints for this configuration, we ended up fixing 6 entities and creating 11 new entities, as shown in the following table.

Extraction Hints	*Fixed Entities*	*Created Entities*	*YAML Lines*
Topic Publisher	1	4	41
Topic Subscriber	1	1	12
Service Server	0	1	8
Parameter Setter	0	2	16
Parameter Getter	4	3	32
Overall	6	11	109

Note that the creation of a single entity via hints fixes multiple missed attributes in the previous table. For instance, we can see from this table that we have created 4 Publishers. Each Publisher corresponds to 7 attributes, as we can see in the following example (`topic`, `msg_type`, `queue_size`, `package`, `file`, `line`, `column`). Thus, 4 entities with 7 attributes each, yields a total of 28 missed attributes in the first table.

```yaml
publishers:
    - topic: "/tf"
      create: true
      msg_type: tf2_msgs/TFMessage
      queue_size: 100
      traceability:
        package: kobuki_node
        file: src/library/odometry.cpp
        line: 31
        column: 1
```

AMCL Navigation Configuration

For the AMCL Navigation configuration, we were able to achieve an F_1-score of 94.5%, as shown in the following table.

All Attributes	*COR*	*INC*	*PAR*	*MIS*	*SPU*	*Precision*	*Recall*	*F1-score*
Node	126	0	0	0	0	1.000	1.000	1.000
Parameter	1208	2	0	0	0	0.998	0.998	0.998
Topic Publisher	158	3	0	42	0	0.981	0.782	0.871
Topic Subscriber	117	2	1	18	0	0.979	0.858	0.914
Service Client	0	0	0	0	0	1.000	1.000	1.000
Service Server	0	0	0	10	0	1.000	0.000	0.000
Parameter Setter	0	0	0	42	0	1.000	0.000	0.000
Parameter Getter	134	4	0	72	0	0.971	0.641	0.772
Overall	1743	11	1	184	0	0.993	0.901	0.945

A large part of this configuration is shared with the previous one, so the previous issues are also present. In addition, there are a few more uses of `dynamic_reconfigure`, increasing the number of missed Links, there are more subscribers to set up in the multiplexer node, and launch file parsing is no longer perfect. We now have a parameter, defined in a launch file, whose value is the text output of a command-line application that is supposed to run in place – `xacro`, a XML macro utility that is used in ROS to read XML descriptions of physical robot models. HAROS does not support this feature of running arbitrary commands during launch file interpretation, so it is not capable of determining the parameter's value.

In terms of extraction hints for this configuration, we ended up fixing 7 entities and creating 30 new entities, as shown in the following table.

Extraction Hints	*Fixed Entities*	*Created Entities*	*YAML Lines*
Parameter	1	0	3
Topic Publisher	1	6	61
Topic Subscriber	1	3	30
Service Server	0	2	16
Parameter Setter	0	7	56
Parameter Getter	4	12	104
Overall	7	30	270

8.2.5 *Results for the AgRob V16 Case Study*

Basic Configuration

For the AgRob V16 Basic configuration, we were able to achieve an F_1-score of 91.8%, as shown in the following table.

All Attributes	*COR*	*INC*	*PAR*	*MIS*	*SPU*	*Precision*	*Recall*	*F1-score*
Node	132	0	0	0	0	1.000	1.000	1.000
Parameter	503	2	0	0	0	0.996	0.996	0.996
Topic Publisher	54	1	1	14	0	0.973	0.779	0.865
Topic Subscriber	102	0	0	0	0	1.000	1.000	1.000
Service Client	5	0	0	0	0	1.000	1.000	1.000
Service Server	0	0	0	5	0	1.000	0.000	0.000
Parameter Setter	0	0	0	66	0	1.000	0.000	0.000
Parameter Getter	84	0	0	66	0	1.000	0.560	0.718
Overall	880	3	1	151	0	0.996	0.851	0.918

In a similar fashion to the TurtleBot2 case study, many of the extraction issues are actually missing Links due to the use of the `dynamic_reconfigure` package. There is also a single parameter whose value we able unable to determine, due to its value being the result of a dynamic call to the `xacro` utility. Lastly – and a novelty in this case study – we have a ROS name for a publisher that we are no able to resolve fully. The string is the result of a string concatenation, and one of the substrings is a dynamic value, set by a ROS parameter. HAROS is unable to resolve this, in its current version. The purpose of this dynamic value in code is to promote code reuse with different robot versions (by changing the dynamic value that is set via ROS parameters).

In terms of extraction hints for this configuration, we ended up fixing 2 entities and creating 25 new entities, as shown in the following table.

Extraction Hints	*Fixed Entities*	*Created Entities*	*YAML Lines*
Parameter	1	0	3
Topic Publisher	1	2	22
Service Server	0	1	8
Parameter Setter	0	11	88
Parameter Getter	0	11	88
Overall	2	25	209

Path Planning Configuration

For the Path Planning configuration, we were able to achieve an F_1-score of 87.4%, as shown in the following table.

All Attributes	*COR*	*INC*	*PAR*	*MIS*	*SPU*	*Precision*	*Recall*	*F1-score*
Node	138	0	0	0	0	1.000	1.000	1.000
Parameter	583	2	0	0	0	0.997	0.997	0.997
Topic Publisher	166	1	1	42	0	0.991	0.793	0.881
Topic Subscriber	156	0	0	0	0	1.000	1.000	1.000
Service Client	14	3	3	0	0	0.775	0.775	0.775
Service Server	0	0	0	15	0	1.000	0.000	0.000
Parameter Setter	0	0	0	156	0	1.000	0.000	0.000
Parameter Getter	339	9	0	156	0	0.974	0.673	0.796
Overall	1396	15	4	369	0	0.988	0.784	0.874

This configuration shares much of its code with the Basic configuration, so many of the issues are the same. The use of `dynamic_reconfigure`, in particular, is even more prominent, resulting in 61 missed Links (mostly parameter getters and setters). This configuration also has a few more ROS names that HAROS was unable to resolve fully, due to the namespace part coming from member variables of a class. For the moment, we assume that member variables can change anywhere in the code (especially if more threads are involved), and treat the value as dynamic. There are also a number of parameter getters whose default values are string concatenations – once again, beyond the current capabilities of the tool.

In terms of extraction hints for this configuration, we ended up fixing 14 entities and creating 61 new entities, as shown in the following table.

Extraction Hints	*Fixed Entities*	*Created Entities*	*YAML Lines*
Parameter	1	0	3
Topic Publisher	1	6	62
Service Client	3	0	6
Service Server	0	3	24
Parameter Setter	0	26	208
Parameter Getter	9	26	226
Overall	14	61	529

8.3 Evaluation of the Online Runtime Monitors

The domain on Runtime Verification is vast, and there are many available tools to perform this task, with different levels of complexity of the underlying formalism, and different optimisation goals, as discussed in Chapter 6. Despite the existing range of tools, we implemented our own approach to monitor the properties defined by our property specification language (Chapter 4). The main reason for this decision is that most of the existing tools lack at least one of the required features for our setting. Some tools are based on LTL (or similar logics), and have no access to first-order predicates nor real-time operators. Others focus on just one of these aspects, while lacking the other. However, there is one tool in particular that stands out among the rest.

The MonPoly tool [18, 20] was designed to monitor a safety fragment of MFOTL formulae; the same logic formalism that makes the foundation of our own approach. With its capability to monitor a vast array of formulae and it being a general-purpose tool, rather than a ROS-based one, we expect its performance to be worse than our custom implementation's. On the other hand, it is implemented in OCaml, an industrial-strength programming language with its own compiler, that is leagues faster and more efficient than any equivalent (interpreted) Python script. Thus, in this section, we describe a prototype implementation to integrate MonPoly with ROS, and compare it to our own monitoring approach.

8.3.1 *Prototype* MonPoly *Implementation*

Our proposed approach is based on asynchronous online monitors, for minimal intrusiveness, requiring no instrumentation of the target system. Fortunately, MonPoly works with these conditions out-of-the-box. One major obstacle for its direct adoption – besides the fact that it is ROS-agnostic – is that it is implemented in OCaml. ROS does not provide any official bindings for the OCaml language. Thus, we face a choice between unofficial bindings[6] [7], or using the tool externally. For the purposes of this evaluation, we opt for the latter.

Simply put, our solution is to reuse much of the same Python code that makes the implementation of our proposed monitors – namely, the entry point and the interface with ROS – replacing only the monitors themselves with a child process that runs MonPoly. To avoid complex, platform-dependent features, such as asynchronous input/output, for each MonPoly monitor we spawn an additional native thread that continuously reads and parses the output of the tool. The ROS Python client automatically spawns a thread per subscribed topic, so, as soon as a new message is received and processed, the subscriber thread can convert the message into the MonPoly event format and pass it along to the child process. An internal timer of the main script sends periodic empty events to MonPoly, so that it can update its data structures and detect time-based violations earlier. Figure 83 illustrates this architecture.

Granted, there are some drawbacks to this implementation that might affect performance, in particular the time it takes to register a violation. This is due to the communication between processes being slower

6 https://github.com/hlinhn/ocaml-ros
7 https://github.com/AestheticIntegration/imandra-ros

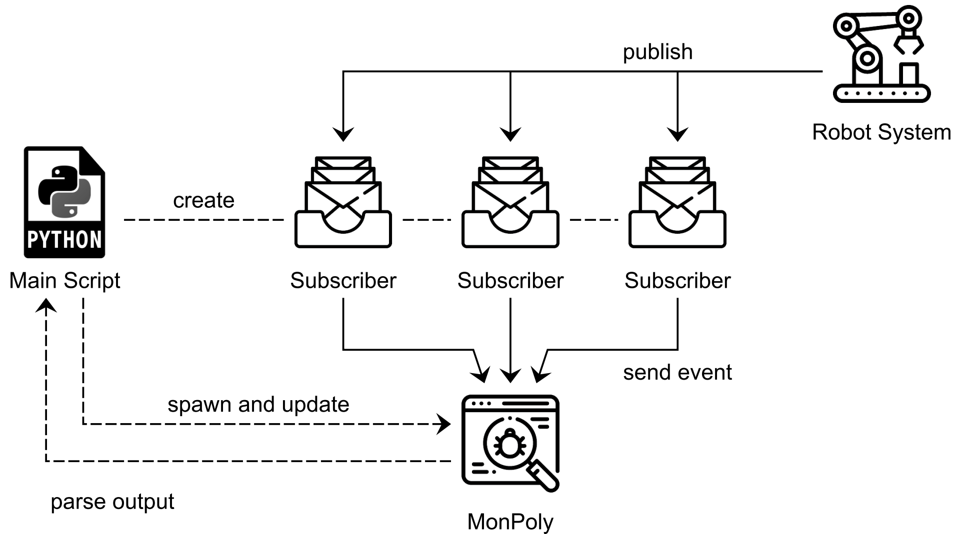


Figure 83: Architecture of the MonPoly integration with a ROS system.

than shared memory within a single process, and due to each MonPoly monitor spawning an additional thread to continuously parse the tool's output. On the other hand, by reusing a significant part of our own Python code base to abstract away the minutiae of managing ROS communications, we can put both monitor implementations on the same grounds and compare just the time it takes for each monitor to report a violation for the same message trace.

8.3.2 *Methodology*

In order to give a fair chance at both monitor implementations, we encoded a small catalogue of properties covering different patterns. We use our language semantics, defined in Chapter 4, to convert properties into MFOTL formulae, that we can then encode into MonPoly's specification language. As mentioned, MonPoly can only monitor a safety fragment of MFOTL, specifically formulae of the form $\Box \forall \overline{x} : \varphi$. In some cases, this requires formula rewriting or simplification to fit this format.

In practice, MonPoly negates the input formula and searches the input trace for a match; i.e., a counterexample. This implies that, unlike our implementation, MonPoly can only report violations. It is unable to report early success for (bounded) liveness properties, e.g., '`globally: some /bumper within 500 ms`'. Thus, the focus of this evaluation is in terms of time taken for each monitor to report the same violation.

For each property, we devise both valid and counterexample traces to be automatically reproduced, once the monitoring systems are ready. The purpose of the counterexample traces is obvious – they allow us to check whether both monitors are able to detect the violation, and measure the time it takes to do so. The valid traces, on the other hand, serve us by reinforcing our confidence in the encoding of the properties and in the correctness of both implementations. Also, each implementation is tested independently, so that one does not interfere with another's performance.

Below we list our catalogue of properties, where /p is always the activator topic to enter a scope, /q is the corresponding terminator topic, /a is the topic of stimulus (or precondition) messages, and /b is the topic of all behaviour messages. All topics are of the type std_msgs/Int8, i.e., 8-bit signed integers.

```
1 globally: /a as A causes /b {data = @A.data} within 100 ms
2 globally: /a causes /b within 100 ms
3 globally: /b as B requires /a {data = @B.data} within 100 ms
4 globally: /b requires /a within 100 ms
5 after /p {data > 0}: /b as B requires /a {data = @B.data} within 100 ms
6 after /p {data > 0}: /b requires /a within 100 ms
7 after /p until /q: no /b within 100 ms
```

Most of the properties with global scopes are relatively straightforward to encode in MonPoly's specification language, in part due to how much we can simplify them. Properties such as the last one, using the '`after /p until /q`' scope, are much harder to encode. The semantics of our language do not require /q to happen, but MonPoly does not accept unbounded future-time operators. This requires that either *(i)* the formula can be rewritten in such a way that /q does not appear in any unbounded future operators, or *(ii)* we have to bound such future operators with an ad-hoc, sufficiently large timeout. In this case, we are able to avoid /q in unbounded future operators, and, hence, never fall into the second scenario for any property.

For the sake of example, consider the third property in the catalogue. After translation to MFOTL, negation of the formula and simplification, we end up with the following MonPoly encoding:

```
1 b__data(x) AND (PAST_ALWAYS[0, 100) (NOT (EXISTS y. a__data(y) AND x = y)))
```

Recall that MonPoly wraps the input formula with implicit operators, $\Box\forall\overline{x} : \varphi$, but actually monitors the negated formula, to find a counterexample, i.e., $\Diamond\exists\overline{x} : \neg\varphi$. We negated the formula beforehand, so our encoding actually represents the $\neg\varphi$ part of the overall formula. The events b__data and a__data refer to the data field in the /b and /a messages, respectively. Variable x in the encoding is captured by the implicit quantifier. As such, the negated, simplified formula in MFOTL is:

$$\Diamond\exists x : (b(x) \land \blacklozenge_{[0,100)}(\neg\exists y : (a(y) \land x = y)))$$

This is a relatively simple case, in terms of encoding. Other properties, such as the last one, turn out to be much more verbose. The simplified MFOTL formula, negated using MonPoly's embedded formula negation capabilities, is the following:

$$\Diamond(((\neg(((\nexists x : p(x)) \lor \exists y : q(y)) \lor ((\neg\bullet_{[0,\infty)}((\neg\blacklozenge_{[0,\infty]}\exists y : p(y)) \\ \lor((\nexists z : p(z))\mathcal{S}_{[0,\infty)}\exists y : q(y)))) \land\bullet_{[0,\infty)}\top))) \land\Diamond_{[0,100)}\exists y : b(y)) \\ \land\neg((\nexists z : b(z))\mathcal{U}_{[0,100)}\exists y : q(y)))$$

And the MonPoly encoding for this formula is:

```
((NOT
    (((NOT EXISTS x. p(x)) OR EXISTS y. q(y))
        OR
        ((NOT PREVIOUS[0,*)
                ((NOT ONCE[0,*) EXISTS y. p(y))
                OR ((NOT EXISTS z. p(z)) SINCE[0,*) EXISTS y. q(y))))
            AND PREVIOUS[0,*) 0 = 0)))
    AND EVENTUALLY[0,100) EXISTS y. b(y))
AND NOT ((NOT EXISTS z. b(z)) UNTIL[0,100) EXISTS y. q(y))
```

Without a doubt, this sort of property shows how much more convenient it is to use a high-level specification language, such as the one we propose. The property itself is rather simple. There are no constraints over message fields, nor cross-references between messages. The use of the `after-until` scope, however, makes the encoding process much more complex.

As for the example traces, when exercising counterexamples we try to violate properties in various ways. Take property 1, for instance; a message on topic `/a` should lead to a subsequent message on topic `/b`, containing the same 8-bit signed integer, no longer than 100 milliseconds afterwards. We can violate this property in (at least) three different ways, even though the ultimate cause is always the lack of an appropriate response for the stimulus.

1. The correct response is published, but too late, e.g., 200 milliseconds afterwards.
2. A response message is published within the allowed period, but its data differs from the stimulus message.
3. No response message is published at all.

This categorisation of counterexamples is reflected in the trace schemas we presented in Chapter 7, for the purposes of testing. Using such schemas as our basis, we have a systematic method to build a relatively small number of traces that are all different in some meaningful aspect. For this specific property, the first counterexample category, for instance, is given by the following schema.

```
+0..:   publish /a as A
         forbid /a
         forbid /b {data = @A.data}
+100..: publish /b {data = @A.data}
```

As we can see, the last interval, `+100..` guarantees that the response, ‘`/b {data = @A.data}`’, despite being correct, is published too late, past the 100 millisecond limit. For the last property, however, we had to impose a limit on the number of counterexample traces. Since the property’s scope is reentrant, the number of counterexamples is infinite, in theory. We settled on six counterexample traces, covering both cases with single scopes and with multiple scopes.

Table 3: Comparison of our custom monitor implementation versus MonPoly.

Property #	Counterexample #	Custom (ms)	MonPoly (ms)	Difference (ms)	Gain (%)
1	1	152	220	68	30.91
1	2	156	1081	925	85.57
1	3	156	1023	867	84.75
2	1	153	221	68	30.77
2	2	152	1012	860	84.98
3	1	2	17	15	88.24
3	2	53	70	17	24.29
3	3	202	210	8	3.81
4	1	2	13	11	84.62
4	2	203	216	13	6.02
5	1	54	63	9	14.29
5	2	105	114	9	7.89
5	3	253	290	37	12.76
6	1	53	71	18	25.35
6	2	258	268	10	3.73
7	1	53	120	67	55.83
7	2	53	119	66	55.46
7	3	55	112	57	50.89
7	4	10 512	10 547	35	0.33
7	5	52	109	57	52.29
7	6	55	121	66	54.55

8.3.3 *Comparison of Implementations*

For each monitor implementation, we ran each example trace 10 times, and then took the averages of the results. This is to rule out the possibility of performance spikes due to garbage collection, process scheduling, or other interferences. Overall, we observed no false positives, and both implementations found all counterexamples. For the seven listed properties, we have a total of 21 counterexamples. Every trace has a delay of about one second at the end, before signaling the end of the trace. Table 3 shows how each implementation fares in terms of average time to report a violation for each counterexample.

As we can see in Table 3, there is a consistent difference ranging from about 10 milliseconds to nearly 70 milliseconds between the two implementations, with our custom implementation being the fastest. There are a few cases, such as counterexamples 2 and 3 for property 1 and counterexample 2 for property 2 for which MonPoly is unable to detect a violation until it reads the whole trace, thus the larger discrepancy of nearly one second (the delay we introduce at the end of a trace). We will discuss these counterexamples further.

Counterexample 2 for property 1 is given by the following trace.

```
@50ms  /a {data: 1}
@100ms /b {data: 2}
```

In this counterexample a response '`/b {data: 2}`' is produced for the stimulus '`/a {data: 1}`'. However, the property requires that '`/b {data = @A.data} within 100 ms`', which is not the case. The `data` field does not match, which is effectively the same as there being no response at all. In this case, our monitors fail via a timeout transition, 100 milliseconds after observing the stimulus. The average time to report a violation is 156 milliseconds, revealing that there are about 6 milliseconds of delay overall, likely due to the ROS network layer and the precision of the monitor's internal clock. MonPoly, on the other hand, is only capable of reporting the violation after the artificial delay we introduce at the end, before signaling the end of the trace. As explained in Chapter 6, this is because MonPoly expects further relevant events (`/a` or `/b`) with a different timestamp, to be sure that, at the 100 millisecond mark, there is only the single event '`/b {data: 2}`'. But the trace contains no more events. Only when we signal the end of the trace can MonPoly evaluate the formula and be sure that more than 100 milliseconds have elapsed since the stimulus message; it does not keep internal timers of its own.

For counterexamples 1.3 and 2.2 the story is similar. We are still dealing with Response properties, and the counterexample trace is the same in both cases.

```
@50ms /a {data: 1}
```

We simply publish a stimulus. There are no messages, at all, on the response's topic. Again, our monitors report failure after the 100 millisecond response window, while MonPoly is unable to do so before the end of the trace.

In terms of monitor creation time and memory consumption there are also discrepancies. The former is simple to understand – our implementation creates an object in memory, while the MonPoly implementation has to spawn a child process and establish communication between the two processes. On average, our monitors took about 0.133 milliseconds to create, while the MonPoly ones took about 14.512 milliseconds. Regarding memory consumption, we observed a consistent difference between implementations, with MonPoly requiring an additional 6 megabytes of memory. We attribute this to the memory required by MonPoly itself, as well as the additional threads and communication channels between processes.

In summary, our implementation appears to be competitive, in terms of performance, even though we are not using MonPoly to its fullest, i.e., a full OCaml process compiled to machine code. Still, there are numerous ways in which we can improve the performance of our own implementation. We used the standard Python interpreter, CPython, but there are others, such as PyPy or Cython that are known to be much faster (while having their own trade-offs). Alternatively, we could implement the monitors themselves in C or C++ and use Python's C and C++ extensions to import the resulting native code. It would be interesting to pursue such optimisations in the future, and then re-evaluate how both monitors perform.

8.4 Evaluation of the Property-based Tests

In Chapter 7 we proposed an approach to automate the testing of ROS systems based on the property specification language we presented in Chapter 4. Simply put, this testing approach takes a system specification and automatically generates property-based tests that try to falsify the given properties. In the context of this approach, a *ROS system* can be composed of any number of nodes; it is treated as a pure black-box with a publisher-subscriber interface. Properties are checked at runtime using our proposed runtime monitoring approach, presented in Chapter 6 and evaluated in Section 8.3.

We have found that the evaluation of test automation tools is not entirely straightforward. The evaluation criteria depend mainly on the target projects of the test automation tools, and the purpose of the evaluation itself. For instance, businesses seeking to adopt test automation tools tend to use qualitative (and partly subjective) criteria, such as:

- suitability for technical staff (e.g., developers and test automation experts);
- suitability for non-technical staff (e.g., business analysts and other non-programmers);
- quality and quantity of provided test templates;
- how effective the test automation tool is at identifying the right tests to prioritise.

On the other hand, researchers prefer objective, quantitative metrics, such as:

- number of (unique) test cases generated (see [166]);
- number of detected errors (see [141, 166]);
- mean time to failure (see [101]);
- diverse coverage metrics, e.g., model coverage, code coverage, state transition coverage (see [141, 166]).

We have not evaluated our testing approach, or the specification language, in terms of suitability for technical and non-technical roles, although an empirical study would be a good addition to our plan of future work. Our approach does not include any kind of prioritisation (all properties are given the same importance), and we only provide templates for all possible combinations of property scopes and patterns that our specification language allows. Thus, we evaluate our automated testing tool in terms of quantitative criteria, and we present the observed results in the remainder of this section.

8.4.1 *Methodology*

Systems Under Test

We use the TurtleBot2 and the AgRob V16 as our systems under test (SUT), in order to evaluate our testing approach. In particular, we aim to reuse the same configurations already used to evaluate the architecture model extraction, defined in Section 8.2, namely:

- TurtleBot2's Random Walker Controller configuration;
- TurtleBot2's AMCL Navigation configuration;
- AgRob V16's Basic configuration; and
- AgRob V16's Path Planning configuration.

There is one caveat, however. Our Property-based Testing tool must be able to reset the SUT multiple times, to guarantee that, for each generated example, we run the SUT from a stable (and hopefully deterministic) state. This series of resets occurs in rapid succession, and is entirely impractical if hardware is involved. Thus, we opt to modify the aforementioned configurations slightly, first to remove any nodes that directly control hardware (e.g., mobile bases or cameras), and second to remove any other nodes that become pointless after the first pass (e.g., image processing, after removing camera nodes). An alternative approach would be to use simulation nodes as a replacement for the hardware drivers, when available. However, node removal turns out to be more beneficial than node replacement for testing purposes, for two reasons: *(i)* the involvement of the simulator would have a considerable negative impact on the performance of the tests; *(ii)* removing nodes opens up several topics that we can exploit as inputs, versus a system that could, otherwise, be fully connected.

After removing nodes from all configurations, we notice that both pairs of configurations, from TurtleBot2 and AgRob V16, become very similar, retaining only a few small differences. In our opinion, these differences are too small to be relevant, so we opt for another route. We test only one configuration from each system, but, to compensate, we also test some nodes individually. For instance, in TurtleBot2 the *Safety Controller* node is a central component of the system. Instead of simply writing a specification for the full configuration, we first write a specification for this node, test it, and then proceed to the integrated environment. Not only does this reflect a more natural testing process adopted during development, it also helps us coming up with properties to write. Depending on the target system and on the specific nodes we are testing, some of the node's properties might convert, more or less directly, either to properties of the full system or to axioms that we want to assume for other tests. Note that, despite the additional work in writing the specifications, from the perspective of the testing tool there is no difference between testing individual nodes or entire systems – the SUT is always treated as a black-box, with input and output message topics.

Writing specifications

One of the main characteristics of Property-based Testing is that, in addition to the SUT, it also requires a (possibly partial) system specification. In this regard, we already face a limitation. While our specification language provides a comfortable array of features and operators, our test generator implementation is not able to support them all, at the time of writing. The most notable handicap is the lack of support for quantifiers, which makes writing predicates over arrays all the more daunting. Another limitation is the absence of arithmetic expressions, as these have the potential of making static checks for message generators hard to implement.

Another of trait of Property-based Testing, that is a consequence of the first one, is that all it does is finding discrepancies between implementation and specification [86]. Judgement on whether such discrepancies are an error in the system or in the specification is left to the user.

We have spent a considerable amount of time studying the implementations of our target systems, and, from the limited documentation sources that we have, we assume that they are relatively stable, with correct functional behaviour for the most part. This is a crucial assumption, as we are writing specifications mostly based on the actual (and expected) behaviour of the implementation. Given that PBT focuses on discrepancies, our evaluation process involves two types of testing, *positive testing* and *negative testing*.

Positive testing is the act of testing properties that we know to be true in advance. The main purpose of positive testing is simply to confirm that the implementation does not introduce false positives – i.e., that it does not report an error if there is none to report, regardless of the number of examples that we allow it to run. These are the first tests that we conduct. Albeit time-consuming, they reinforce our confidence that the properties we write describe the system's behaviour accurately.

Negative testing, on the other hand, is the act of testing properties that we know to be false in advance. In this case, the tool should be able to find at least one counterexample, for most properties – i.e., it should be able to detect discrepancies between specification and implementation. For complex or contrived counterexamples, it is possible that the tool does not hit the error unless we raise the limit of how many examples it can test for a given property; such is the nature of Property-based Testing.

Considering this, we need specifications that include both true and false properties. Instead of simply writing arbitrary properties, we follow a systematic method to write specifications. We start by writing a catalogue of true properties and axioms (assumptions) for each system. Since the goal of testing these properties is to rule out the presence of false positives, we opt for variety rather than quantity – i.e., we try to cover all the different types of scopes and patterns that our specification language allows for, rather than trying to be exhaustive. Then, we apply the principles of *specification mutation* [8, 28, 40, 91, 131, 163] to generate *mutants* (i.e., variants) from the initial catalogue of true properties. Mutations are often relatively small changes, such as changing a single operator or a variable. If the initial properties are as precise as we can write them, any small change introduced by the mutation process should end up generating a false property. However, this still implies a validation step to discard duplicates, trivial properties and *equivalent mutants* (properties that, despite being altered, remain true). We validate mutants manually, since the detection of equivalent mutants is an undecidable problem [163].

The existing literature provides many rules by which one can mutate specifications, depending on the underlying language and logic. The following list includes some common mutations, from which we draw inspiration.

- **Operand Replacement Operator**. Replace an operand, that is, a variable or constant, with another syntactically legal operand that does not result in a trivial expression (e.g., $x = x$). Replacing numbers should be avoided, as it may lead to a very large number of mutants.
- **Expression Negation Operator**. Replace an expression with its negation.
- **Logical Operator Replacement**. Replace a logical operator with another logical operator.

- **Relational Operator Replacement**. Replace a relational operator with any other relational operator, except its opposite (as this is covered by the negation rule).
- **Missing Condition Operator**. Delete conditions (only simple expressions) from conjunctions, disjunctions, and implications.
- **Shifting Timing Constraints**. Shift existing timing bounds either by increasing or decreasing their values.
- **Exchanging Input/Output Actions** (from Timed Automata specifications). Exchange an input/output action of a transition with another valid action.

Most of these rules are applicable to our language, to some extent. However, applying them blindly generates a large number of equivalent mutants. By refining the mutation ruleset, we can achieve a systematic process that yields far less equivalent mutants. Take, for instance, the following property as an example.

```
1 after /p {phi} until /q {psi}: /b {data > 0} requires /a {data < 0}
```

One of the rules that we can consider is the elimination of predicates, entirely. We can replace the activator event '`/p {phi}`' with a weaker version of it, '`/p`'. That is, if the original property states that the scope should only start after a specific message such that '`phi`' is true, stating, instead, that the scope should start after *any* message on that same topic is likely false – save for the exceptional case in which all messages can only have '`phi`' be true.

The same principle applies to the behaviour message. Knowing that '`/b {data > 0}`' requires a specific message '`/a {data < 0}`', does not imply that this remains true for any other message on '`/b`'. In fact, if the property describes the implementation's behaviour as accurately as possible, the mutant is likely false.

Applying the predicate elimination rule to '`/q {psi}`' or '`/a {data < 0}`' is not as straightforward. Weakening the predicates on these events yields a property that is likely to remain true, only more vague than the initial property. For instance, if '`/b {data > 0} requires /a {data < 0}`' is true, it is also true that '`/b {data > 0} requires /a`'. Put another way:

$$\exists x : a(x) \land (\exists y : data(x,y) \land y < 0) \qquad \text{implies} \qquad \exists x : a(x)$$

In a way, we face a similar challenge as in specifying program contracts in Hoare logic. We should strive for describing the weakest precondition and the strongest post-condition. Strengthening preconditions and weakening post-conditions yields mutants that cannot be killed. For this particular example, if we consider '`/a {data < 0}`' to be our post-condition, weakening it will never achieve a non-equivalent mutant. What we can do is replace the predicate with another class of values (e.g., '`/a {data > 0}`'), or strengthen the predicate (e.g., '`/a {data < 0 and data > -2}`'). Thus, we propose the following list of mutation operators.

- **Operand Replacement Operator**. Replace an operand in an expression with another syntactically valid operand of the same type. Avoid replacing number or string literals with other arbitrary literals,

as this could lead to too many mutants. Whether the resulting mutant is equivalent to the original property must be evaluated on a case by case basis. Not applicable to the provided example.

- **Expression Negation Operator**. Replace an expression with its negation. In general, this rule can always be applied. E.g., '`/p {phi}`' becomes '`/p {not phi}`'.
- **Logical Operator Replacement**. Replace a logical operator (`and`, `or`) with another logical operator. Replacing `or` with `and` in scope activators, stimuli messages (`causes` and `forbids` patterns) or behaviour messages (`no` and `requires` patterns) always produces equivalent mutants. Replacing `and` with `or` in precondition messages (`requires` pattern) or behaviour messages (`some`, `causes` and `forbids` patterns) always produces equivalent mutants. Not applicable to the provided example.
- **Relational Operator Replacement**. Replace a relational operator with another relational operator, except its opposite. E.g., '`/a {data < 0}`' becomes '`/a {data > 0}`' or '`/a {data = 0}`'.
- **Missing Predicate Operator**. Delete a predicate from an activator event, stimulus event or behaviour event (for `requires` properties only), as per the previous discussion. E.g., '`/p {phi}`' becomes '`/p`'.
- **Shifting Time Constraints**. Widen or shorten existing time constraints of a property, for instance by halving or doubling the specified time amount. Absence and Prevention properties (`no`, `forbids`) widen existing time constraints, while the remaining patterns (`some`, `requires`, `causes`) should shorten time constraints. Not applicable to the provided example.
- **Adding Time Constraints**. Adds a time bound to a property without one. The time constraint should be based on a sensible value, such as the publishing frequency of the SUT, to avoid generating a large number of examples. Liveness properties (`some`, `causes`) must always be bounded, so this rule is not applicable. Applying time constraints to absence and prevention patterns (`no`, `forbids`) always yields equivalent mutants. This mutation is only applicable to `requires` properties. E.g., '`/b {data > 0} requires /a {data < 0}`' becomes '`/b {data > 0} requires /a {data < 0} within 100 ms`'.
- **Topic Replacement Operator**. Exchange a published (resp. subscribed) topic with another published (resp. subscribed) topic of the same message type. E.g., '`/a {data < 0}`' becomes '`/c {data < 0}`'.
- **Scope Widening Operator**. Widen a scope by removing events, down to the global scope. Applying this operator to liveness properties (`some`, `causes`) always yields equivalent mutants. E.g., '`after /p {phi} until /q {psi}`' becomes '`after /p {phi}`', or '`until /q {psi}`' or '`globally`'.
- **Pattern Replacement Operator**. Replace a property pattern with another pattern. Replacing unary patterns with binary patterns (and vice-versa) always retains the behaviour event. In the former case, a suitable precondition/stimulus topic should be chosen from the open subscribed topics of the SUT. E.g., '`/b {data > 0} requires /a {data < 0}`' becomes '`no /b {data > 0}`' or '`/a {data < 0} forbids /b {data > 0}`'.

Evaluation Criteria

After we generate a catalogue of property mutants as per the previous ruleset, we do a manual check to discard equivalent, trivial and duplicate mutants. The final catalogue of properties can be found in Appendix D. Then, we run the PBT tool on each mutated property twice. For each property mutant, we are interested in gathering the following metrics (or the averages of the two runs):

- number of required examples until a test fails for the first time;
- number of required examples until the final test result (includes the shrinking phase);
- number of invalid examples (discarded due to being unable to satisfy assumptions);
- required time to run each example (both from start to end, including launching the SUT, and to replay the example trace);
- size of the produced counterexample trace.

With these metrics we are able to calculate aggregate metrics to provide an overview of the tool's performance. For instance, we are able to calculate:

- average number of examples needed to find a failure;
- average number of examples needed to report the final result;
- mean time to failure;
- excess messages in the counterexample input trace (i.e., how close it is to truly being a minimal counterexample);
- *specification coverage* [8], a metric that assigns a score to a test suite and tells how effective it is at exercising the given specification.

The penultimate metric involves a manual step. For each false property, we have to figure out a minimal counterexample and write down the size of its input trace, so that we can compare it to the one produced by the tool.

The considered criteria make for a fairly standard evaluation profile for a testing tool, with the notable exception that we lack code coverage metrics. The absence of code coverage is due to the fact that we are performing black-box testing. We refrain from altering the system's implementation, since code coverage is not the primary goal of our Property-based Testing approach. Despite this, an alternative evaluation using instrumented implementation code would be a good addition to our work, to assess how effective PBT truly is in exploring different program paths. Model and transition coverage, as seen in Model-based Testing, are not directly applicable in this setting. Instead, we consider specification coverage, a metric that is, in essence, a *mutation score*, i.e., the ratio of mutants that the testing suite kills. The major difference is that, instead of creating mutants of the implementation, we should mutate specifications, as per the process we described.

8.4.2 *Results for the TurtleBot2 Case Study*

We used an altered version of the TurtleBot2 Random Walker Controller configuration for our tests. In this case study, we test 3 key nodes individually besides the full system, namely the Safety Controller node, the Random Walker Controller node and the Multiplexer node. Most of the results show nothing out of the ordinary, and the observed numbers tend to follow the same patterns in all cases. To avoid repetition, we will present here our main observations for the Safety Controller node and the overall system. The results for the remaining two nodes can be found in Appendix E. No false positives have been observed, in any of the systems under test.

Safety Controller Node

For the Safety Controller node, we wrote a specification consisting of 12 axioms and 6 properties, as shown in Appendix D.1. Out of these properties, we were able to generate 59 useful mutants, while 40 mutants were discarded. The following table shows the distribution of the various language features in the final specification.

Language Feature	Axioms	Properties	Mutants
`globally`	12	4	36
`after`	-	-	1
`until`	-	1	11
`after` ...`until`	-	1	11
`no`	9	2	17
`some`	-	-	5
`causes`	-	2	22
`forbids`	3	1	10
`requires`	-	1	5
`within` ...`ms`	3	3	40

There are no existence properties (`some`) for this node due to the fact that its behaviour is reactive. If there are no incoming events from the base's sensors, the Safety Controller node remains silent for its entire execution.

In terms of test results, we achieved a near-perfect mutant score, failing to kill a single mutant with a limit of 200 examples. The mutant that we failed to kill requires a specific timing and interaction of messages, in an input trace that is, at least, 4 messages long. The original property and its mutant are given by the following statements.

```
1  The original property:
2 after /mobile_base/events/wheel_drop {state = DROPPED and wheel = LEFT}
3 until /mobile_base/events/wheel_drop {state = RAISED and wheel = LEFT}:
4     no /cmd_vel_mux/safety_controller {not linear.x = 0.0}
5
```

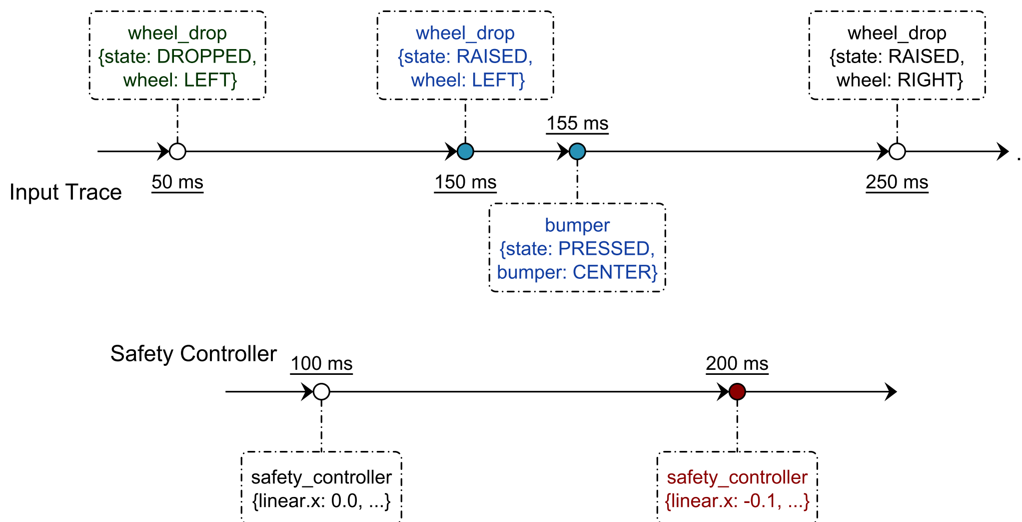


Figure 84: Hypothetical counterexample trace for the observed false negative.

```
6  The property's mutant:
7 after /mobile_base/events/wheel_drop {state = DROPPED and wheel = LEFT}
8 until /mobile_base/events/wheel_drop {state = RAISED and not wheel = LEFT}:
9     no /cmd_vel_mux/safety_controller {not linear.x = 0.0}
```

Simply put, the property states that from the moment that the robot detects a wheel drop until the moment that the same wheel is raised back to a normal level, no movement messages other than stop messages (linear velocity of zero) should be published. This is true because the Safety Controller node emits stop messages once it detects a wheel drop until both wheels are back up, and the wheel drop sensors have precedence over all other sensors. The mutant property introduces a single negation operator, '`not wheel = LEFT`', stating that only stop messages are permitted from the moment that one wheel drops until the moment that the *other* wheel is raised back up. The counterexample input trace, thus, includes four messages, two of which are the activator and the terminator, as seen in Figure 84.

Figure 84 shows both the input trace that we should generate and the sequence of messages produced by the Safety Controller node. The node evaluates its internal state and publishes a new command every 100 milliseconds. The scope activator message, sending a drop signal for the left wheel, is the first message of the input trace. This message leaves the node in a state such that it can only publish stop messages, as shown at the 100 millisecond mark. The last message of the input trace, shows the corresponding scope terminator message, a raise signal for a wheel that is *not* the left wheel. Between those messages, we must send a raise signal for the wheel that activated the scope (in this case, the left wheel, depicted in blue), so as to cancel the zero-only state. Then, still within the scope, we must publish a message on another sensor, such as a bumper press (in blue) that triggers a non-zero linear velocity on the next update of the Safety Controller node (shown at the 200 millisecond mark). The scope must not terminate too early, and Hypothesis has to figure out the two messages to send within the scope. Plus, they must be processed by the node still within the scope, and allow it enough time to reply back. It is not a trivial counterexample, by any means.

Regardless, the tool was able to detect errors for other similar examples of the same counterexample length, so we believe that this instance is not beyond the tool's capabilities, but is, rather, a matter of allowing it a greater number of examples. Out of the 58 killed mutants, 2 were reported as flaky tests, i.e., the results were not deterministic. Sometimes the same example was able to falsify the property, other times it was not. In this case, flaky tests happen with mutants that require timing interactions, such as falsifying the property via time bounds. For example, one of the mutants is the following property:

```
1 globally: some /cmd_vel_mux/safety_controller {not linear.x = 0.0} within 100 ms
```

Hypothesis found a counterexample (probably non-optimal, including random messages) for which, in some cases, the behaviour message is observed within the 100 millisecond window, while, in other cases, it is not observed in time. In general, this property is false, and the minimal counterexample is simply an empty trace – the node is reactive and will not publish any messages if not given the proper stimulus.

The table below summarises the registered results.

	Value
Mutants	59
Flaky Tests	2 (3.390%)
False Negatives	1 (1.695%)
Mutant Score	0.98305

Regarding other performance criteria, as per the following table, we did not observe anything out of the ordinary. On average, each mutant requires between 21 and 38 examples to falsify (including the shrinking phase). The tool tends to find an error for the first time relatively early (with less than 10 examples). For most mutants, no invalid examples were generated. Each example takes about 3.6 seconds to execute, and most of this time (2.6 seconds) is spent on launching and tearing down the SUT between examples. In total, this means that the average time to kill a mutant and obtain a counterexample is between one minute and three minutes. If a SUT could be reliably reset programatically (without killing the process and launching a new one), it would take about half a minute instead. Despite the relatively long execution time, there are only 3 mutants for which the tool uses more messages than necessary. One of these is the false negative, and another one is a flaky test, which might have misled the Hypothesis search strategy.

Metric	Mean	Std. Deviation	Median
Number of examples	38	43	21
Examples to failure	9	29	1
Invalid examples	2	8	0
Shrink attempts	29	27	20
Time per example (s)	3.596	1.238	3.192
Trace duration (s)	1.015	1.055	0.623
Setup/Teardown time (s)	2.581	0.183	2.569
Input trace size (messages)	2	3	1
Excess Messages	0	3	0

Random Walker Configuration

For the full Random Walker configuration, we wrote a specification consisting of 14 axioms and 5 properties, as shown in Appendix D.4. These axioms are a combination of the axioms for the Safety Controller and the Random Walker Controller nodes. Out of these properties, we were able to generate 45 useful mutants, while 34 mutants were discarded. The following table shows the distribution of the various language features in the final specification.

Language Feature	Axioms	Properties	Mutants
globally	14	4	35
after	-	-	1
until	-	-	-
after ...**until**	-	1	9
no	11	2	15
some	-	1	5
causes	-	1	14
forbids	3	-	5
requires	-	1	6
within ...**ms**	3	2	26

In terms of test results, we achieved a perfect mutant score. Out of the 45 killed mutants, only 2 were reported as flaky tests, with one of them being essentially the same property that led to false negatives in the Safety Controller and Random Walker Controller nodes. The table below summarises these results.

	Value
Mutants	45
Flaky Tests	2 (4.444%)
False Negatives	0
Mutant Score	1.0

Regarding other performance criteria, as per the following table, we did not observe anything out of the ordinary. The performance metrics seem to be on par with the tests for individual nodes. On average, each mutant requires about 20 examples to falsify (including the shrinking phase), considering the median of 16 and the mean of 22, although the tool tends to find an error for the first time relatively early (within the first 3 examples). Each example takes about 4.0 seconds to execute, and most of this time (3.2 seconds) is spent on launching and tearing down the SUT between examples. In total, this means that the average time to kill a mutant and obtain a counterexample is about 80 seconds. If a full ROS system could be reliably reset programatically (without killing the processes and launching new ones), it would take about 16 seconds instead. Despite the relatively long execution time, there are only 2 mutants for which the tool uses one message more than necessary. Curiously, in this case neither of the non-minimal counterexamples is associated with the two flaky mutants.

Metric	Mean	Std. Deviation	Median
Number of examples	22	23	16
Examples to failure	3	9	1
Invalid examples	3	12	0
Shrink attempts	19	17	15
Time per example (s)	3.967	1.602	3.309
Trace duration (s)	0.800	1.068	0.308
Setup/Teardown time (s)	3.167	0.535	3.001
Input trace size (messages)	1	1	1
Excess Messages	0	0	0

8.4.3 *Results for the AgRob V16 Case Study*

We used an altered version of the AgRob V16 Basic configuration for our tests. In this case study, we test 3 key nodes individually besides the full system, namely the Safety Controller node, the Supervisor node and the Joystick Controller node. Most of the results show nothing out of the ordinary, and the observed numbers tend to follow the same patterns in all cases. To avoid repetition, we will present here our main observations for the Supervisor node and the overall system. The results for the remaining two nodes can be found in Appendix E. No false positives have been observed, in any of the systems under test.

Supervisor Node

For the Supervisor node, we wrote a specification consisting of 10 axioms and 4 properties, as shown in Appendix D.6. One of the properties turned out to reveal two errors in the system (that have been reported to the developers and promptly fixed), so we consider it separately. Out of the remaining 3 properties, we were able to generate 36 useful mutants, while 30 mutants were discarded. The following table shows the distribution of the various language features in the final specification.

Language Feature	Axioms	Properties	Mutants
globally	10	2	8
after	-	-	1
until	-	-	2
after ...**until**	-	2	25
no	9	3	13
some	-	1	7
causes	-	-	8
forbids	1	-	4
requires	-	-	4
within ...**ms**	1	1	26

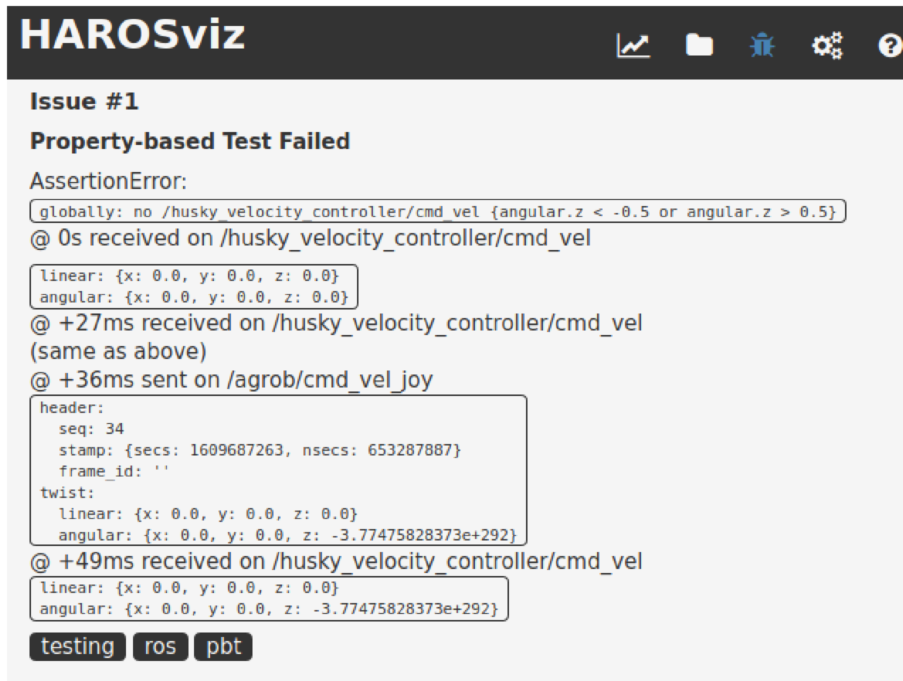


Figure 85: Error report for the error found in the AgRob V16 Supervisor.

There are no response, prevention or precedence properties (**causes**, **forbids**, **requires**) for this node due to the fact that they require axioms beyond our current tool's capabilities. Namely, some messages should not be produced too often, on multiple topics, and with the current implementation and input trace strategy approach, this just leads to the tests taking too long, or the tool giving up after a certain number of failed attempts at generating a valid input trace.

The property that led to us uncovering system errors is the following.

```
globally: no /husky_velocity_controller/cmd_vel {angular.z < -0.5 or angular.z > 0.5}
```

This property states that the Supervisor does not allow any velocity commands whose angular velocities (the speed at which the robot turns) are higher than 0.5 meters per second. Negative velocities turn to the left, positive velocities turn to the right. This value is set via ROS parameters, and 0.5 meters per second happens to be the value used in the AgRob V16 configurations; we replicate the same parameters for the individual node testing. The error (whose report is shown in Figure 85) happens in the following piece of C++ code, which is meant to fix the maximum velocities of a velocity message, should it surpass the intended limit.

```
// agrob_cmd is the velocity message to be published to the robot base
if (agrob_cmd.twist.linear.x > max_values[0]) {          // max. forward velocity
    agrob_cmd.twist.linear.x = max_values[0];
} else if (agrob_cmd.twist.linear.x < max_values[1]) {  // max. backward velocity
    agrob_cmd.twist.linear.x = max_values[1];
} else if (agrob_cmd.twist.angular.z > max_values[2]) { // max. angular velocity
    agrob_cmd.twist.angular.z = max_values[2];
}
```

The first error is the use of `else if`, instead of `if`, for the last statement. Fixing the linear and angular velocities should be independent steps. As it stands, a message that contains both linear and angular velocities beyond the limits would have its linear component correctly set to the maximum, but the angular component would remain unchecked. In practice, this would lead to the robot turning around too fast. In terms of safety, considering the size of the robot, the fact that it can incorporate a robotic arm, and that it operates in sloping terrain, the effects could be disastrous.

The second error is that the condition only checks for positive angular velocities (i.e., to turn right). The robot is allowed to turn left at speeds beyond the intended limit. The fix would be to compare the absolute value of the incoming velocity with the limit, rather than the raw value, or to add an additional `if` statement for negative velocities.

In terms of test results, we achieved a near-perfect mutant score, failing to kill two mutants with a limit of 200 examples. The mutants that we failed to kill are related to the same property (although irrelevant) and require very specific messages to detect the counterexample, in input traces that are, at least, 3 messages long. The original property and its mutants are given by the following statements.

```
1  The original property:
2  'x in y, z' is syntactic sugar for 'x = y or x = z'
3  after /agrob/safe_mode
4      {safe_modes in {"003", "009", "013", "019", "090", "092", "093", "099", "100",
5          "102", "103", "109", "110", "112", "113", "119", "190", "192", "193", "199"}}
6  until /agrob/safe_mode {safe_modes in {"000", "002", "010", "012"}}:
7      no /husky_velocity_controller/cmd_vel {linear.x > 0.0}
8
9  The property's mutants:
10 after /agrob/safe_mode
11 until /agrob/safe_mode {safe_modes in {"000", "002", "010", "012"}}:
12     no /husky_velocity_controller/cmd_vel {linear.x > 0.0}
13
14 after /agrob/safe_mode
15     {not safe_modes in {"003", "009", "013", "019", "090", "092", "093", "099", "100",
16         "102", "103", "109", "110", "112", "113", "119", "190", "192", "193", "199"}}
17 until /agrob/safe_mode {safe_modes in {"000", "002", "010", "012"}}:
18     no /husky_velocity_controller/cmd_vel {linear.x > 0.0}
```

Simply put, the property states that from the moment that any of those 20 safety states is registered until the moment that one of the other 4 safety states is registered, there should be no forward movement. The former safety states are all related to nearby obstacles detected by the laser sensors. The latter set of states denotes relative safety, in which forward movement should be allowed.

The first mutant drops the whole predicate of the activator message, meaning that any message on that topic should enter the state in which forward movement is forbidden. The counterexample involves sending one of the safe states (the same that can be seen for the terminator message), and then triggering forward movement, which requires a joystick velocity message with a timestamp that is less than one second old,

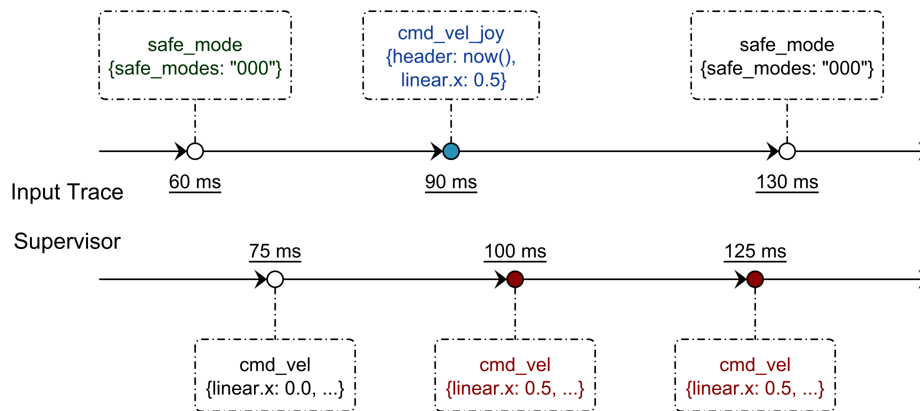


Figure 86: Counterexample trace for the first observed false negative.

relative to the current time at which the Supervisor node is running its processing iteration (once every 25 milliseconds). Figure 86 shows a possible counterexample.

The second mutant property introduces a single negation operator in the activator message, meaning that the activator should *not* be one of the dangerous safety states. This, combined with an axiom that enumerates all valid safety states, means that the second mutant can be killed with the same counterexample as the first one, shown in Figure 86.

The tool was able to detect errors for other mutants with similar counterexamples, in terms of length and structure, so we believe that these cases are not beyond the tool's capabilities, but are, rather, solvable with a larger number of examples. Out of the 34 killed mutants, 7 were reported as flaky tests, i.e., the results were not deterministic. Sometimes the same example was able to falsify the property, other times it was not. In all cases, flaky tests are related to timing. In particular, regarding the same issue as with the false negative mutants – that some messages must contain timestamps within a certain range. The table below summarises these results.

	Value
Mutants	36
Flaky Tests	7 (19.444%)
False Negatives	2 (5.556%)
Mutant Score	0.94444

Regarding other performance criteria, as per the following table, we did not observe anything out of the ordinary, except for the larger number of examples required to falsify properties. On average, each mutant requires about 19 examples to falsify (including the shrinking phase), although the tool tends to find an error for the first time relatively early (within the first 6 examples). Each example takes about 6 seconds to execute, and half of this time is spent on launching and tearing down the SUT between examples. This means that the average time to kill a mutant and obtain a counterexample is two minutes or less. We registered 7 outliers for the Supervisor node, taking between 200 and 400 examples to provide a verdict, and influencing the mean value. Most of these are either flaky tests or false negatives. There are only two mutants for which the tool uses more messages than necessary, and one of them is a flaky test.

Metric	Mean	Std. Deviation	Median
Number of examples	81	130	19
Examples to failure	25	91	1
Invalid examples	25	62	2
Shrink attempts	56	99	17
Time per example (s)	8.005	9.718	5.760
Trace duration (s)	4.708	7.077	2.789
Setup/Teardown time (s)	3.297	2.642	2.971
Input trace size (messages)	2	1	2
Excess Messages	0	1	0

AgRob V16 Basic Configuration

For the full AgRob V16 configuration, we wrote a specification consisting of 52 axioms and 4 properties, as shown in Appendix D.8. The axioms are essentially a collage of all axioms for individual nodes. Out of the 4 properties, we were able to generate 48 useful mutants, while 32 mutants were discarded. The following table shows the distribution of the various language features in the final specification.

Language Feature	Axioms	Properties	Mutants
`globally`	52	2	21
`after`	-	1	16
`until`	-	1	11
`after` ...**`until`**	-	-	-
`no`	52	2	8
`some`	-	-	4
`causes`	-	-	13
`forbids`	-	1	8
`requires`	-	1	15
`within` ...**`ms`**	-	1	27

The lack of response properties (**`causes`**) is mostly due to the same reason as in the Joystick Controller case, shown in Appendix E. Most properties depend on joystick input, and there is no rate limit for joystick messages. A message can be sent immediately after the stimulus, cancelling its effects, before the system has a chance to respond.

In terms of test results, our scores were much lower than in previous cases. Overall, the tool was unable to kill 11 mutants with a limit of 200 examples. Although the surviving mutants are related to different properties, the respective counterexamples share a common structure and require a specific message ordering, in input traces that are 3 or 4 messages long.

Most mutants start with a sensor message (either an activator or a stimulus) that represents a potentially dangerous scenario, e.g., too high a roll in a IMU message, or the camera detecting a person in front

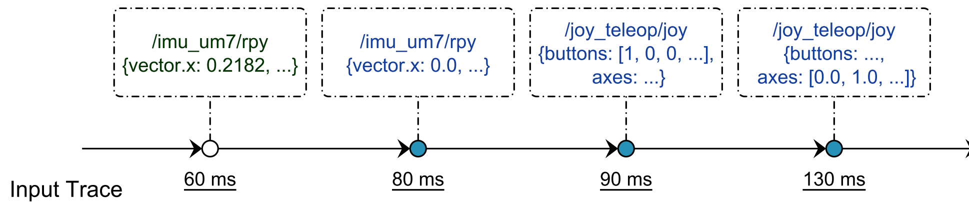


Figure 87: Counterexample trace structure for the observed false negatives.

of the robot. Then, either *(i)* they forbid forward movement messages, or *(ii)* they require another *safe* sensor message before a forward movement message is published. To falsify such mutants, the first thing to do is to produce another sensor message that cancels the first one, i.e., a sensor message, on the same topic, that asserts that the environment is safe. This is to prevent the Supervisor from discarding forward movement messages. The second thing we must do is to provide the actual joystick messages that translate to forward movement commands. This is done in two steps, as shown in Figure 87. The first message sets the button at index 0 to a value of 1 (indicating that the user pressed that specific button); this tells the Joystick Controller that the user wants to teleoperate the robot. The second message must contain a positive floating-point value, between 0 and 1, at the index 1 of the `axes` array; this is the joystick axis that is used for teleoperation. Considering that there are 8 axes and 11 buttons, the counterexample turns out to be a very specific combination of messages. Nonetheless, Hypothesis was able to find counterexamples following this structure for 3 other mutants.

Out of the 37 killed mutants, 4 were reported as flaky tests, i.e., the results were not deterministic. Sometimes the same example was able to falsify the property, other times it was not. In all cases, flaky tests are related to timing interactions. The table below summarises these results.

	Value
Mutants	48
Flaky Tests	4 (8.333%)
False Negatives	11 (22.917%)
Mutant Score	0.77083

Regarding other performance criteria, as per the following table, our results match the same scenarios as with testing the individual nodes, with a relatively large number of examples required to falsify properties. On average, each mutant requires about 18 examples to falsify (including the shrinking phase), although the tool tends to find an error for the first time relatively early (within the first 2 examples). Each example takes about 6 seconds to execute, and half of this time is spent on launching and tearing down the SUT between examples. This means that the average time to kill a mutant and obtain a counterexample is two minutes or less. We registered 5 outliers (besides the false negatives), taking about 400 examples to provide a verdict and raising the mean number of examples to 93. Two of the outliers are flaky tests. There are only four mutants for which the tool uses more messages than necessary, and two of them are flaky tests.

Metric	Mean	Std. Deviation	Median
Number of examples	93	139	18
Examples to failure	41	64	1
Invalid examples	0	0	0
Shrink attempts	53	129	13
Time per example (s)	7.094	2.893	5.903
Trace duration (s)	4.073	2.083	2.965
Setup/Teardown time (s)	3.021	0.810	2.937
Input trace size (messages)	2	2	1
Excess Messages	0	1	0

8.4.4 *Overview and Discussion*

Overall, we have observed a consistently good performance of our Property-based Testing approach. Having exercised more than 300 tests, almost evenly split across two robotic systems, and with a relatively low limit of maximum examples, only 18 property mutants have survived (about 5.6%). The system performed better with the TurtleBot2 case study, which is simpler in terms of code design and message interactions. The two most glaring drawbacks of our approach are the required time to execute the tests, and the limitations of the specification language.

In total, the tests for TurtleBot2 took 5 hours and a half to complete one round. Since we ran each property twice, this amounts to 11 hours. The tests for AgRob V16 took considerably longer, with nearly 30 hours to complete one round. Putting everything together, we have a total runtime of 71 hours.

In the remainder of this section we show the observed results for both systems combined, and discuss some specification challenges that we faced.

Global Overview

For both systems combined, we wrote a specification consisting of 148 axioms and 33 properties, as shown in Appendix D. Out of these properties, we were able to generate 322 useful mutants, while 233 mutants were discarded. The following table shows the distribution of the various language features in the final specification.

Language Feature	Axioms	Properties	Mutants
`globally`	148	19	168
`after`	-	2	35
`until`	-	6	55
`after` ...`until`	-	6	64
`no`	136	15	89
`some`	-	4	34
`causes`	-	6	100
`forbids`	12	3	44
`requires`	-	5	55
`within` ...`ms`	12	12	196

As we can see from this table, our axioms are fairly limited, mostly in the form of '`globally: no ...`', i.e., constructive axioms that specify what we expect from the values of input messages. We also have some `forbids` axioms, used to specify publishing rates when they are fixed and known in advance, e.g., '`/a forbids /a within 100 ms`'. As for the properties themselves, we can see an emphasis on global properties and absence properties. This is expectable, as these kinds of properties are often easier to identify. Causal relations between messages are more subtle, both because of the publisher-subscriber paradigm itself – which, as opposed to the client-server paradigm, does not dictate that every input should have an output – and because of the existence of obvious counterexamples, as we have discussed. Although some features have more coverage than others, we can see from the mutant column that no feature in particular is notably underrepresented.

In terms of test results, as aforementioned, our scores were mostly consistent across the various systems. Overall, the tool was unable to kill 18 mutants with a limit of 200 examples. Although the surviving mutants are vastly different, the respective counterexamples share a certain level of complexity and subtlety that make them unlikely to be found via random exploration with such a low maximum example limit. In some cases, the tool was able to find counterexamples for mutants with similar characteristics, leading us to believe that the current set of false negatives is not beyond the tool's capabilities, had we given it a sufficiently large margin to work with.

Out of the 304 killed mutants, 39 were reported as flaky tests, i.e., the results were not deterministic. Sometimes the same example was able to falsify the property, other times it was not. In many cases, flaky tests are related to timing interactions, which seems to be a weak spot of the current implementation. In a future version, we might explore the concept of simulated time in ROS. Just like robot simulators have the ability of speeding up or slowing down the clocks used in the ROS environment, we might harness this capability to manipulate time more meticulously. This would allow us to send messages at precise instants and craft traces that easily expose such subtle errors. The table below summarises the observed results.

	Value
Mutants	322
Flaky Tests	39 (12.112%)
False Negatives	18 (5.590%)
Mutant Score	0.94410

Regarding other performance criteria, the following table shows the observed average results. On average, each mutant requires about 18 examples to falsify (including the shrinking phase), although the tool tends to find an error for the first time relatively early (within the first 6 examples). Each example takes about 5 seconds to execute, and at least half of this time is spent on launching and tearing down the SUT between examples. This means that the average time to kill a mutant and obtain a counterexample is two minutes or less. We registered 22 outliers (excluding the false negatives), taking upwards of 200 examples to provide a verdict and raising the mean number of examples to 67. There are 29 mutants for which the tool uses more messages than necessary, and about half of them are flaky tests.

Metric	Mean	Std. Deviation	Median
Number of examples	67	128	18
Examples to failure	16	48	1
Invalid examples	2	9	0
Shrink attempts	50	117	16
Time per example (s)	5.385	5.888	4.625
Trace duration (s)	2.357	3.170	2.017
Setup/Teardown time (s)	3.029	2.718	2.608
Input trace size (messages)	2	2	1
Excess Messages	0	1	0

Specification Challenges

Although we do not have an evaluation process specifically dedicated to the specification language we proposed in Chapter 4, the evaluation of our Property-based Testing approach required a fair amount of specification. This gives us an opportunity to assess how useful the language is, in practice, to describe the behaviours of real robot systems, at least in qualitative terms.

Overall, the language was apt to describe the behaviours and coding patterns that we commonly observe in ROS nodes and systems. As stated in the original specification patterns paper by Dwyer et al. [58, 59], the proposed patterns and scopes suffice to capture most use cases. However, in the context of ROS and publisher-subscriber systems, we felt that there is plenty of room for future work, and that a few key patterns are not yet expressible with the current language.

First, we had to discard a good number of potential properties due to the lack of an *exception* statement. With *exception* we mean to specify a message that cancels further monitoring of the pattern, but without leaving the scope, especially in patterns that imply a sequence of messages, such as the response (**causes**)

and prevention (**forbids**) patterns. For instance, a common pattern in ROS – observed in the Safety Controller nodes of both the TurtleBot2 and the AgRob V16 systems – is to have message callbacks copy interesting fields of the most recent message to an internal state field that is to be processed at a later time. In the case of TurtleBot2, bumper, cliff and wheel drop sensor messages have their fields copied to the internal state of the node, which is evaluated once every 100 milliseconds. Intuitively, we tend to write properties such as the following.

```
1 globally: /mobile_base/events/bumper {state = PRESSED}
2     causes /cmd_vel_mux/safety_controller {not linear.x > 0} within 100 ms
```

The property above states that receiving a bumper message with a pressed bumper causes the Safety Controller to publish a velocity command that is either zero (a full stop) or negative (backward movement) within at most 100 milliseconds (the rate at which it runs its internal loop). But this is not true. It has an obvious counterexample, that our testing tool is quickly able to find. If a second bumper message is received immediately after the first one, cancelling it (i.e., stating that the bumper is no longer pressed), then, the node will never publish the expected velocity command. When it gets to its next iteration, it will see that the bumper is released, because the two previous messages were received too close to one another, and both before an iteration of the main loop. If we had an **unless** or **except** keyword, we could state that the **causes** relation holds within the specified time period, *unless* another message is received in the meantime (in this case, the message releasing the bumper). The final property would be:

```
1 globally: /mobile_base/events/bumper {state = PRESSED}
2     causes /cmd_vel_mux/safety_controller {not linear.x > 0} within 100 ms
3     unless /mobile_base/events/bumper {state = RELEASED}
```

Alternatively, we can specify axioms stating that bumper messages are only published every 100 milliseconds. This prevents the issue and allows for the original property to hold, but might not correspond to the real system behaviour anymore.

```
1 globally: /mobile_base/events/bumper forbids /mobile_base/events/bumper within 100 ms
2
3 globally: /mobile_base/events/bumper {state = PRESSED}
4     causes /cmd_vel_mux/safety_controller {not linear.x > 0} within 100 ms
```

The second major limitation that we faced was the lack of topic disjunctions, i.e., '`/a requires /b or /c`'. There are cases, both in TurtleBot2 and AgRob V16, where the same behaviour is triggered by different stimulus messages. For instance, in the TurtleBot2 Safety Controller node, backward movement is issued both for bumper and cliff sensors. We cannot simply state that backward movement requires a bumper message.

```
1 globally: /cmd_vel_mux/safety_controller {not linear.x > 0}
2     requires /mobile_base/events/bumper {state = PRESSED}
```

This property is false, and the testing tool quickly shows the counterexample where the cliff sensor triggers the behaviour instead of the bumper. We would really want to state that either message is the precondition for this behaviour.

```
globally: /cmd_vel_mux/safety_controller {not linear.x > 0}
    requires /mobile_base/events/bumper {state = PRESSED}
    or /mobile_base/events/cliff {state = CLIFF}
```

As it stands, both with this limitation and the previous one, we are restricted in what we can say about the global behaviour of the Safety Controller node. The causality relation in the forward direction (`causes`) is false, as we discussed, and so is the backward direction (`requires`).

The addition of disjunctions can, in a way, alleviate both issues. For the former property, with disjunctions, we could state:

```
globally: /mobile_base/events/bumper {state = PRESSED}
    causes /cmd_vel_mux/safety_controller {not linear.x > 0}
    or /mobile_base/events/bumper {state = RELEASED} within 100 ms
```

The modified property is no longer testable, as per our previous definition, since its behaviour event is now a disjunction of a published topic and an open subscribed topic. The approach would have to be refined to allow behaviour events such that *at least one* of the topics is published by the system. The open subscribed topics in the disjunction would be avoided in the generated input traces.

A final limitation is the specification of proper states of a state machine. We can approximate such specifications with the provided language scopes, but they only work for one activation topic and one termination topic at a time. Again, for states that are dependent on multiple variables, from different topics, we are unable to write any properties. Having topic disjunctions, at least, would help circumvent the issue, partially. We could use '`after /a or /b`' to initiate the same scope from multiple entry points, and use '`until /c or /d`' to terminate a scope with different messages as well. If we also had message sequences – e.g., '`after /a into /b`' – we would be able to cover most use cases. Alternatively, we could provide explicit support to define states, in the fashion of traditional state machine specifications. States would be entered and exited with messages, and we could define the appropriate behaviour while within a specific state.

```
after /a until /b: /c enters STATE

globally: /d exits STATE

during STATE: /e requires /f
```

8.5 Summary

Over the course of this thesis, we have implemented tools for model extraction, runtime verification and property-based testing. We have interweaved the tools under the HAROS framework, in a unified workflow that takes a partial system specification from the user, and checks whether the robotic system complies with the specification. We then evaluated the performance of the proposed tools at their respective tasks, when applied to two distinct case studies: the TurtleBot2, a popular open source robot used to learn and

research with ROS, and the AgRob V16, an agriculture robot, developed as part of a research project, with the aim of becoming a commercial product. In this chapter, we presented the results of these evaluation processes – which, in some cases, have shown near-optimal performance – and discussed the challenges we faced, as well as the limitations of the current approaches.

9

CONCLUSION

Robots are here to stay. The world population of service and industrial robots is increasing at a rapid rate, and the Robot Operating System is an established backbone of robotic software development. Initiatives such as the ROSIN EU Horizon 2020 project[1] and the ROS Quality Assurance Working Group[2] are fundamental steps in promoting established software engineering practices, such as Model-driven Engineering. However, despite their efforts, adoption by the general ROS community is still a slow work in progress. Traditional, code-first development processes are the norm, with software models nowhere to be seen.

In a fast developing world of robotic software where source code is the major (and, in some cases, the only) artefact one can work with, dependability cases become a useful mechanism to show that a system satisfies a given critical property, using evidence from various analysis techniques. Thus, we set out to answer the following research question.

> How can existing software analysis techniques (such as Model Checking, Runtime Verification, etc.) and tools be used by non-experts, to improve the quality of ROS applications and to provide the basis for dependability cases?

In the final chapter of this dissertation, we restate our thesis and summarise our contributions in this regard. We briefly discuss their practical impact in the community, so far, and opportunities for collaboration that presented themselves along the way. Lastly, we finish by recalling some of the shortcomings of our current approach, and setting out prospects for future work on this topic.

9.1 Summary

Over the course of this dissertation, we have provided evidence to support our thesis.

> Standard software analysis techniques can be employed behind an interface that caters to ROS roboticists. Namely, an interface that *(i)* takes source code as input, *(ii)* reverse engineers (formal) models as needed, and that *(iii)* uses a high-level property specification language that addresses ROS concepts directly.

1 https://www.rosin-project.eu/
2 https://discourse.ros.org/c/quality/

We have discussed a number of approaches to safety property verification in the literature that work well, to an extent, but are not really an answer to our original question. We have seen approaches in Specification-based Testing that target safety properties specified in Linear Temporal Logic [12, 128], but neither is this logic expressive enough to capture the full complexity of ROS systems, nor is it a friendly approach for non-experts. A similar situation happens in the domain of Runtime Verification. There are plenty of powerful tools [5, 18, 51, 67, 84, 85], some even working with more expressive logics than LTL, but they present a steep learning curve to non-experts or are simply not capable of verifying certain key properties. On the other end of the spectrum, some authors recognise the need to build architectural models of ROS applications [127, 142, 156, 170], as we do, in a user-friendly fashion. Their work, however, is limited to publishers and subscribers only, in some cases, and does not build upon the extracted models to check safety properties that address behaviour, rather than structure.

Our proposed approach combines the best of both worlds – ease of use by non-experts and verification of complex safety properties – in a single workflow. In this regard, we have made the following contributions:

1. a study on how ROS features are used in practice [149];
2. a metamodel for ROS applications [151];
3. an automatic procedure to build models from source code [151];
4. a specification language to annotate models with behavioural properties [41];
5. a property-based test generator [150];
6. the HAROS framework [148] for the analysis of ROS systems.

The first contribution is straightforward. It does not have a direct impact towards improving the quality of ROS applications, but it helped us prioritise certain features over others. For instance, our decision to base the property specification language around ROS publishers and subscribers was based on this study.

Establishing a metamodel is definitely the first step towards enabling ROS-specific analyses. The ROS community has not proposed an official metamodel, and the benefits of model-based approaches are too good to pass up. At the same time, it allows us to work at a convenient abstraction level of Nodes, Topics and other ROS resources, rather than work only with low-level source code entities. But source code entities are not completely disregarded either. Our metamodel includes such entities, in order to establish a traceability relation between source code entities and ROS computation graph resources. This is a distinguishing feature that we have not seen in other approaches, in spite of its usefulness in terms of lowering maintenance effort – i.e., issues can be traced back to a concrete source code artefact.

To complement the proposed metamodel, we put together an automatic model extraction procedure based on static analysis. The choice of static analysis benefits from ease of use and earlier detection, at the cost of increased complexity. In some cases, it might not be able to fully extract all entities. We address this limitation by presenting the incomplete entities up front, and allowing the user to provide the missing pieces of information. On top of these models, we are able to verify a large range of structural properties – e.g., *"all Topics have, at most, a single publisher"*, or *"there is only one Topic for laser scan messages"*. More importantly, being a mostly automated process, it is immediately usable by non-experts.

Our next contribution comes in to address a limitation of the model extraction process, as well as a shortcoming of ROS systems in general: the lack of behaviour properties. While automatic extraction of behaviour properties with static analysis is possible, it requires extensive effort and is, more than likely, very limited in terms of what it can deliver. Instead, we propose a specification language, designed around the exchange of ROS messages.

The language is minimalistic, yet capable of expressing many common properties. It is based on Metric First-Order Temporal Logic, in order to be able to capture real-time constraints as well as the full structure of message data. Its formal semantics enable us to reuse state of the art knowledge in property verification, while its syntax masks the details and lowers the barrier to adoption. In addition, this language is based on established specification patterns [58, 59] that capture a large portion of use cases. One of its drawbacks, in its current form, is its sole focus on Topics, leaving out Services and Parameters. This was a deliberate decision, given our time constraints. The decision to prioritise Topics over the other primitives was, as mentioned, based on our empirical study.

Our penultimate contribution shows one of the many applications of architectural models annotated with behavioural properties. We are able to generate black-box tests that target each of the specified properties, and rely on runtime monitors as oracles. From the architectural models, we know which topics to observe, and what message types are associated with each topic. From the property patterns we build test strategies that attempt to falsify the property. This approach is built on a Property-based Testing tool, that handles input space exploration and minimisation of the counterexamples it finds – yet another major step towards reducing maintenance effort. Our evaluation shows this technique to be effective at finding faults and providing minimal counterexamples – it has even uncovered a previously unknown fault in AgRob V16, a robotic platform for hillside agriculture. Its major drawback is the time it requires to run all generated tests.

Piecing together all contributions, as our final contribution, we achieved a workflow that has been fully integrated in the HAROS framework – a ROS-specific analysis tool, initially built as part of the Master's thesis that preceeded this work, that is progressively gaining traction among the ROS community.

9.2 Impact and Collaborations

Our work and our contributions (our technical contributions, in particular) are not purely academic. HAROS and our newly proposed workflow have gathered a fair number of users and supporters, especially in the ROS-Industrial community. They were featured multiple times, in talks and tutorials, in events hosted by the ROS-Industrial Consortium Europe, such as the ROS-Industrial Conference[3], for instance. HAROS has been used by a small number of companies, and is often used by researchers (both in the domains of Robotics and Software Engineering). It has also been promoted multiple times by the ROS Quality Assurance Working Group as one of the main, readily available tools to analyse ROS systems. More importantly, this line of research has opened up opportunities for a few interesting collaborations that we detail in this section.

3 https://rosindustrial.org/events/2016/11/3/2016-ros-industrial-conference
https://rosindustrial.org/events/2017/12/12/ros-industrial-conference-2017
https://rosindustrial.org/events/2018/12/11/ros-industrial-conference-2018

ROBUST – ROS Bug Study

We start with our longest-running collaboration effort, with members of the ROSIN European project, that is also the least directly related to this thesis. One of the objectives of the ROSIN project is to raise the overall quality of ROS robotics software. They aim to contribute to this goal by developing code scanners that continuously and automatically analyze ROS software, and detect, as well as report, programming errors and quality issues in the code. In this regard, tools such as HAROS are of great importance to the project.

In order to figure out what kind of analysis capabilities are needed – both to extend HAROS and to build other independent tools – we first need to understand what kind of errors ROS developers often make and what kind of code quality issues they often have. To this end, we systematically harvested 266 real documented bugs from a representative collection of ROS code repositories[4]. We then analysed this collection, classified each fault and failure according to a number of categories, and gathered various relevant statistics. The gathered observations will guide the subsequent development of new analysis tools in the ROSIN project.

Another goal of this study, besides collecting error reports, is to provide a repository of historically reproducible errors. Analysis and automated repair of historical bugs often requires access to the build and runtime environments in which those bugs were first identified and later fixed. Reproducing such historical environments is challenging, especially with ROS, due to the complex dependencies between ROS versions, tool versions and operating system versions. To accomplish this task, we built a series of tools that aid in tracking down the correct versions. Then, we build Docker[5] images of the environment and ROS application as they were at the time of the error report. In addition, we include test scripts within the reproduction image that show precisely the reported failure. The tests fail for the faulty version of the ROS repository, but pass for the fixed version. The work required to build test scripts for all cases proved to be quite a burden, which turned out to be an additional motivation for our test generator approach. Not all kinds of errors can be captured by our approach, because not all of them are related to software behaviour (e.g., build errors, missing dependencies, etc.), but it can be used to automate in a small number of cases.

At the time of writing, this work has only been presented at ROSCon 2019[6]. We are in the process of documenting our findings and preparing a submission to a top Software Engineering journal.

Model Extraction for ROS Python Code

Our architectural model extraction procedure, as previously stated in Chapter 5, was only implemented for the C++ ROS client library. C++ code makes up a large portion of the existing ROS ecosystem, being the go-to choice to implement low-level or performance-critical components, such as drivers and controllers. Python, on the other hand, is a common choice for high-level components, such as planners and orchestrators; components whose performance is not favoured over using an easier programming language. Automated

4 https://github.com/robust-rosin/robust
5 https://www.docker.com/
6 https://roscon.ros.org/2019/talks/roscon2019_188_bugs_later.pdf
https://vimeo.com/378916121

extraction of models for Python Nodes turned out to be a much-requested feature by HAROS users, and, eventually, it was proposed as a topic for a Master's thesis.

Davide Laezza, a Master's student of the SQUARE group[7] at the IT University of Copenhagen, under the supervision of professor Andrzej Wąsowski – group leader and one of the heads of the ROSIN project – picked up this task and replicated our model extraction approach for the ROS Python client library. Davide defended his thesis in April 2019, and his contribution is now a part of the current HAROS release.

Bootstrapping Model-Driven Engineering in ROS

Our metamodel for ROS applications, and the respective model extraction procedure, was the catalyst for a second collaborative work, this time with members of the Fraunhofer Institute for Manufacturing Engineering and Automation IPA[8]. The aim of this work is to promote Model-driven Engineering practices in the ROS community. In particular, it shows how modelling can be a complement, rather than an alternative, to manually written code. One way to achieve this is to build system models from existing source code – the same premise that backs our thesis. However, this research initiative does not rely solely on static analysis to build its models.

Its first contribution is an approach that merges information gathered at static time with information gathered at runtime. We handled the static analysis model extraction with HAROS, while the runtime part was implemented by members of Fraunhofer IPA. A second contribution is an easy-to-use web infrastructure that performs model extraction using this approach (to alleviate the burden on end users), while, at the same time, building a database of models extracted from various open source projects.

The main objective of this tooling, publicly available both as-a-service[9] and as source code[10], is to lower the MDE barrier for practitioners and leverage models to:

- improve the understanding of manually written code;
- perform correctness checks; and
- systematize the definition and adoption of best practices through large-scale generation of models from existing code.

This work, and a case study of its application on Care-O-bot 4[11] (a commercial mobile robot assistant to actively support humans in domestic environments), has been published at the MODELS 2019 conference [73]. An improved version of this work has been submitted [74] to the Software and Systems Modeling journal.

7 https://square.itu.dk/
8 https://www.ipa.fraunhofer.de/en.html
9 http://153.97.4.193/
10 https://github.com/ipa320/ros-model
11 https://www.care-o-bot.de/en/care-o-bot-4.html

HAROS Model Checking Plug-in

Lastly, Renato Carvalho, a student at our institution, worked on a new plug-in for HAROS as part of his Master's thesis. This thesis builds on our property specification language, but, rather than building tests, it is used for model checking purposes. Its main goal is to verify system-wide properties in ROS applications, i.e., properties that span a whole Configuration, rather than single Node behaviour. The proposed technique [41] is based in the formalisation of architectural models and node behaviour in Electrum[12] [35], over which system-wide specifications are subsequently model checked.

The overall workflow is essentially the same one we propose for automated testing.

1. The user annotates Nodes and Configurations, defined in a HAROS project file, with behavioural properties written in our proposed specification language (Chapter 4).
2. HAROS extracts the system's architecture, including the Computation Graph.
3. The model checking plug-in uses the system architecture, the user specification and the ROS message definitions to generate a Electrum model.
4. The model checker tries to falsify the model's assertions.
5. In the case that Electrum finds a counterexample for the property, the counterexample is presented to the user via the HAROS interface.

One of the main differences is that, since this approach is based on system-wide properties, node properties are used as axioms of the formalised model, while the system-wide properties are converted to assertions. Another difference is that Electrum is unable to handle real time. The current approach ignores real-time constraints and focuses on finding counterexamples only via classes of values or invalid sequences of messages.

This plug-in has been applied not only to the AgRob V16 case sutdy, but also to an internal case study provided by the VORTEX Colab[13] – a Collaborative Laboratory in Cyberphysical Systems and Cybersecurity. This second case study is a ROS prototype for an Advanced Driver-Assistance System.

9.3 Prospect for Future Work

We have established a workflow for the analysis and verification of safety properties for ROS applications. The techniques we rely on have some limitations that pose interesting research problems. Furthermore, our approach is not strictly tied to any particular verification technique. Its foundation, based on static analysis and model extraction, provides a great deal of freedom to build on top of. As we have seen, we can verify structural properties on the extracted models themselves, or we can use the models to generate property-based tests, or even to perform model checking. There are a number of other techniques that we can integrate in this workflow. In this section, we discuss some directions for future work that we are likely to explore.

12 http://haslab.github.io/Electrum/
13 http://www.vortex-colab.com/

On Static Analysis and Model Extraction

The first step of our proposed workflow is based on static analysis of source code. Without it, we have no access to models, and, thus, are incapable of verifying any properties of the target system. This, of course, depends on the availability of the source code in the first place. While it is often not an issue for systems under development, in many cases ROS developers tend to reuse off-the-shelf components. If such components are installed as binaries, rather than being built from source, they become opaque to our analysis. One of the measures we take to alleviate this problem, is to integrate within HAROS a database of hard-coded models from popular components of the standard ROS distribution. Maintaining such a database, however, is laborious, and is never going to cover all cases, in practice. This limitation is also one of the arguments that is often used in favour of dynamic analyses, rather than static analyses.

A massive improvement to this workflow, that would eliminate this initial limitation and make our approach feasible for any ROS system, would be to extend our static analysis techniques to also incorporate *binary static analysis*, i.e., analysing compiled binaries without executing them. This is based on state of the art techniques from the domain of reverse engineering, and is commonly used for security purposes [24, 94, 155, 162]. In this case, we would be using it to find calls to the ROS primitives, and reconstruct the architectural model. Powerful reverse engineering tools are freely available, e.g., Radare2[14], and could be used as a basis for this approach.

Binary static analysis could be seen as extending our static analysis capabilities in breadth. On the other hand, we can extend them in depth, using techniques from control flow analysis and data flow analysis. Inspired in previous works for ROS [142, 156], we can extend our metamodel to incorporate additional information about a Node's behaviour. Classifying publishers as proactive, reactive, periodic or rate-independent, we get some insights into the *message flow* of the system. Moreover, we can explore symbolic execution to determine predicates and properties about published data. This, in turn, would enable us to:

- automatically annotate Nodes with behavioural properties, using our specification language;
- verify behavioural properties in static time (making, e.g., model checking easier), rather than dynamically.

At the level of the models, we can also take various courses of action. One such course is based in the fact that many ROS Configurations are slight variations of one another. We have anecdotally seen this in multiple systems, and we have also mentioned this in Chapter 8 of this thesis. TurtleBot2 has numerous instances of very similar configurations, where the difference is in the presence or absence of a single node, such as the safety controller, or keyboard teleoperation, for instance. Even in AgRob V16, many configurations just change initialisation parameters, such as the map to be loaded or the maximum velocity limits. By extracting and comparing models across all configurations, we can find common denominators, build Feature Models and identify Software Product Lines, and their points of variability [34, 118, 119, 132, 173].

14 https://rada.re/n/index.html

Another course of action is to identify issues in the extracted architecture models – i.e., differences between the designed architecture and the implemented architecture [164] – and provide suggestions for architecture repair. This can be achieved in various forms, for instance by defining general architectural guidelines [115] or by using our extraction hint system, with hints being used to repair systems, rather than simply fix the models.

Last, but not least, we have proposed in Chapter 3 an extension to the metamodel itself, to incorporate high-level concepts provided by libraries, such as Actions and Dynamic Reconfiguration. Granted, these concepts are implemented on top of the basic building blocks of ROS (Topics, Services and Parameters). Even if the metamodel itself is not extended, we can, at least, improve the static analysis step to understand these compound primitives, and convert them into the respective low-level resources.

On Property Specification and Verification

As we have discussed in Chapter 8, there are a few ways to improve the property specification language itself. The most immediate improvements are the inclusion of event disjunctions, e.g., '`/a or /b`', and the specification of state machines. Other less straightforward enhancements would be the inclusion of other ROS resources, besides Topics, i.e., the inclusion of Services and Parameters. The other resources have fundamentally different mechanics from Topics, which would pose a challenge to specify their semantics. In particular, we anticipate Parameters, which are subject to concurrency and can change at any time without notification, to be the most challenging aspect to formalise.

In terms of Runtime Verification, the addition of Services and Parameters would require new monitoring approaches. Topics can be observed by any Node in the network, without restrictions. Services and Parameters, on the other hand, are based on one-to-one connections and cannot be externally observed without interfering with the system. In this regard, we have two main paths that we could follow.

1. Replace the ROS Master node and/or the client libraries with a modified version. Implementing a *debug* version of the core communication infrastructure of ROS would allow us to inject observers that would monitor Service or Parameter connections. This approach is transparent to the system under test and still allows for treating the system as a black box.
2. Instrument the target system, so that it monitors all calls to Services and Parameters, and relays the events to a central monitor. This approach does not require tampering with the main infrastructure of ROS, but requires building a separate, instrumented version of the target system. It is easier to adopt in a production environment, but it becomes challenging if the system's source code is not available.

In terms of Property-based Testing, as also discussed in Chapter 8, we believe that using simulated time in ROS would be benefitial to uncover hard-to-find timing interactions. However, such carefully crafted scenarios might hardly happen in a real environment, with real time. Furthermore, Nodes are not required to use the ROS-provided clocks, any Node can use real-time clocks instead (thus invalidating the use of simulated time). As such, this technique should be used as a complement, rather than as a substitute for

the current approach. In addition, counterexamples provided by exploiting simulated time would have to be considered with extra care. The exploited timing interactions might not be possible in a real environment for a multitude of reasons, with the most obvious being performance limitations – e.g., being impossible for a Node to produce multiple large messages within less than one millisecond of one another.

Other improvements to the Property-based Testing approach, as suggested in Chapter 7, are mainly concerned with being able to test more properties, and using smarter data generation. Our current approach requires that all topics referenced in a property (except for the behaviour message) be open subscribed topics. As we have discussed, this restriction can be alleviated, making tests more passive and more reliant on runtime monitors. Regarding data generation, we have proposed some best-effort approaches to analyse axioms statically and to automatically refine test schemas; these have not been implemented yet. We have also proposed the use of SMT solvers (Satisfiability Modulo Theories, e.g., Z3 [54]) or symbolic computing libraries (e.g., SymPy [123]) to embed arithmetic and quantified expressions directly into Hypothesis strategies. In addition, it is also worth experimenting with custom shrinking approaches to generated input traces, to aim for shorter tests overall.

On Technical Improvements

Lastly, on a technical side, and covering the whole workflow, our approach can naturally be extended to support ROS2. As more companies and researchers transition to the new version of ROS, for its improved communication infrastructure and real-time capabilities, this feature has been requested multiple times for HAROS. Some of the differences between ROS and ROS2 are minimal. For instance, to extract architectural models from C++ source code, it requires little more than searching for different function names in the parsed Abstract Syntax Tree. However, other changes have a significant impact in the overall approach, with launch files being a notorious one. The original launch files were deemed to be rather inflexible, despite allowing composition, conditional statements and namespace grouping. In ROS2, launch files were replaced by Python scripts. This change alone greatly diminishes the usefulness of static analysis. It is likely that, in the event that we are not able to fully parse the launch script, we would have to ask the user for parsing hints. Alternatively, we would have to move away from static analysis and run the script itself in a sandbox environment.

On a lighter note, ROS2 provides support to implement Nodes that follow a standardised life cycle. This is something that can be leveraged to enhance the extraction of both the system's architecture as well as the behaviour. Furthermore, it represents an out-of-the-box mechanism to *reset* a system's state without shutting it down and bringing it back up, which is one of the major performance obstacles for our Property-based Testing approach, as we have seen in Chapter 8.

BIBLIOGRAPHY

[1] M. Abi-Antoun and J. Aldrich. Static extraction of sound hierarchical runtime object graphs. In *ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, pages 51–64, 2009. doi: 10.1145/1481861.1481869.

[2] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire. Reverse engineering architectural feature models. In *European Conference on Software Architecture (ECSA)*, pages 220–235, 2011. doi: 10.1007/978-3-642-23798-0_25.

[3] K. Adam, K. Hölldobler, B. Rumpe, and A. Wortmann. Engineering robotics software architectures with exchangeable model transformations. In *IEEE International Conference on Robotic Computing (IRC)*, pages 172–179, 2017. doi: 10.1109/IRC.2017.16.

[4] M. S. Adam, M. Larsen, K. Jensen, and U. P. Schultz. Rule-based dynamic safety monitoring for mobile robots. *Journal of Software Engineering for Robotics*, 7(1):120–141, 2016.

[5] S. Adam, M. Larsen, K. Jensen, and U. P. Schultz. Towards rule-based dynamic safety monitoring for mobile robots. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pages 207–218, 2014. doi: 10.1007/978-3-319-11900-7_18.

[6] A. Alami, Y. Dittrich, and A. Wasowski. Influencers of quality assurance in an open source community. In *International Workshop on Cooperative and Human Aspects of Software Engineering (ICSE)*, pages 61–68, 2018. doi: 10.1145/3195836.3195853.

[7] N. Alzahrani, M. Spichkova, and J. O. Blech. From temporal models to property-based testing. In *International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pages 241–246, 2017. doi: 10.5220/0006340302410246.

[8] P. Ammann and P. E. Black. A specification-based coverage metric to evaluate test sets. In *IEEE International Symposium on High-Assurance Systems Engineering (HASE)*, pages 239–248. IEEE Computer Society, 1999. doi: 10.1109/HASE.1999.809499.

[9] A. Anand and R. A. Knepper. ROSCoq: Robots powered by constructive reals. In *International Conference on Interactive Theorem Proving (ITP)*, pages 34–50, 2015. doi: 10.1007/978-3-319-22102-1_3.

[10] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov. Evaluating the “small scope hypothesis”. Technical Report MIT-LCS-TR-921, MIT CSAIL, 2003.

[11] M. Aniche. Software testing: From theory to practice. https://sttp.site/. [Online; accessed 30-November-2020].

[12] P. Arcaini, A. Gargantini, and E. Riccobene. Online testing of LTL properties for java code. In *International Haifa Verification Conference on Hardware and Software: Verification and Testing (HVC)*, volume 8244 of *Lecture Notes in Computer Science*, pages 95–111. Springer, 2013. doi: 10.1007/978-3-319-03077-7_7.

[13] J. L. Armstrong and S. R. Virding. Erlang – an experimental telephony programming language. In *International Switching Symposium*, volume 3, pages 43–48. IEEE, 1990.

[14] T. Arts, J. Hughes, J. Johansson, and U. T. Wiger. Testing telecoms software with quviq quickcheck. In *ACM SIGPLAN Workshop on Erlang*, pages 2–10, 2006. doi: 10.1145/1159789.1159792.

[15] D. P. Attard and A. Francalanza. A monitoring tool for a branching-time logic. In *International Conference on Runtime Verification (RV)*, pages 473–481, 2016. doi: 10.1007/978-3-319-46982-9_31.

[16] G. Bardaro and M. Matteucci. Using AADL to model and develop ROS-based robotic application. In *IEEE International Conference on Robotic Computing (IRC)*, pages 204–207, 2017. doi: 10.1109/IRC.2017.59.

[17] J. Barnat, L. Brim, and J. Stríbrná. Distributed LTL model-checking in SPIN. In *International SPIN Workshop on Model Checking Software*, pages 200–216, 2001. doi: 10.1007/3-540-45139-0_13.

[18] D. A. Basin, M. Harvan, F. Klaedtke, and E. Zalinescu. MONPOLY: monitoring usage-control policies. In *International Conference on Runtime Verification (RV)*, volume 7186 of *Lecture Notes in Computer Science*, pages 360–364, 2011. doi: 10.1007/978-3-642-29860-8_27.

[19] D. A. Basin, F. Klaedtke, and E. Zalinescu. Failure-aware runtime verification of distributed systems. In *IARCS Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 590–603, 2015. doi: 10.4230/LIPIcs.FSTTCS.2015.590.

[20] D. A. Basin, F. Klaedtke, and E. Zalinescu. The monpoly monitoring tool. In *International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES)*, volume 3 of *Kalpa Publications in Computing*, pages 19–28, 2017.

[21] A. Bauer and Y. Falcone. Decentralised LTL monitoring. *Formal Methods in System Design*, 48 (1-2):46–93, 2016. doi: 10.1007/s10703-016-0253-8.

[22] A. Bauer and M. Leucker. The theory and practice of SALT. In *NASA Formal Methods (NFM)*, pages 13–40, 2011. doi: 10.1007/978-3-642-20398-5_3.

[23] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4):14:1–14:64, 2011. doi: 10.1145/2000799.2000800.

[24] J. Bergeron, M. Debbabi, M. M. Erhioui, and B. Ktari. Static analysis of binary code to isolate malicious behaviors. In *Workshop on Enabling Technologies, Infrastructure for Collaborative Enterprises (WETICE)*, pages 184–189. IEEE Computer Society, 1999. doi: 10.1109/ENABL.1999.805197.

[25] G. Bernot, M. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991. doi: 10.1049/sej.1991.0040.

[26] A. Bihlmaier and H. Wörn. Robot unit testing. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pages 255–266, 2014. doi: 10.1007/978-3-319-11900-7_22.

[27] A. Bihlmaier and H. Wörn. Increasing ROS reliability and safety through advanced introspection capabilities. In *44. Jahrestagung der Gesellschaft für Informatik*, pages 1319–1326, 2014.

[28] P. E. Black, V. Okun, and Y. Yesha. Mutation operators for specifications. In *IEEE International Conference on Automated Software Engineering (ASE)*, page 81, 2000. doi: 10.1109/ASE.2000.873653.

[29] Y. Blein. *ParTraP: A Language for the Specificationand Runtime Verification of Parametric Properties*. PhD thesis, Grenoble Alpes University, France, 2019. URL https://tel.archives-ouvertes.fr/tel-02269062.

[30] I. Block. Robot science museum in seoul will be built by robots and drones. https://www.dezeen.com/2019/02/20/robot-science-museum-melike-altinisik-architects-maa-seoul/. [Online; accessed 19-September-2020].

[31] R. Bloem, G. Fey, F. Greif, R. Könighofer, I. Pill, H. Riener, and F. Röck. Synthesizing adaptive test strategies from temporal logic specifications. *Formal Methods in System Design*, 55(2):103–135, 2019. doi: 10.1007/s10703-019-00338-9.

[32] E. Bodden. A lightweight LTL runtime verification tool for java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 306–307, 2004. doi: 10.1145/1028664.1028776.

[33] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, 2(1):14–23, 1986. doi: 10.1109/JRA.1986.1087032.

[34] D. Brugali and N. Hochgeschwender. Software product line engineering for robotic perception systems. *International Journal of Semantic Computing*, 12(1):89–108, 2018. doi: 10.1142/S1793351X18400056.

[35] J. Brunel, D. Chemouil, A. Cunha, and N. Macedo. The electrum analyzer: Model checking relational first-order temporal specifications. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 884–887. ACM, 2018. doi: 10.1145/3238147.3240475.

[36] S. G. Brunner, F. Steinmetz, R. Belder, and A. Dömel. RAFCON: A graphical tool for engineering complex, robotic tasks. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3283–3290, 2016. doi: 10.1109/IROS.2016.7759506.

[37] H. Bruyninckx. Open robot control software: the OROCOS project. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2523–2528. IEEE, 2001. doi: 10.1109/ROBOT.2001.933002.

[38] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. K. Kraetzschmar, L. Gherardi, and D. Brugali. The BRICS component model: a model-based development paradigm for complex robotics software systems. In *ACM Symposium on Applied Computing (SAC)*, pages 1758–1764, 2013. doi: 10.1145/2480362.2480693.

[39] A. Bubeck, F. Weißhardt, and A. Verl. BRIDE - a toolchain for framework-independent development of industrial service robot applications. In *International Symposium on Robotics (ISR/Robotik)*, pages 1–6. VDE, 2014.

[40] T. A. Budd and A. S. Gopal. Program testing by specification mutation. *Computer Languages*, 10 (1):63–73, 1985. doi: 10.1016/0096-0551(85)90011-6.

[41] R. Carvalho, A. Cunha, N. Macedo, and A. Santos. Verification of system-wide safety properties of ROS applications. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2020.

[42] I. Cassar, A. Francalanza, L. Aceto, and A. Ingólfsdóttir. A survey of runtime monitoring instrumentation techniques. In *International Workshop on Pre- and Post-Deployment Verification Techniques (PrePost@iFM)*, pages 15–28, 2017. doi: 10.4204/EPTCS.254.2.

[43] N. Chinchor. MUC-4 evaluation metrics. In *Conference on Message Understanding (MUC)*, pages 22–29. ACL, 1992. doi: 10.3115/1072064.1072067.

[44] N. Chinchor and B. Sundheim. MUC-5 evaluation metrics. In *Conference on Message Understanding (MUC)*, pages 69–78. ACL, 1993. doi: 10.3115/1072017.1072026.

[45] S. Chitta, E. Marder-Eppstein, W. Meeussen, V. Pradeep, A. R. Tsouroukdissian, J. Bohren, D. Coleman, B. Magyar, G. Raiola, M. Lüdtke, and E. F. Perdomo. ros_control: A generic and simple control framework for ROS. *Journal of Open Source Software*, 2(20):456, 2017. doi: 10.21105/joss.00456.

[46] J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Transactions on Database Systems*, 20(2):149–186, 1995. doi: 10.1145/210197.210200.

[47] M. Christakis and K. Sagonas. Detection of asynchronous message passing errors using static analysis. In *International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 5–18, 2011. doi: 10.1007/978-3-642-18378-2_3.

[48] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 268–279, 2000. doi: 10.1145/351240.351266.

[49] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Workshop on Logics of Programs*, pages 52–71, 1981. doi: 10.1007/BFb0025774.

[50] E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In *International Conference on Computer Aided Verification (CAV)*, pages 415–427, 1994. doi: 10.1007/3-540-58179-0_72.

[51] L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, and D. Thoma. TeSSLa: Temporal stream-based specification language. In *Brazilian Symposium on Formal Methods: Foundations and Applications (SBMF)*, pages 144–162, 2018. doi: 10.1007/978-3-030-03044-5_10.

[52] W. Curran, T. Thornton, B. Arvey, and W. D. Smart. Evaluating impact in the ROS ecosystem. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 6213–6219, 2015. doi: 10.1109/ICRA.2015.7140071.

[53] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *International Conference on Software Engineering (ICSE)*, pages 285–294, 1999. doi: 10.1145/302405.302640.

[54] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24.

[55] L. R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3): 297–302, 1945. doi: https://doi.org/10.2307/1932409.

[56] D. D'Souza. A logical characterisation of event clock automata. *International Journal of Foundations of Computer Science*, 14(4):625–640, 2003. doi: 10.1142/S0129054103001923.

[57] J. Duregård. *Automating Black-Box Property Based Testing*. PhD thesis, Chalmers University of Technology, Gothenburg, Sweden, 2016. URL `http://publications.lib.chalmers.se/publication/240807-automating-black-box-property-based-testing`.

[58] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *Workshop on Formal Methods in Software Practice*, pages 7–15, 1998. doi: 10.1145/298595.298598.

[59] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering (ICSE)*, pages 411–420, 1999. doi: 10.1145/302405.302672.

[60] J. P. Ernits, E. Halling, G. Kanter, and J. Vain. Model-based integration testing of ROS packages: A mobile robot case study. In *European Conference on Mobile Robots (ECMR)*, pages 1–7, 2015. doi: 10.1109/ECMR.2015.7324210.

[61] P. Estefo, J. Simmonds, R. Robbes, and J. Fabry. The robot operating system: Package reuse and community dynamics. *Journal of Systems and Software*, 151:226–242, 2019. doi: 10.1016/j.jss.2019.02.024.

[62] Y. Falcone, S. Krstić, G. Reger, and D. Traytel. A taxonomy for classifying runtime verification tools. In *International Conference on Runtime Verification (RV)*, pages 241–262, Nov. 2018. doi: 10.1007/978-3-030-03769-7_14.

[63] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical report, DTIC Document, 2006.

[64] G. Fink and M. Bishop. Property-based testing: A new approach to testing for assurance. *SIGSOFT Software Engineering Notes*, 22(4):74–80, July 1997. ISSN 0163-5948. doi: 10.1145/263244.263267.

[65] S. Fleury, M. Herrb, and R. Chatila. G^{en}om: a tool for the specification and the implementation of operating modules in a distributed robot architecture. In *IEEE/RSJ International Conference on Intelligent Robot and Systems (IROS)*, pages 842–849. IEEE, 1997. doi: 10.1109/IROS.1997.655108.

[66] R. Fraer, G. Kamhi, L. Fix, and M. Y. Vardi. Evaluating semi-exhaustive verification techniques for bug hunting. *Electronic Notes in Theoretical Computer Science*, 23(2):11–22, 1999. doi: 10.1016/S1571-0661(04)80665-0.

[67] A. Francalanza and A. Seychell. Synthesising correct concurrent runtime monitors - (extended abstract). In *International Conference on Runtime Verification (RV)*, pages 112–129, 2013. doi: 10.1007/978-3-642-40787-1_7.

[68] A. Francalanza, J. A. Pérez, and C. Sánchez. Runtime verification for decentralised and distributed systems. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, pages 176–210. 2018. doi: 10.1007/978-3-319-75632-5_6.

[69] L. Fredlund, Á. Herranz-Nieva, and J. Mariño. Applying property-based testing in teaching safety-critical system programming. In *Euromicro Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA)*, pages 309–316, 2015. doi: 10.1109/SEAA.2015.53.

[70] D. M. Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. In *Temporal Logic in Specification*, pages 409–448, 1989. doi: 10.1007/3-540-51803-7_36.

[71] D. M. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 163–173, 1980. doi: 10.1145/567446.567462.

[72] D. Ganesan, M. Lindvall, L. Ruley, R. Wiegand, V. Ly, and T. Tsui. Architectural analysis of systems based on the publisher-subscriber style. In *Working Conference on Reverse Engineering (WCRE)*, pages 173–182, 2010. doi: 10.1109/WCRE.2010.27.

[73] N. H. Garcia, L. Delval, M. Lüdtke, A. Santos, B. Kahl, and M. Bordignon. Bootstrapping MDE development from ROS manual code - part 2: Model generation. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 95–105. IEEE, 2019. doi: 10.1109/MODELS.2019.00-11.

[74] N. H. Garcia, H. Deshpande, A. Santos, B. Kahl, and M. Bordignon. Bootstrapping MDE development from ROS manual code - part 2: Model generation and leveraging models at runtime. *Software and Systems Modeling*, 2021 (Accepted).

[75] E. Gat. On three-layer architectures. *Artificial Intelligence and Mobile Robots*, 195:210, 1998.

[76] V. Graefe and R. Bischoff. From ancient machines to intelligent robots – a technical evolution. In *International Conference on Electronic Measurement & Instruments (ICEMI)*, pages 3–418–3–431, 2009.

[77] V. Gribov and H. Voos. Safety oriented software engineering process for autonomous robots. In *IEEE Conference on Emerging Technologies & Factory Automation (ETFA)*, pages 1–8, 2013. doi: 10.1109/ETFA.2013.6647969.

[78] R. Halder, J. Proença, N. Macedo, and A. Santos. Formal verification of ROS-based robotic applications using timed-automata. In *IEEE/ACM International FME Workshop on Formal Methods in Software Engineering (FormaliSE@ICSE)*, pages 44–50, 2017. doi: 10.1109/FormaliSE.2017.9.

[79] S. Hallé and R. Villemaire. Runtime monitoring of message-based workflows with data. In *International IEEE Enterprise Distributed Object Computing Conference (ECOC)*, pages 63–72, Sep. 2008. doi: 10.1109/EDOC.2008.32.

[80] S. Hallé and R. Villemaire. Flexible and reliable messaging using runtime monitoring. In *IEEE International Enterprise Distributed Object Computing Conference (EDOCw)*, pages 116–125, 2009. doi: 10.1109/EDOCW.2009.5332002.

[81] K. Havelund, D. Peled, and D. Ulus. DejaVu: A monitoring tool for first-order temporal logic. In *Workshop on Monitoring and Testing of Cyber-Physical Systems (MT@CPSWeek)*, pages 12–13, 2018. doi: 10.1109/MT-CPS.2018.00013.

[82] C. Heer. Robots double worldwide by 2020. `https://ifr.org/ifr-press-releases/news/robots-double-worldwide-by-2020`. [Online; accessed 19-September-2020].

[83] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5): 279–295, 1997. doi: 10.1109/32.588521.

[84] C. Hu, W. Dong, Y. Yang, H. Shi, and G. Zhou. Runtime verification on hierarchical properties of ROS-based robot swarms. *IEEE Transactions on Reliability*, pages 1–16, 2019. doi: 10.1109/TR.2019.2923681.

[85] J. Huang, C. Erdogan, Y. Zhang, B. M. Moore, Q. Luo, A. Sundaresan, and G. Rosu. ROSRV: runtime verification for robots. In *International Conference on Runtime Verification (RV)*, pages 247–254, 2014. doi: 10.1007/978-3-319-11164-3_20.

[86] J. Hughes. Experiences with quickcheck: Testing the hard stuff and staying sane. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 169–186, 2016. doi: 10.1007/978-3-319-30936-1_9.

[87] IFR. Mobile robot transports sterile goods in hospital. `https://ifr.org/ifr-press-releases/news/mobile-robot-transports-sterile-goods-in-hospital`, . [Online; accessed 19-September-2020].

[88] IFR. Battling the coronavirus with uv lighting. `https://ifr.org/ifr-press-releases/news/battling-the-coronavirus-with-uv-lighting`, . [Online; accessed 19-September-2020].

[89] D. Jackson. A direct path to dependable software. *Communications of the ACM*, 52(4):78–88, 2009. doi: 10.1145/1498765.1498787.

[90] D. Jackson and C. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 239–249, 1996. doi: 10.1145/229000.226322.

[91] Y. Jiang, S. Hou, J. Shan, L. Zhang, and B. Xie. An approach to testing black-box components using contract-based mutation. *International Journal of Software Engineering and Knowledge Engineering*, 18(1):93–117, 2008. doi: 10.1142/S0218194008003556.

[92] A. Kane, O. Chowdhury, A. Datta, and P. Koopman. A case study on runtime monitoring of an autonomous research vehicle (ARV) system. In *International Conference on Runtime Verification (RV)*, pages 102–117. Springer International Publishing, 2015. ISBN 978-3-319-23820-3. doi: 10.1007/978-3-319-23820-3_7.

[93] G. Kanter and J. Vain. TestIt: an open-source scalable long-term autonomy testing toolkit for ROS. In *International Conference on Dependable Systems, Services and Technologies (DESSERT)*, pages 45–50, 2019. doi: 10.1109/DESSERT.2019.8770011.

[94] N. Karampatziakis. Static analysis of binary executables using structural svms. In *Advances in Neural Information Processing Systems 23*, pages 1063–1071. Curran Associates, Inc., 2010.

[95] R. M. Karp. An algorithm to solve the $m \times n$ assignment problem in expected time $O(mn \log n)$. *Networks*, 10(2):143–152, 1980. doi: 10.1002/net.3230100205.

[96] A. Khalili, L. Natale, and A. Tacchella. Reverse engineering of middleware for verification of robot control architectures. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pages 315–326, 2014. doi: 10.1007/978-3-319-11900-7_27.

[97] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4): 255–299, 1990. doi: 10.1007/BF01995674.

[98] K. Krogmann. *Reconstruction of Software Component Architectures and Behaviour Models Using Static and Dynamic Analysis*. PhD thesis, Karlsruhe Institute of Technology, Germany, 2010.

[99] P. S. Kumar, W. Emfinger, A. Kulkarni, G. Karsai, D. Watkins, B. Gasser, C. Ridgewell, and A. Anilkumar. ROSMOD: a toolsuite for modeling, generating, deploying, and managing distributed real-time component-based software using ROS. In *International Symposium on Rapid System Prototyping (RSP)*, pages 39–45, 2015. doi: 10.1109/RSP.2015.7416545.

[100] L. Lamport, J. Matthews, M. R. Tuttle, and Y. Yu. Specifying and verifying systems with TLA+. In *ACM SIGOPS European Workshop*, pages 45–48. ACM, 2002. doi: 10.1145/1133373.1133382.

[101] L. Lampropoulos, M. Hicks, and B. C. Pierce. Coverage guided, property based testing. *ACM on Programming Languages*, 3(OOPSLA):181:1–181:29, 2019. doi: 10.1145/3360607.

[102] R. Larrieu and N. Shankar. A framework for high-assurance quasi-synchronous systems. In *ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 72–83, 2014. doi: 10.1109/MEMCOD.2014.6961845.

[103] M. Larsen, M. Adam, U. Schultz, and R. Jørgensen. *Towards Automatic Consistency Checking of Software Components in Field Robotics*, pages 409–418. 2014. ISBN 978-84-697-0248-2.

[104] T. Latvala, A. Biere, K. Heljanko, and T. A. Junttila. Simple bounded LTL model checking. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 186–200, 2004. doi: 10.1007/978-3-540-30494-4_14.

[105] C. Lesire, D. Doose, and H. Cassé. Mauve: a component-based modeling framework for real-time analysis of robotic applications. In *Workshop on Software Development and Integrationin Robotics (SDIR-VII ICRA)*, 2012.

[106] C. Lesire, S. Roussel, D. Doose, and C. Grand. Synthesis of real-time observers from past-time linear temporal logic and timed specification. In *International Conference on Robotics and Automation (ICRA)*, pages 597–603, 2019. doi: 10.1109/ICRA.2019.8793754.

[107] W. Li, L. Gérard, and N. Shankar. Design and verification of multi-rate distributed systems. In *ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 20–29, 2015. doi: 10.1109/MEMCOD.2015.7340463.

[108] X. Li, R. Wang, Y. Jiang, Y. Guan, X. Li, and X. Song. Formal modeling and automatic code synthesis for robot system. In *International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 146–149, 2017. doi: 10.1109/ICECCS.2017.17.

[109] A. Löscher and K. Sagonas. Targeted property-based testing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 46–56. ACM, 2017. doi: 10.1145/3092703.3092711.

[110] A. Löscher and K. Sagonas. Automating targeted property-based testing. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 70–80, 2018. doi: 10.1109/ICST.2018.00017.

[111] A. Löscher, K. Sagonas, and T. Voigt. Property-based testing of sensor networks. In *IEEE International Conference on Sensing, Communication, and Networking (SECON)*, pages 100–108, 2015. doi: 10.1109/SAHCN.2015.7338296.

[112] D. MacIver, Z. Hatfield-Dodds, and M. Contributors. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 11 2019. ISSN 2475-9066. doi: 10.21105/joss.01891.

[113] D. R. MacIver. What is property based testing? `https://hypothesis.works/articles/what-is-property-based-testing/`. [Online; accessed 30-November-2020].

[114] D. R. MacIver. Hypothesis 5.6.0. https://github.com/HypothesisWorks/hypothesis, 2020.

[115] I. Malavolta, G. Lewis, B. Schmerl, P. Lago, and D. Garlan. How do you architect your robots? state of the practice and guidelines for ros-based systems. *ICSE-CEIP. ACM*, 10(3377813.3381358), 2020.

[116] Z. Manna and A. Pnueli. Verification of concurrent programs, part I: The temporal framework. In *The Correctness Problem in Computer Science*, pages 215–273, 1981.

[117] N. Markey. Temporal logic with past is exponentially more succinct. *Bulletin of the EATCS*, 79: 122–128, 2003.

[118] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon. Automating the extraction of model-based software product lines from model variants (T). In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 396–406. IEEE Computer Society, 2015. doi: 10.1109/ASE.2015.44.

[119] J. Martinez, X. Tërnava, and T. Ziadi. Software product line extraction from variability-rich systems: The robocode case study. In *International Systems and Software Product Line Conference - Volume 1 (SPLC)*, pages 132–142. ACM, 2018. doi: 10.1145/3233027.3233038.

[120] B. Mayer and R. Weinreich. An approach to extract the architecture of microservice-based software systems. In *IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 21–30, 2018. doi: 10.1109/SOSE.2018.00012.

[121] W. Meng, J. Park, O. Sokolsky, S. Weirich, and I. Lee. Verified ROS-based deployment of platform-independent control systems. In *NASA Formal Methods (NFM)*, pages 248–262, 2015. doi: 10.1007/978-3-319-17524-9_18.

[122] G. Metta, P. Fitzpatrick, and L. Natale. YARP: Yet Another Robot Platform. *International Journal of Advanced Robotic Systems*, 3(1):8, 2006. doi: 10.5772/5761.

[123] A. Meurer, C. P. Smith, M. Paprocki, O. Certík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, S. Roucka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. M. Scopatz. SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103, 2017. doi: 10.7717/peerj-cs.103.

[124] M. Micallef and C. Colombo. Lessons learnt from using DSLs for automated software testing. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–6, 2015. doi: 10.1109/ICSTW.2015.7107472.

[125] A. Michlmayr, P. Fenkam, and S. Dustdar. Specification-based unit testing of publish/subscribe applications. In *International Conference on Distributed Computing Systems Workshops (ICDCS)*, page 34, 2006. doi: 10.1109/ICDCSW.2006.103.

[126] M. Mostafa and B. Bonakdarpour. Decentralized runtime verification of LTL specifications in distributed systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 494–503, 2015. doi: 10.1109/IPDPS.2015.95.

[127] B. J. Muscedere, R. Hackman, D. Anbarnam, J. M. Atlee, I. J. Davis, and M. W. Godfrey. Detecting feature-interaction symptoms in automotive software using lightweight analysis. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 175–185, 2019. doi: 10.1109/SANER.2019.8668042.

[128] M. Narizzano, L. Pulina, A. Tacchella, and S. Vuotto. Automated requirements-based testing of black-box reactive systems. In *NASA Formal Methods (NFM)*, volume 12229 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 2020. doi: 10.1007/978-3-030-55754-6_9.

[129] J. P. Near, A. Milicevic, E. Kang, and D. Jackson. A lightweight code analysis and its role in evaluation of a dependability case. In *International Conference on Software Engineering (ICSE)*, pages 31–40. ACM, 2011. doi: 10.1145/1985793.1985799.

[130] N. J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann Publishers Inc., 1980. ISBN 0934613109.

[131] V. Okun. *Specification Mutation For Test Generation And Analysis*. PhD thesis, University of Maryland, Baltimore County, 2004.

[132] P. Oliveira, G. Vale, P. A. Júnior, and H. Costa. Extraction of a software product line using conditional compilation - an exploratory study. In *Latin American Computing Conference (CLEI)*, pages 1–10. IEEE, 2019. doi: 10.1109/CLEI47609.2019.9089045.

[133] J. Ore, C. Detweiler, and S. G. Elbaum. Lightweight detection of physical unit inconsistencies without program annotations. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 341–351, 2017. doi: 10.1145/3092703.3092722.

[134] J. Ore, C. Detweiler, and S. G. Elbaum. Phriky-units: A lightweight, annotation-free physical unit inconsistency detection tool. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 352–355, 2017. doi: 10.1145/3092703.3098219.

[135] J. Ore, S. G. Elbaum, and C. Detweiler. Dimensional inconsistencies in code and ROS messages: A study of 5.9M lines of code. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 712–718, 2017. doi: 10.1109/IROS.2017.8202229.

[136] D. L. Parnas, A. J. van Schouwen, and S. P. Kwan. Evaluation of safety-critical software. *Communications of the ACM*, 33(6):636–648, 1990. doi: 10.1145/78973.78974.

[137] S. Pearson. The digital and electronic revolution: Some important milestones. `http://www.thepeoplehistory.com/electronics.html`. [Online; accessed 19-September-2020].

[138] I. Perez and H. Nilsson. Testing and debugging functional reactive programming. *PACMPL*, 1(ICFP): 2:1–2:27, 2017. doi: 10.1145/3110246.

[139] M. Pichler, B. Dieber, and M. Pinzger. Can I depend on you? mapping the dependency and quality landscape of ROS packages. In *IEEE International Conference on Robotic Computing (IRC)*, pages 78–85, 2019. doi: 10.1109/IRC.2019.00020.

[140] A. Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science*, pages 46–57, 1977. doi: 10.1109/SFCS.1977.32.

[141] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. *CoRR*, abs/1701.06815, 2017.

[142] R. Purandare, J. Darsie, S. G. Elbaum, and M. B. Dwyer. Extracting conditional component dependence for distributed robotic systems. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1533–1540, 2012. doi: 10.1109/IROS.2012.6385719.

[143] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: An open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009. URL https://www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf.

[144] L. Ramasubramanian. The digital revolution. In *Geographic Information Science and Public Participation*, Advances in Geographic Information Science, pages 19–32. Springer, 2008. ISBN 978-3-540-75400-8.

[145] N. Rapin. ARTiMon monitoring tool, the time domains. In *International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES)*, pages 106–122, 2017.

[146] J. Raskin and P. Schobbens. State clock logic: A decidable real-time logic. In *International Workshop on Hybrid and Real-Time Systems (HART)*, pages 33–47, 1997. doi: 10.1007/BFb0014711.

[147] C. Riva and J. V. Rodríguez. Combining static and dynamic views for architecture reconstruction. In *European Conference on Software Maintenance and Reengineering (CSMR)*, page 47, 2002. doi: 10.1109/CSMR.2002.995789.

[148] A. Santos, A. Cunha, N. Macedo, and C. Lourenço. A framework for quality assessment of ROS repositories. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4491–4496, 2016. doi: 10.1109/IROS.2016.7759661.

[149] A. Santos, A. Cunha, N. Macedo, R. Arrais, and F. N. dos Santos. Mining the usage patterns of ROS primitives. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3855–3860, 2017. doi: 10.1109/IROS.2017.8206237.

[150] A. Santos, A. Cunha, and N. Macedo. Property-based testing for the robot operating system. In *ACM SIGSOFT International Workshop on Automating Test Case Design, Selection, and Evaluation (A-TEST@ESEC/SIGSOFT FSE)*, pages 56–62, 2018. doi: 10.1145/3278186.3278195.

[151] A. Santos, A. Cunha, and N. Macedo. Static-time extraction and analysis of the ROS computation graph. In *IEEE International Conference on Robotic Computing (IRC)*, pages 62–69, 2019. doi: 10.1109/IRC.2019.00018.

[152] L. Santos, F. N. dos Santos, S. Magalhães, P. Costa, and R. Reis. Path planning approach with the extraction of topological maps from occupancy grid maps in steep slope vineyards. In *IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 1–7. IEEE, 2019. doi: 10.1109/ICARSC.2019.8733630.

[153] J. Schumann, P. Moosbrugger, and K. Y. Rozier. R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. In *International Conference on Runtime Verification (RV)*, pages 233–249, 2015. doi: 10.1007/978-3-319-23820-3_15.

[154] I. Segura-Bedmar, P. Martínez, and M. Herrero-Zazo. SemEval-2013 task 9 : Extraction of drug-drug interactions from biomedical texts (ddiextraction 2013). In *International Workshop on Semantic Evaluation (SemEval@NAACL-HLT)*, pages 341–350, 2013.

[155] A. Sepp, B. Mihaila, and A. Simon. Precise static analysis of binaries by extracting relational information. In *Working Conference on Reverse Engineering (WCRE)*, pages 357–366. IEEE Computer Society, 2011. doi: 10.1109/WCRE.2011.50.

[156] N. Sharma, S. G. Elbaum, and C. Detweiler. Rate impact analysis in robotic systems. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2089–2096, 2017. doi: 10.1109/ICRA.2017.7989240.

[157] C. E. Silva and J. C. Campos. Combining static and dynamic analysis for the reverse engineering of web applications. In *ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS)*, pages 107–112, 2013.

[158] M. Sokolova, N. Japkowicz, and S. Szpakowicz. Beyond accuracy, f-score and ROC: A family of discriminant measures for performance evaluation. In *Advances in Artificial Intelligence (AI)*, volume 4304 of *Lecture Notes in Computer Science*, pages 1015–1021. Springer, 2006. doi: 10.1007/11941439_114.

[159] T. J. Sørensen. A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on Danish commons. *Det Kongelige Danske Videnskabernes Selskab*, 5(4):1–34, 1948.

[160] D. Stampfer, A. Lotz, M. Lutz, and C. Schlegel. The SmartMDSD toolchain: An integrated MDSD workflow and integrated development environment (IDE) for robotics software. *Journal of Software Engineering for Robotics (JOSER)*, 7:3–19, 08 2016.

[161] L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with linear temporal logic. In *IEEE International Conference on Information Reuse and Integration (IRI)*, pages 493–498, 2004. doi: 10.1109/IRI.2004.1431509.

[162] H. Theiling. Extracting safe and precise control flow from binaries. In *International Workshop on Real-Time Computing and Applications Symposium (RTCSA)*, pages 23–30. IEEE Computer Society, 2000. doi: 10.1109/RTCSA.2000.896367.

[163] M. S. A. Trab, S. Counsell, and R. M. Hierons. Specification mutation analysis for validating timed testing approaches based on timed automata. In *IEEE Computer Software and Applications Conference (COMPSAC)*, pages 660–669, 2012. doi: 10.1109/COMPSAC.2012.93.

[164] J. B. Tran and R. C. Holt. Forward and reverse repair of software architecture. In *Conference of the Centre for Advanced Studies on Collaborative Research*, page 12. IBM, 1999.

[165] M. Y. Vardi. Branching vs. linear time: Final showdown. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 1–22, 2001. doi: 10.1007/3-540-45319-9_1.

[166] L. Villalobos-Arias, C. Quesada-López, A. Martínez, and M. Jenkins. Evaluation of a model-based testing platform for Java applications. *IET Software*, 14(2):115–128, 2020. doi: 10.1049/iet-sen.2019.0036.

[167] M. Webster, C. Dixon, M. Fisher, M. Salem, J. Saunders, K. L. Koay, and K. Dautenhahn. Formal verification of an autonomous personal robotic assistant. In *AAAI Spring Symposia*, 2014.

[168] M. Webster, C. Dixon, M. Fisher, M. Salem, J. Saunders, K. L. Koay, K. Dautenhahn, and J. Saez-Pons. Toward reliable autonomous robotic assistants through formal verification: A case study. *IEEE Transactions on Human-Machine Systems*, 46(2):186–196, 2016. doi: 10.1109/THMS.2015.2425139.

[169] M. Wenger, W. Eisenmenger, G. Neugschwandtner, B. Schneider, and A. Zoitl. A model based engineering tool for ROS component compositioning, configuration and generation of deployment information. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2016. doi: 10.1109/ETFA.2016.7733559.

[170] T. Witte and M. Tichy. Checking consistency of robot software architectures in ROS. In *IEEE/ACM International Workshop on Robotics Software Engineering (RoSE)*, pages 1–8, 2018. doi: 10.1145/3196558.3196559.

[171] S. Zaman, G. Steinbauer, J. Maurer, P. Lepej, and S. Uran. An integrated model-based diagnosis and repair architecture for ROS-based robot systems. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 482–489, 2013. doi: 10.1109/ICRA.2013.6630618.

[172] S. Zander, G. Heppner, G. Neugschwandtner, R. Awad, M. Essinger, and N. Ahmed. A model-driven engineering approach for ROS using ontological semantics. *CoRR*, abs/1601.03998, 2016.

[173] B. Zhang and M. Becker. Code-based variability model extraction for software product line improvement. In *International Software Product Line Conference (SPLC)*, pages 91–98. ACM, 2012. doi: 10.1145/2364412.2364428.

A

FICTIBOT'S SOURCE CODE

This appendix chapter contains the source code of our Fictibot running example, so as to provide a reference version that will not change over time and to make the document self-contained.

a.1 Fictibot Driver Package

Figure 88: The `fictibot_drivers` package tree.

```xml
<?xml version="1.0"?>
<package>
  <name>fictibot_drivers</name>
  <version>0.1.0</version>
  <description>The fictibot_drivers package</description>

  <maintainer email="andre.f.santos@inesctec.pt">Andre Santos</maintainer>
  <license>MIT</license>
  <author email="andre.f.santos@inesctec.pt">Andre Santos</author>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>std_msgs</run_depend>
</package>
```

Listing A.1: The *package.xml* file.

```
cmake_minimum_required(VERSION 2.8.3)
project(fictibot_drivers)

find_package(catkin REQUIRED COMPONENTS roscpp std_msgs)
catkin_package(INCLUDE_DIRS include CATKIN_DEPENDS roscpp std_msgs)
include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(fictibot_driver
  src/driver_node.cpp
  src/sensor_manager.cpp
  src/motor_manager.cpp
)

add_dependencies(fictibot_driver
  ${${PROJECT_NAME}_EXPORTED_TARGETS}
  ${catkin_EXPORTED_TARGETS}
)

target_link_libraries(fictibot_driver ${catkin_LIBRARIES})
```

Listing A.2: The *CMakeLists.txt* file.

```
#ifndef MOTOR_MANAGER_H_
#define MOTOR_MANAGER_H_

#include <ros/ros.h>
#include <std_msgs/Empty.h>
#include <std_msgs/Float64.h>

class MotorManager {
public:
  MotorManager(ros::NodeHandle& n, double hz);
  ~MotorManager(){};
  void spin();
private:
  int teleop_priority_, stop_cycles_;
  double command_, velocity_;
  ros::Subscriber teleop_subscriber_, controller_subscriber_,
                  stop_subscriber_;

  void teleop_callback(const std_msgs::Float64::ConstPtr& msg);
  void controller_callback(const std_msgs::Float64::ConstPtr& msg);
  void stop_callback(const std_msgs::Empty::ConstPtr& msg);
  void apply_commands();
};

#endif /*MOTOR_MANAGER_H_*/
```

Listing A.3: The *include/fictibot_drivers/motor_manager.h* file.

```
1  #ifndef SENSOR_MANAGER_H_
2  #define SENSOR_MANAGER_H_
3
4  #include <ros/ros.h>
5
6  class SensorManager {
7  public:
8    SensorManager(ros::NodeHandle& n, double hz);
9    ~SensorManager(){};
10   void spin();
11 private:
12   ros::Publisher bumper_publisher_, laser_publisher_,
13                  wheel_drop_publisher_;
14   int8_t read_bumper();
15   int8_t read_laser();
16   int8_t read_wheels();
17 };
18
19 #endif /*SENSOR_MANAGER_H_*/
```

Listing A.4: The *include/fictibot_drivers/sensor_manager.h* file.

```
#define _USE_MATH_DEFINES

#include <math.h>
#include "fictibot_drivers/motor_manager.h"

MotorManager::MotorManager(ros::NodeHandle& n, double hz)
  : teleop_priority_(0)
  , stop_cycles_(0)
  , command_(0)
  , velocity_(0) {
  uint32_t queue_size = (uint32_t) hz * 2 + 1;
  stop_subscriber_ = n.subscribe("stop_cmd", queue_size,
      &MotorManager::stop_callback, this);
  teleop_subscriber_ = n.subscribe("teleop_cmd", queue_size,
      &MotorManager::teleop_callback, this);
  controller_subscriber_ = n.subscribe("controller_cmd", queue_size,
      &MotorManager::controller_callback, this);
}

void MotorManager::spin() {
  ros::spinOnce();
  apply_commands();
}

void MotorManager::teleop_callback(const std_msgs::Float64::ConstPtr& msg) {
  command_ = msg->data;
  teleop_priority_ = 5;
}

void MotorManager::controller_callback(const std_msgs::Float64::ConstPtr& msg) {
  if (teleop_priority_ < 0) { command_ = msg->data; }
}

void MotorManager::stop_callback(const std_msgs::Empty::ConstPtr& msg) {
  stop_cycles_ = 50;
}
```

Listing A.5: The *src/motor_manager.cpp* file (part 1 of 2).

```
void MotorManager::apply_commands() {
  teleop_priority_--;
  stop_cycles_--;
  if (stop_cycles_ >= 0) {
    velocity_ -= 0.125;
    if (velocity_ < 0) { velocity_ = 0; }
    return;
  }
  if (command_ > M_PI / 8 || command_ < -M_PI / 8) {
    velocity_ -= 0.125;
    if (velocity_ < 0) { velocity_ = 0; }
    if (velocity_ == 0) {
      if (command_ < 0) {
        command_ += M_PI / 64;
        if (command_ > 0) { command_ = 0; }
      } else {
        command_ -= M_PI / 64;
        if (command_ < 0) { command_ = 0; }
      }
    }
  } else {
    if (command_ < 0) {
      command_ += M_PI / 64;
      if (command_ > 0) { command_ = 0; }
    } else {
      command_ -= M_PI / 64;
      if (command_ < 0) { command_ = 0; }
    }
  }
  if (command_ == 0) {
    velocity_ += 0.0625;
    if (velocity_ > 0.5) { velocity_ = 0.5; }
  }
}
```

Listing A.6: The *src/motor_manager.cpp* file (part 2 of 2).

```cpp
#include <cstdlib>
#include <std_msgs/Int8.h>
#include "fictibot_drivers/sensor_manager.h"

SensorManager::SensorManager(ros::NodeHandle& n, double hz) {
  uint32_t queue_size = (uint32_t) hz * 2 + 1;
  bumper_publisher_ = n.advertise<std_msgs::Int8>("bumper", queue_size);
  laser_publisher_ = n.advertise<std_msgs::Int8>("laser", queue_size);
  wheel_drop_publisher_ = n.advertise<std_msgs::Int8>("wheel", queue_size);
}

void SensorManager::spin() {
  std_msgs::Int8 bumper_msg;
  bumper_msg.data = read_bumper();
  bumper_publisher_.publish(bumper_msg);
  std_msgs::Int8 laser_msg;
  laser_msg.data = read_laser();
  laser_publisher_.publish(laser_msg);
  std_msgs::Int8 wheel_msg;
  wheel_msg.data = read_wheels();
  wheel_drop_publisher_.publish(wheel_msg);
}

int8_t SensorManager::read_bumper() {
  // left[0,1] | center[0,1] | right[0,1]
  return (int8_t) (std::rand() % 8);
}

int8_t SensorManager::read_laser() {
  return (int8_t) (std::rand() % 128);
}

int8_t SensorManager::read_wheels() {
  // left[0,3] | right[0,3]
  return (int8_t) (std::rand() % 16);
}
```

Listing A.7: The *src/sensor_manager.cpp* file.

```cpp
#include <ros/ros.h>
#include "fictibot_drivers/sensor_manager.h"
#include "fictibot_drivers/motor_manager.h"

int main(int argc, char **argv) {
  ros::init(argc, argv, "fictibot_driver");
  ros::NodeHandle n;
  SensorManager sensor_man(n, 10 /*Hz*/);
  MotorManager motor_man(n, 10 /*Hz*/);
  ros::Rate loop_rate(10 /*Hz*/);
  while (ros::ok()) {
    sensor_man.spin();
    motor_man.spin();
    loop_rate.sleep();
  }
  return 0;
}
```

Listing A.8: The *src/driver_node.cpp* file.

a.2 Fictibot Controller Package

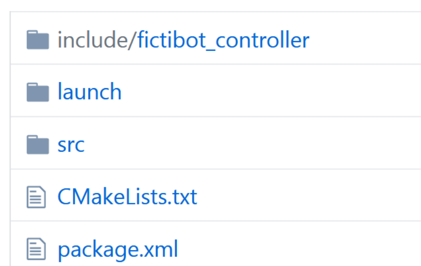


Figure 89: The `fictibot_controller` package tree.

```xml
<?xml version="1.0"?>
<package>
  <name>fictibot_controller</name>
  <version>0.1.0</version>
  <description>The fictibot_controller package</description>

  <maintainer email="andre.f.santos@inesctec.pt">Andre Santos</maintainer>
  <license>MIT</license>
  <author email="andre.f.santos@inesctec.pt">Andre Santos</author>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>fictibot_msgs</build_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>std_msgs</run_depend>
  <run_depend>fictibot_msgs</run_depend>
  <run_depend>fictibot_drivers</run_depend>
</package>
```

Listing A.9: The *package.xml* file.

```cmake
cmake_minimum_required(VERSION 2.8.3)
project(fictibot_controller)

find_package(catkin REQUIRED COMPONENTS roscpp std_msgs fictibot_msgs)
catkin_package(INCLUDE_DIRS include CATKIN_DEPENDS roscpp std_msgs fictibot_msgs)
include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(fictibot_controller
  src/controller_node.cpp
  src/random_controller.cpp
)

add_dependencies(fictibot_controller
  ${${PROJECT_NAME}_EXPORTED_TARGETS}
  ${catkin_EXPORTED_TARGETS}
)

target_link_libraries(fictibot_controller ${catkin_LIBRARIES})
```

Listing A.10: The *CMakeLists.txt* file.

```
#ifndef RANDOM_CONTROLLER_H_
#define RANDOM_CONTROLLER_H_

#include <ros/ros.h>
#include <std_msgs/Int8.h>
#include <fictibot_msgs/Custom.h>

class RandomController {
public:
  RandomController(ros::NodeHandle& n, double hz);
  ~RandomController(){};
  void spin();
private:
  bool stop_;
  bool laser_proximity_;
  bool bumper_left_pressed_;
  bool bumper_center_pressed_;
  bool bumper_right_pressed_;
  bool wheel_left_dropped_;
  bool wheel_right_dropped_;
  double stop_cycles_, stop_counter_;
  ros::Publisher stop_publisher_, command_publisher_;
  ros::Subscriber laser_subscriber_, bumper_subscriber_,
                  wheel_drop_subscriber_, custom_subscriber_;

  void laser_callback(const std_msgs::Int8::ConstPtr& msg);
  void bumper_callback(const std_msgs::Int8::ConstPtr& msg);
  void wheel_callback(const std_msgs::Int8::ConstPtr& msg);
  void custom_callback(const fictibot_msgs::Custom::ConstPtr& msg);
};

#endif /*RANDOM_CONTROLLER_H_*/
```

Listing A.11: The *include/fictibot_controller/random_controller.h* file.

```
 1 #define _USE_MATH_DEFINES
 2 
 3 #include <math.h>
 4 #include <cstdlib>
 5 #include <std_msgs/Empty.h>
 6 #include <std_msgs/Float64.h>
 7 #include "fictibot_controller/random_controller.h"
 8 
 9 RandomController::RandomController(ros::NodeHandle& n, double hz)
10   : stop_(false)
11   , laser_proximity_(false)
12   , bumper_left_pressed_(false)
13   , bumper_center_pressed_(false)
14   , bumper_right_pressed_(false)
15   , wheel_left_dropped_(false)
16   , wheel_right_dropped_(false)
17   , stop_counter_(0) {
18   stop_cycles_ = 2 * hz + 1;
19   uint32_t queue_size = (uint32_t) hz * 2 + 1;
20   std::string some_param;
21   n.param<std::string>("param", some_param, "nothing");
22   n.setParam("set_param", some_param);
23   command_publisher_ = n.advertise<std_msgs::Float64>("controller_cmd", 1);
24   stop_publisher_ = n.advertise<std_msgs::Empty>("/stop_cmd", 0);
25   laser_subscriber_ = n.subscribe("laser", queue_size,
26       &RandomController::laser_callback, this);
27   bumper_subscriber_ = n.subscribe("bumper", queue_size,
28       &RandomController::bumper_callback, this);
29   wheel_drop_subscriber_ = n.subscribe("wheel", queue_size,
30       &RandomController::wheel_callback, this);
31   if (some_param == "nothing") {
32     custom_subscriber_ = n.subscribe("custom_noparam", queue_size,
33         &RandomController::custom_callback, this);
34   } else {
35     custom_subscriber_ = n.subscribe("custom_w_param", queue_size,
36         &RandomController::custom_callback, this);
37   }
38 }
39 
40 void RandomController::laser_callback(const std_msgs::Int8::ConstPtr& msg) {
41   laser_proximity_ = msg->data <= 50;
42 }
43 
44 void RandomController::custom_callback(
45     const fictibot_msgs::Custom::ConstPtr& msg) {
46   ROS_INFO("Received custom message!");
47 }
```

Listing A.12: The *src/random_controller.cpp* file (part 1 of 2).

```
void RandomController::spin() {
  ros::spinOnce();
  bool prev_stop = stop_;
  stop_counter_--;
  stop_ = laser_proximity_ || bumper_left_pressed_ || bumper_center_pressed_
      || bumper_right_pressed_ || wheel_left_dropped_ || wheel_right_dropped_;
  if (!prev_stop && stop_) {
    std_msgs::Empty stop_msg;
    stop_publisher_.publish(stop_msg);
    stop_counter_ = stop_cycles_;
    std_msgs::Float64 vel_msg;
    vel_msg.data = (double) (std::rand() % 360 - 180) * M_PI / 180.0;
    command_publisher_.publish(vel_msg);
  }
  if (stop_ && stop_counter_ < 0) {
    stop_counter_ = stop_cycles_;
    std_msgs::Float64 vel_msg;
    vel_msg.data = (double) (std::rand() % 360 - 180) * M_PI / 180.0;
    command_publisher_.publish(vel_msg);
  }
}

void RandomController::bumper_callback(const std_msgs::Int8::ConstPtr& msg) {
  int left = msg->data & 4;
  int center = msg->data & 2;
  int right = msg->data & 1;
  if (left) { bumper_left_pressed_ = true; }
  else { bumper_left_pressed_ = false; }
  if (center) { bumper_center_pressed_ = true; }
  else { bumper_center_pressed_ = false; }
  if (right) { bumper_right_pressed_ = true; }
  else { bumper_right_pressed_ = false; }
}

void RandomController::wheel_callback(const std_msgs::Int8::ConstPtr& msg) {
  int left = msg->data & 12;
  int right = msg->data & 3;
  if (left == 3) { wheel_left_dropped_ = true; }
  else if (left == 2 && right >= 1) { wheel_left_dropped_ = true; }
  else if (left < 2) { wheel_left_dropped_ = false; }
  if (right == 3) { wheel_right_dropped_ = true; }
  else if (right == 2 && left >= 1) { wheel_right_dropped_ = true; }
  else if (right < 2) { wheel_right_dropped_ = false; }
}
```

Listing A.13: The *src/random_controller.cpp* file (part 2 of 2).

```cpp
#include <ros/ros.h>
#include "fictibot_controller/random_controller.h"

int main(int argc, char **argv) {
  ros::init(argc, argv, "fictibot_controller");
  ros::NodeHandle n;
  RandomController controller(n, 10 /*Hz*/);
  ros::Rate loop_rate(10 /*Hz*/);
  while (ros::ok()) {
    controller.spin();
    loop_rate.sleep();
  }
  return 0;
}
```

Listing A.14: The *src/controller_node.cpp* file.

```xml
<launch>
  <node name="fictibase" pkg="fictibot_drivers" type="fictibot_driver" />
  <node name="ficticontrol" pkg="fictibot_controller" type="fictibot_controller" />
</launch>
```

Listing A.15: The *launch/minimal.launch* file.

```xml
<launch>
  <node name="fictibase" pkg="fictibot_drivers" type="fictibot_driver" />
  <node name="fictiplex" pkg="fictibot_multiplex" type="fictibot_multiplex" />
  <node name="ficticontrol" pkg="fictibot_controller" type="fictibot_controller">
    <remap from="controller_cmd" to="normal_priority_cmd" />
    <remap from="/stop_cmd" to="normal_priority_stop" />
  </node>
</launch>
```

Listing A.16: The *launch/multiplexer.launch* file.

a.3 Fictibot Multiplexer Package

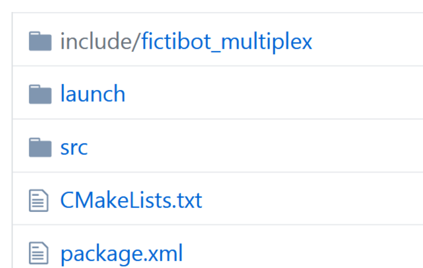


Figure 90: The `fictibot_multiplex` package tree.

```xml
<?xml version="1.0"?>
<package>
  <name>fictibot_multiplex</name>
  <version>0.1.0</version>
  <description>The fictibot_multiplex package</description>

  <maintainer email="andre.f.santos@inesctec.pt">Andre Santos</maintainer>
  <license>MIT</license>
  <author email="andre.f.santos@inesctec.pt">Andre Santos</author>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>std_msgs</run_depend>
  <run_depend>fictibot_drivers</run_depend>
</package>
```

Listing A.17: The *package.xml* file.

```cmake
cmake_minimum_required(VERSION 2.8.3)
project(fictibot_multiplex)

find_package(catkin REQUIRED COMPONENTS roscpp std_msgs)
catkin_package(INCLUDE_DIRS include CATKIN_DEPENDS roscpp std_msgs)
include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(fictibot_multiplex
  src/multiplex_node.cpp
  src/trichannel_multiplex.cpp
)

add_dependencies(fictibot_multiplex
  ${${PROJECT_NAME}_EXPORTED_TARGETS}
  ${catkin_EXPORTED_TARGETS}
)

target_link_libraries(fictibot_multiplex ${catkin_LIBRARIES})
```

Listing A.18: The *CMakeLists.txt* file.

```
1  #ifndef TRICHANNEL_MULTIPLEX_HPP_
2  #define TRICHANNEL_MULTIPLEX_HPP_
3
4  #include <ros/ros.h>
5  #include <std_msgs/Empty.h>
6  #include <std_msgs/Int8.h>
7  #include <std_msgs/Float64.h>
8
9  class TriChannelMultiplexer {
10 public:
11   TriChannelMultiplexer(ros::NodeHandle& n, double hz);
12   ~TriChannelMultiplexer(){};
13   void spin();
14 private:
15   int channel_, priority_cycles_, inactivity_counter_;
16   ros::Publisher stop_publisher_, command_publisher_, state_publisher_;
17   ros::Subscriber high_cmd_subscriber_, high_stop_subscriber_,
18                   normal_cmd_subscriber_, normal_stop_subscriber_,
19                   low_cmd_subscriber_, low_stop_subscriber_;
20
21   void high_cmd_callback(const std_msgs::Float64::ConstPtr& msg);
22   void high_stop_callback(const std_msgs::Empty::ConstPtr& msg);
23   void normal_cmd_callback(const std_msgs::Float64::ConstPtr& msg);
24   void normal_stop_callback(const std_msgs::Empty::ConstPtr& msg);
25   void low_cmd_callback(const std_msgs::Float64::ConstPtr& msg);
26   void low_stop_callback(const std_msgs::Empty::ConstPtr& msg);
27 };
28
29 #endif /*TRICHANNEL_MULTIPLEX_HPP_*/
```

Listing A.19: The *include/fictibot_multiplex/trichannel_multiplex.hpp* file.

```
#include <cstdlib>
#include "fictibot_multiplex/trichannel_multiplex.hpp"

#define LOW_PRIORITY -1
#define NORMAL_PRIORITY 0
#define HIGH_PRIORITY 1

TriChannelMultiplexer::TriChannelMultiplexer(ros::NodeHandle& n, double hz)
  : channel_(LOW_PRIORITY)
  , priority_cycles_(10)
  , inactivity_counter_(0) {
  uint32_t queue_size = (uint32_t) hz * 2 + 1;
  command_publisher_ = n.advertise<std_msgs::Float64>("controller_cmd", queue_size);
  stop_publisher_ = n.advertise<std_msgs::Empty>("stop_cmd", queue_size);
  state_publisher_ = n.advertise<std_msgs::Int8>("state", queue_size);
  high_cmd_subscriber_ = n.subscribe("high_priority_cmd", queue_size,
      &TriChannelMultiplexer::high_cmd_callback, this);
  high_stop_subscriber_ = n.subscribe("high_priority_stop", queue_size,
      &TriChannelMultiplexer::high_stop_callback, this);
  normal_cmd_subscriber_ = n.subscribe("normal_priority_cmd", queue_size,
      &TriChannelMultiplexer::normal_cmd_callback, this);
  normal_stop_subscriber_ = n.subscribe("normal_priority_stop", queue_size,
      &TriChannelMultiplexer::normal_stop_callback, this);
  low_cmd_subscriber_ = n.subscribe("low_priority_cmd", queue_size,
      &TriChannelMultiplexer::low_cmd_callback, this);
  low_stop_subscriber_ = n.subscribe("low_priority_stop", queue_size,
      &TriChannelMultiplexer::low_stop_callback, this);
  std_msgs::Int8 state_msg;
  state_msg.data = (int8_t) LOW_PRIORITY;
  state_publisher_.publish(state_msg);
}

void TriChannelMultiplexer::spin() {
  inactivity_counter_--;
  if (inactivity_counter_ < 0) {
    channel_ = LOW_PRIORITY;
    inactivity_counter_ = priority_cycles_;
    std_msgs::Int8 state_msg;
    state_msg.data = (int8_t) LOW_PRIORITY;
    state_publisher_.publish(state_msg);
  }
  ros::spinOnce();
}
```

Listing A.20: The *src/trichannel_multiplex.cpp* file (part 1 of 3).

```
void TriChannelMultiplexer::high_cmd_callback(const std_msgs::Float64::ConstPtr& msg)
    {
  command_publisher_.publish(msg);
  channel_ = HIGH_PRIORITY;
  inactivity_counter_ = priority_cycles_;
  std_msgs::Int8 state_msg;
  state_msg.data = (int8_t) HIGH_PRIORITY;
  state_publisher_.publish(state_msg);
}

void TriChannelMultiplexer::high_stop_callback(const std_msgs::Empty::ConstPtr& msg) {
  stop_publisher_.publish(msg);
  channel_ = HIGH_PRIORITY;
  inactivity_counter_ = priority_cycles_;
  std_msgs::Int8 state_msg;
  state_msg.data = (int8_t) HIGH_PRIORITY;
  state_publisher_.publish(state_msg);
}

void TriChannelMultiplexer::normal_cmd_callback(const std_msgs::Float64::ConstPtr&
    msg) {
  if (channel_ != HIGH_PRIORITY) {
    command_publisher_.publish(msg);
    channel_ = NORMAL_PRIORITY;
    inactivity_counter_ = priority_cycles_;
    std_msgs::Int8 state_msg;
    state_msg.data = (int8_t) NORMAL_PRIORITY;
    state_publisher_.publish(state_msg);
  }
}

void TriChannelMultiplexer::normal_stop_callback(const std_msgs::Empty::ConstPtr&
    msg) {
  if (channel_ != HIGH_PRIORITY) {
    stop_publisher_.publish(msg);
    channel_ = NORMAL_PRIORITY;
    inactivity_counter_ = priority_cycles_;
    std_msgs::Int8 state_msg;
    state_msg.data = (int8_t) NORMAL_PRIORITY;
    state_publisher_.publish(state_msg);
  }
}
```

Listing A.21: The *src/trichannel_multiplex.cpp* file (part 2 of 3).

```cpp

void TriChannelMultiplexer::low_cmd_callback(const std_msgs::Float64::ConstPtr& msg) {
  if (channel_ == LOW_PRIORITY) {
    command_publisher_.publish(msg);
    inactivity_counter_ = priority_cycles_;
  }
}

void TriChannelMultiplexer::low_stop_callback(const std_msgs::Empty::ConstPtr& msg) {
  if (channel_ == LOW_PRIORITY) {
    stop_publisher_.publish(msg);
    inactivity_counter_ = priority_cycles_;
  }
}
```

Listing A.22: The *src/trichannel_multiplex.cpp* file (part 3 of 3).

```cpp
#include <ros/ros.h>
#include "fictibot_multiplex/trichannel_multiplex.hpp"

int main(int argc, char **argv) {
  ros::init(argc, argv, "fictibot_multiplex");
  ros::NodeHandle n;
  TriChannelMultiplexer multiplexer(n, 10 /*Hz*/);
  ros::Rate loop_rate(10 /*Hz*/);
  while (ros::ok()) {
    multiplexer.spin();
    loop_rate.sleep();
  }
  return 0;
}
```

Listing A.23: The *src/multiplex_node.cpp* file.

a.4 Fictibot Messages Package

Figure 91: The `fictibot_msgs` package tree.

```xml
<?xml version="1.0"?>
<package>
  <name>fictibot_msgs</name>
  <version>0.1.0</version>
  <description>The fictibot_msgs package</description>

  <maintainer email="andre.f.santos@inesctec.pt">Andre Santos</maintainer>
  <license>MIT</license>
  <author email="andre.f.santos@inesctec.pt">Andre Santos</author>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>geometry_msgs</build_depend>
  <build_depend>message_generation</build_depend>
  <run_depend>message_runtime</run_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>rospy</run_depend>
  <run_depend>geometry_msgs</run_depend>
</package>
```

Listing A.24: The *package.xml* file.

```cmake
cmake_minimum_required(VERSION 2.8.3)
project(fictibot_msgs)

find_package(catkin REQUIRED COMPONENTS roscpp rospy geometry_msgs message_generation)
add_message_files(FILES Custom.msg)
generate_messages(DEPENDENCIES geometry_msgs)
catkin_package(CATKIN_DEPENDS roscpp rospy geometry_msgs message_runtime)
```

Listing A.25: The *CMakeLists.txt* file.

```
uint8 state
geometry_msgs/Pose2D pose
bool[3] bumpers
```

Listing A.26: The *msg/Custom.msg* file.

B

ROS METAMODEL SCHEMA

This appendix chapter contains the JSON schema, in YAML notation, for the various entities we define in the proposed metamodel for ROS applications (see Chapter 3). An up-to-date version can also be found online[1].

Common Definitions

```yaml
%YAML 1.1
---
$schema: "http://json-schema.org/draft-07/schema#"
$id: "https://github.com/git-afsantos/haros/tree/master/haros/schema/common.yaml"
title: Common Definitions
description: Common definitions for ROS entity specification.
type: object
additionalProperties: false
definitions:
    package_name:
        type: string
        pattern: "^[a-zA-Z][a-zA-Z0-9_]*$"
    file_name:
        # this is a relative path within the package
        type: string
        pattern: "^[^/]+(/[^/]+)*$"
    file_ref:
        # this is "<pkg>/<relative-path>"
        type: string
        pattern: "^[a-zA-Z0-9_]+/[^/]+(/[^/]+)*$"
    config_ref:
        type: string
        pattern: "^[a-zA-Z][a-zA-Z0-9_]*$"
    ros_type:
        # a "<pkg>/<type>" string
        type: string
```

1 https://github.com/git-afsantos/haros/tree/master/haros/schema

```yaml
27          pattern: "^[a-zA-Z][a-zA-Z0-9_]*/[^/]+$"
28      ros_name:
29          # global, relative and private names allowed
30          type: string
31          pattern: "^[/~]?[a-zA-Z][0-9a-zA-Z_]*(/[a-zA-Z][0-9a-zA-Z_]*)*$"
32      global_ros_name:
33          type: string
34          pattern: "^/[a-zA-Z][0-9a-zA-Z_]*(/[a-zA-Z][0-9a-zA-Z_]*)*$"
35      person:
36          type: object
37          properties:
38              name:
39                  type: string
40                  minLength: 1
41              email:
42                  oneOf:
43                      - type: string
44                        minLength: 3
45                      - type: null
46          required:
47              - name
48              - email
49      source_tree:
50          oneOf:
51              - type: object
52              - type: null
53      param_type:
54          type: string
55          enum:
56              - bool
57              - int
58              - double
59              - str
60              - yaml
61      base_location:
62          type: object
63          properties:
64              package:
65                  $ref: "#/definitions/package_name"
66              file:
67                  $ref: "#/definitions/file_name"
68              line:
69                  oneOf:
70                    - type: integer
71                      minimum: 1
72                    - type: null
73              column:
```

```
74                  oneOf:
75                    - type: integer
76                      minimum: 1
77                    - type: null
78          required:
79              - package
80              - file
81              - line
82              - column
83      source_location:
84          $ref: "#/definitions/base_location"
85      condition:
86          allOf:
87            - $ref: "#/definitions/base_location"
88            - properties:
89                  statement:
90                      type: string
91                      enum:
92                          - if
93                          - unless
94                          - switch
95                          - case
96                          - for
97                          - while
98                  expression:
99                      oneOf:
100                       - type: string
101                         minLength: 1
102                       - type: boolean
103             required:
104                 - statement
105                 - expression
106     control_path:
107         type: array
108         minItems: 1
109         items:
110             $ref: "#/definitions/condition"
111     control_flow_graph:
112         type: array
113         items:
114             $ref: "#/definitions/control_path"
115     advertise:
116         type: object
117         properties:
118             name:
119                 $ref: "#/definitions/ros_name"
120             type:
```

```
121                 $ref: "#/definitions/ros_type"
122             queue_size:
123                 type: integer
124                 minimum: 0
125             latched:
126                 type: boolean
127             traceability:
128                 $ref: "#/definitions/source_location"
129             conditions:
130                 $ref: "#/definitions/control_flow_graph"
131         required:
132             - name
133             - type
134             - queue_size
135             - traceability
136     subscribe:
137         type: object
138         properties:
139             name:
140                 $ref: "#/definitions/ros_name"
141             type:
142                 $ref: "#/definitions/ros_type"
143             queue_size:
144                 type: integer
145                 minimum: 0
146             traceability:
147                 $ref: "#/definitions/source_location"
148             conditions:
149                 $ref: "#/definitions/control_flow_graph"
150         required:
151             - name
152             - type
153             - queue_size
154             - traceability
155     advertise_service:
156         type: object
157         properties:
158             name:
159                 $ref: "#/definitions/ros_name"
160             type:
161                 $ref: "#/definitions/ros_type"
162             traceability:
163                 $ref: "#/definitions/source_location"
164             conditions:
165                 $ref: "#/definitions/control_flow_graph"
166         required:
167             - name
```

```yaml
168             - type
169             - traceability
170     service_client:
171         type: object
172         properties:
173             name:
174                 $ref: "#/definitions/ros_name"
175             type:
176                 $ref: "#/definitions/ros_type"
177             traceability:
178                 $ref: "#/definitions/source_location"
179             conditions:
180                 $ref: "#/definitions/control_flow_graph"
181         required:
182             - name
183             - type
184             - traceability
185     set_param:
186         type: object
187         properties:
188             name:
189                 $ref: "#/definitions/ros_name"
190             type:
191                 $ref: "#/definitions/param_type"
192             traceability:
193                 $ref: "#/definitions/source_location"
194             conditions:
195                 $ref: "#/definitions/control_flow_graph"
196         required:
197             - name
198             - type
199             - value
200             - traceability
201         allOf:
202           - if: { properties: { type: { const: bool }}}
203             then: { properties: { value: { type: [boolean, null] }}}
204           - if: { properties: { type: { const: int }}}
205             then: { properties: { value: { type: [integer, null] }}}
206           - if: { properties: { type: { const: double }}}
207             then: { properties: { value: { type: [number, null] }}}
208           - if: { properties: { type: { const: str }}}
209             then: { properties: { value: { type: [string, null] }}}
210           - if: { properties: { type: { const: yaml }}}
211             then: { properties: { value: { type: [object, array, null] }}}
212     get_param:
213         type: object
214         properties:
```

```
215             name:
216                 $ref: "#/definitions/ros_name"
217             type:
218                 $ref: "#/definitions/param_type"
219             traceability:
220                 $ref: "#/definitions/source_location"
221             conditions:
222                 $ref: "#/definitions/control_flow_graph"
223         required:
224             - name
225             - type
226             - traceability
227         allOf:
228           - if: { properties: { type: { const: bool }}}
229             then: { properties: { default_value: { type: [boolean, null] }}}
230           - if: { properties: { type: { const: int }}}
231             then: { properties: { default_value: { type: [integer, null] }}}
232           - if: { properties: { type: { const: double }}}
233             then: { properties: { default_value: { type: [number, null] }}}
234           - if: { properties: { type: { const: str }}}
235             then: { properties: { default_value: { type: [string, null] }}}
236           - if: { properties: { type: { const: yaml }}}
237             then: { properties: { default_value: { type: [object, array, null] }}}
238     ros_resource:
239         type: object
240         properties:
241             ros_name:
242                 description: The ROS name of this resource.
243                 $ref: "#/definitions/global_ros_name"
244             configuration:
245                 description: The configuration to which this resource belongs.
246                 $ref: "#/definitions/config_ref"
247         required:
248             - ros_name
249             - configuration
250     publisher_link:
251         allOf:
252           - $ref: "#/definitions/advertise"
253           - properties:
254                 name:
255                     $ref: "#/definitions/global_ros_name"
256                 original_name:
257                     $ref: "#/definitions/ros_name"
258     subscriber_link:
259         allOf:
260           - $ref: "#/definitions/subscribe"
261           - properties:
```

```
262             name:
263                 $ref: "#/definitions/global_ros_name"
264             original_name:
265                 $ref: "#/definitions/ros_name"
266 server_link:
267     allOf:
268       - $ref: "#/definitions/advertise_service"
269       - properties:
270             name:
271                 $ref: "#/definitions/global_ros_name"
272             original_name:
273                 $ref: "#/definitions/ros_name"
274 client_link:
275     allOf:
276       - $ref: "#/definitions/advertise_service"
277       - properties:
278             name:
279                 $ref: "#/definitions/global_ros_name"
280             original_name:
281                 $ref: "#/definitions/ros_name"
282 setter_link:
283     allOf:
284       - $ref: "#/definitions/set_param"
285       - properties:
286             name:
287                 $ref: "#/definitions/global_ros_name"
288             original_name:
289                 $ref: "#/definitions/ros_name"
290 getter_link:
291     allOf:
292       - $ref: "#/definitions/get_param"
293       - properties:
294             name:
295                 $ref: "#/definitions/global_ros_name"
296             original_name:
297                 $ref: "#/definitions/ros_name"
```

Source File

```yaml
%YAML 1.1
---
$schema: "http://json-schema.org/draft-07/schema#"
$id: "https://github.com/git-afsantos/haros/tree/master/haros/schema/source_file.yaml"
title: Source File
description: A Source File specification.
type: object
properties:
    name:
        description: The relative path of the file within the package.
        $ref: "common.yaml#/definitions/file_name"
    package:
        description: The name of the ROS package this file belongs to.
        $ref: "common.yaml#/definitions/package_name"
    language:
        description: |
            The programming, configuration or data serialisation
            language this file is written in.
        type: string
        enum:
            - cpp
            - python
            - launch
            - package
            - cmake
            - msg
            - srv
            - action
            - cfg
            - yaml
            - unknown
    source_tree:
        description: An Abstract Syntax Tree of the file.
        $ref: "common.yaml#/definitions/source_tree"
    dependencies:
        description: A set of dependencies on other files, e.g., included files.
        type: array
        uniqueItems: true
        items:
            $ref: "common.yaml#/definitions/file_ref"
required: [name, package, language]
```

Package

```yaml
%YAML 1.1
---
$schema: "http://json-schema.org/draft-07/schema#"
$id: "https://github.com/git-afsantos/haros/tree/master/haros/schema/package.yaml"
title: Package
description: A ROS Package specification.
type: object
properties:
    name:
        description: The name of the ROS package.
        $ref: "common.yaml#/definitions/package_name"
    authors:
        description: The set of authors of this package.
        type: array
        uniqueItems: true
        items:
            $ref: "common.yaml#/definitions/person"
    maintainers:
        description: The set of maintainers of this package.
        type: array
        uniqueItems: true
        items:
            $ref: "common.yaml#/definitions/person"
    version:
        description: The version number of this package.
        type: string
        minLength: 1
    path:
        description: The file system path pointing to the root of this package.
        type: string
        minLength: 1
    is_metapackage:
        description: Whether this is a metapackage.
        type: boolean
    dependencies:
        description: A set of dependencies on other packages.
        type: array
        uniqueItems: true
        items:
            $ref: "common.yaml#/definitions/package_name"
    project:
        description: The name of the project this package belongs to.
        type: string
    repository:
```

```yaml
            description: The name of the repository this package belongs to.
            type: string
        files:
            description: A set of files that this package includes.
            type: array
            uniqueItems: true
            minItems: 1
            items:
                $ref: "common.yaml#/definitions/file_name"
        nodes:
            description: |
                A set of ROS nodes built from this package.
                The set contains only the executable name,
                since the package is always the same.
                E.g., the 'move_base' package builds the 'move_base' node.
            type: array
            uniqueItems: true
            items:
                type: string
                minLength: 1
required:
    - name
    - version
    - path
    - is_metapackage
    - project
    - files
```

Repository

```yaml
%YAML 1.1
---
$schema: "http://json-schema.org/draft-07/schema#"
$id: "https://github.com/git-afsantos/haros/tree/master/haros/schema/repository.yaml"
title: Repository
description: A source code repository specification.
type: object
properties:
    name:
        description: The name of the repository.
        type: string
        minLength: 1
    vcs:
        description: The version control system used for this repository.
        type: string
        enum:
            - git
            - svn
            - hg
    path:
        description: The file system path pointing to the root of this repository.
        type: string
        minLength: 1
    version:
        description: The version of the repository (e.g., 'master' branch).
        type: string
        minLength: 1
    packages:
        description: A set of ROS packages that this repository includes.
        type: array
        uniqueItems: true
        minItems: 1
        items:
            $ref: "common.yaml#/definitions/package_name"
required: [name, vcs, version, path, project, packages]
```

Node


```yaml
%YAML 1.1
---
$schema: "http://json-schema.org/draft-07/schema#"
$id: "https://github.com/git-afsantos/haros/tree/master/haros/schema/node.yaml"
title: Node
description: A ROS Node specification.
type: object
properties:
    name:
        description: The name of the ROS node.
        type: string
        minLength: 1
    package:
        description: The name of the ROS package this node belongs to.
        $ref: "common.yaml#/definitions/package_name"
    files:
        description: A set of files that build this node.
        type: array
        uniqueItems: true
        minItems: 1
        items:
            $ref: "common.yaml#/definitions/file_name"
    source_tree:
        description: An Abstract Syntax Tree of the node.
        $ref: "common.yaml#/definitions/source_tree"
    is_nodelet:
        description: Whether this is a nodelet.
        type: boolean
    properties:
        description: A list of behavioural properties that this node satisfies.
        type: array
        items:
            type: string
    ros_name:
        description: The default ROS name of the node in runtime.
        $ref: "common.yaml#/definitions/ros_name"
    instances:
        description: The set of runtime instances of this node.
        type: array
        uniqueItems: true
        items:
            type: string
    advertise:
        description: A list of publishers created by this node.
```

```
45          type: array
46          items:
47              $ref: "common.yaml#/definitions/advertise"
48      subscribe:
49          description: A list of subscribers created by this node.
50          type: array
51          items:
52              $ref: "common.yaml#/definitions/subscribe"
53      service_client:
54          description: A list of service clients created by this node.
55          type: array
56          items:
57              $ref: "common.yaml#/definitions/service_client"
58      advertise_service:
59          description: A list of service servers created by this node.
60          type: array
61          items:
62              $ref: "common.yaml#/definitions/advertise_service"
63      set_param:
64          description: A list of parameter setters created by this node.
65          type: array
66          items:
67              $ref: "common.yaml#/definitions/set_param"
68      get_param:
69          description: A list of parameter getters created by this node.
70          type: array
71          items:
72              $ref: "common.yaml#/definitions/get_param"
73  required:
74      - name
75      - package
76      - files
```

Project

```yaml
%YAML 1.1
---
$schema: "http://json-schema.org/draft-07/schema#"
$id: "https://github.com/git-afsantos/haros/tree/master/haros/schema/project.yaml"
title: Project
description: A ROS Project specification.
type: object
properties:
    name:
        description: The name of the ROS project.
        type: string
        minLength: 1
    packages:
        description: The set of packages that are part of this project.
        type: array
        uniqueItems: true
        minItems: 1
        items:
            $ref: "common.yaml#/definitions/package_name"
    configurations:
        description: The set of configurations that are part of this project.
        type: array
        uniqueItems: true
        items:
            $ref: "common.yaml#/definitions/config_ref"
required:
    - name
    - packages
```

Node Instance

```yaml
%YAML 1.1
---
$schema: "http://json-schema.org/draft-07/schema#"
$id:
    "https://github.com/git-afsantos/haros/tree/master/haros/schema/node_instance.yaml"
title: Node Instance
description: A ROS Node Instance specification.
allOf:
  - $ref: "common.yaml#/definitions/ros_resource"
  - properties:
        node_type:
            description: The executable from which this node is spawned.
            $ref: "common.yaml#/definitions/ros_type"
        args:
            description: Provided command-line arguments.
            type: string
        remaps:
            description: Provided name remappings.
            type: object
            patternProperties:
                # global ROS name
                "^/[a-zA-Z][0-9a-zA-Z_]*(/[a-zA-Z][0-9a-zA-Z_]*)*$":
                    $ref: "common.yaml#/definitions/global_ros_name"
        traceability:
            description: |
                Location where the `<node>` tag begins in the
                respective launch file.
                If spawned via `rosrun`, set to `null`.
            oneOf:
                - $ref: "common.yaml#/definitions/source_location"
                - type: null
        conditions:
            description: Unresolved launch file conditions that affect this node.
            $ref: "common.yaml#/definitions/control_flow_graph"
        publishers:
            description: A list of publishers created by this node.
            type: array
            items:
                $ref: "common.yaml#/definitions/publisher_link"
        subscribers:
            description: A list of subscribers created by this node.
            type: array
            items:
                $ref: "common.yaml#/definitions/subscriber_link"
```

```
44        clients:
45            description: A list of service clients created by this node.
46            type: array
47            items:
48                $ref: "common.yaml#/definitions/client_link"
49        servers:
50            description: A list of service servers created by this node.
51            type: array
52            items:
53                $ref: "common.yaml#/definitions/server_link"
54        setters:
55            description: A list of parameter setters created by this node.
56            type: array
57            items:
58                $ref: "common.yaml#/definitions/setter_link"
59        getters:
60            description: A list of parameter getters created by this node.
61            type: array
62            items:
63                $ref: "common.yaml#/definitions/getter_link"
64    required:
65        - node_type
66        - traceability
67  - if: { properties: { traceability: { const: null }}}
68    then: { properties: { conditions: { const: [] }}}
```

Topic

```
1  %YAML 1.1
2  ---
3  $schema: "http://json-schema.org/draft-07/schema#"
4  $id: "https://github.com/git-afsantos/haros/tree/master/haros/schema/topic.yaml"
5  title: Topic
6  description: A ROS Topic specification.
7  allOf:
8    - $ref: "common.yaml#/definitions/ros_resource"
9    - properties:
10         publishers:
11             description: A list of publishers connected to this topic.
12             type: array
13             items:
14                 $ref: "common.yaml#/definitions/publisher_link"
15         subscribers:
16             description: A list of subscribers connected to this topic.
17             type: array
18             items:
19                 $ref: "common.yaml#/definitions/subscriber_link"
20     required:
21         - publishers
22         - subscribers
```

Service

```yaml
%YAML 1.1
---
$schema: "http://json-schema.org/draft-07/schema#"
$id: "https://github.com/git-afsantos/haros/tree/master/haros/schema/service.yaml"
title: Service
description: A ROS Service specification.
allOf:
  - $ref: "common.yaml#/definitions/ros_resource"
  - properties:
        clients:
            description: A list of clients connected to this service.
            type: array
            items:
                $ref: "common.yaml#/definitions/client_link"
        server:
            description: The server that provides this service.
            oneOf:
              - $ref: "common.yaml#/definitions/server_link"
              - type: null
    required:
        - clients
        - server
```

Parameter

```yaml
%YAML 1.1
---
$schema: "http://json-schema.org/draft-07/schema#"
$id: "https://github.com/git-afsantos/haros/tree/master/haros/schema/parameter.yaml"
title: Parameter
description: A ROS Parameter specification.
allOf:
  - $ref: "common.yaml#/definitions/ros_resource"
  - properties:
        type:
            description: The default type for this parameter.
            $ref: "common.yaml#/definitions/param_type"
        default_value:
            description: The default value for this parameter, set at launch.
        traceability:
            description: "Location where the `<param>` or `<rosparam>` tag begins
                          in the launch file; `null` if set via command-line."
            oneOf:
                - $ref: "common.yaml#/definitions/source_location"
                - type: null
        conditions:
            description: Unresolved launch file conditions that affect this parameter.
            $ref: "common.yaml#/definitions/control_flow_graph"
        setters:
            description: A list of parameter writers.
            type: array
            items: { $ref: "common.yaml#/definitions/setter_link" }
        getters:
            description: A list of parameter readers.
            type: array
            items: { $ref: "common.yaml#/definitions/getter_link" }
    required: [type, traceability]
  - if: { properties: { type: { const: bool }}}
    then: { properties: { default_value: { type: [boolean, null] }}}
  - if: { properties: { type: { const: int }}}
    then: { properties: { default_value: { type: [integer, null] }}}
  - if: { properties: { type: { const: double }}}
    then: { properties: { default_value: { type: [number, null] }}}
  - if: { properties: { type: { const: str }}}
    then: { properties: { default_value: { type: [string, null] }}}
  - if: { properties: { type: { const: yaml }}}
    then: { properties: { default_value: { type: [object, array, null] }}}
```

Configuration


```yaml
%YAML 1.1
---
$schema: "http://json-schema.org/draft-07/schema#"
$id:
    "https://github.com/git-afsantos/haros/tree/master/haros/schema/configuration.yaml"
title: Configuration
description: A ROS Configuration specification.
type: object
properties:
    uid:
        description: A unique identifier for the configuration.
        $ref: "common.yaml#/definitions/config_ref"
    name:
        description: |
            A human-readable name for the configuration.
            Defaults to the `uid`.
        type: string
        minLength: 1
    project:
        description: The name of the project this configuration belongs to.
        type: string
    nodes:
        description: A list of node instances that are part of this configuration.
        type: array
        items:
            $ref: "node_instance.yaml#"
    parameters:
        description: A list of parameters that are part of this configuration.
        type: array
        items:
            $ref: "parameter.yaml#"
    launch_commands:
        description: A list of launch commands required to start this configuration.
        type: array
        minItems: 1
        items:
            $ref: "#/definitions/launch_command"
    properties:
        description: A list of behavioural properties that this configuration
    satisfies.
        type: array
        items:
            type: string
    environment:
```

```
43          description: A mapping of necessary environment variables for this
    configuration.
44          type: object
45          additionalProperties:
46              type: string
47      user_attributes:
48          description: A mapping of user-defined attributes.
49          type: object
50  required:
51      - uid
52      - project
53      - nodes
54      - parameters
55      - launch_files
56  definitions:
57      launch_command:
58          type: object
59          properties:
60              command:
61                  type: string
62                  enum: [roslaunch, rosrun]
63              args:
64                  type: array
65          required:
66              - command
67              - args
```

C

PYFLWOR PLUG-IN'S SOURCE CODE

This appendix chapter contains the full source code of example plug-in based on the Pyflwor query engine, so as to provide a reference version that will not change over time and to make the document self-contained. An up-to-date version can also be found online[1].

Listings C.1, C.2, C.3 and C.4 contain the source code for the plug-in itself. Listings C.5 and C.6 contain the HAROS project file used as input.

```
1  %YAML 1.1
2  ---
3  name: haros_plugin_pyflwor
4  version: 0.1.0
5  rules:
6      query:
7          name: User-defined Query
8          description: "A user-defined query found a match."
9          tags:
10             - code-standards
11             - query
12             - pyflwor
13             - computation-graph
```

Listing C.1: The *plugin.yaml* file.

1 https://github.com/git-afsantos/haros-plugin-pyflwor

```
1  import logging
2  import os
3  from haros.metamodel import Resource, RosPrimitive
4  from .pyflwor_monkey_patch import make_parser
5
6  log = logging.getLogger(__name__)
7
8  def configuration_analysis(iface, config):
9      query_data = config.user_attributes.get("haros_plugin_pyflwor", ())
10     if query_data:
11         pyflwor = _setup()
12         for query_datum in query_data:
13             name = query_datum["name"]
14             details = query_datum.get("details", None)
15             query = query_datum["query"]
16             matches = pyflwor.execute(query, config, name=name)
17             _report(iface, matches, name=name, details=details)
18
19 class QueryEngine(object):
20     QUERY_CONTEXT = {
21         "True": True, "False": False, "None": None, "abs": abs, "bool": bool,
22         "cmp": cmp, "divmod": divmod, "float": float, "int": int,
23         "isinstance": isinstance, "len": len, "long": long, "max": max,
24         "min": min, "pow": pow, "sum": sum, "round": round
25     }
26
27     def __init__(self, pyflwor):
28         self.pyflwor = pyflwor
29         self.data = dict(self.QUERY_CONTEXT)
30         self.data["is_rosglobal"] = QueryEngine.is_rosglobal
31
32     def execute(self, query, config, name=None):
33         self.data["config"] = config
34         self.data["nodes"] = config.nodes
35         self.data["topics"] = config.topics
36         self.data["services"] = config.services
37         self.data["parameters"] = config.parameters
38         location = config.location
39         try:
40             return self.pyflwor(query, self.data)
41         except SyntaxError as e:
42             if name:
43                 log.error("SyntaxError on query (%s): %s", name, e)
44             else:
45                 log.error("SyntaxError on unnamed query: %s", e)
46         return ()
47
48     @staticmethod
49     def is_rosglobal(name):
50         return name and name.startswith("/")
51
52 def _setup():
53     pyflwor_dir = os.path.join(os.getcwd(), "pyflwor")
54     os.mkdir(pyflwor_dir)
55     pyflwor = make_parser(pyflwor_dir)
56     query_engine = QueryEngine(pyflwor)
57     return query_engine
```

Listing C.2: The *plugin.py* file (part 1).

```python
RUNTIME_ENTITY = (Resource, RosPrimitive)
def _report(iface, matches, name="<unnamed>", details=None):
    # `matches` can be of types:
    # - pyflwor.OrderedSet.OrderedSet<object> for Path queries
    # - tuple<object> for FLWR queries single return
    # - tuple<tuple<object>> for FLWR queries multi return
    # - tuple<dict<str, object>> for FLWR queries named return
    n = 0
    for match in matches:
        value = None
        runtime_entities = []
        if isinstance(match, tuple):
            # assume tuple<tuple<object>> for FLWR queries multi return
            if len(match) == 1 and isinstance(match[0], tuple):
                match = match[0]
            for item in match:
                if isinstance(item, RUNTIME_ENTITY):
                    n += 1
                    runtime_entities.append(item)
        elif isinstance(match, dict):
            # assume tuple<dict<str, object>> for FLWR queries named return
            for key, item in match.items():
                if isinstance(item, RUNTIME_ENTITY):
                    n += 1
                    runtime_entities.append(item)
        elif isinstance(match, RUNTIME_ENTITY):
            n = 1
            runtime_entities.append(match)
        if not n:
            # literals and other return values
            n = 1
            value = match
        if details:
            msg = details.format(entities=runtime_entities, value=value, n=n)
        else:
            msg = "Query {} found {} matches:\n{}".format(
                name, n, runtime_entities)
        log.info("Query <%s> found %d matches.", name, n)
        iface.report_runtime_violation("query", msg, resources=runtime_entities)
```

Listing C.3: The *plugin.py* file (part 2).

```python
from __future__ import unicode_literals
from builtins import str, bytes
import sys
from pyflwor.parser import Parser
from pyflwor.lexer import Lexer
from ply import lex, yacc

class MonkeyPatchLexer(Lexer):
    def __new__(cls, pyflwor_dir, **kwargs):
        self = super(Lexer, cls).__new__(cls, **kwargs)
        self.lexer = lex.lex(object=self, debug=False, optimize=True,
                             outputdir=pyflwor_dir, **kwargs)
        return self.lexer

class MonkeyPatchParser(Parser):
    def __new__(cls, pyflwor_dir, **kwargs):
        self = super(Parser, cls).__new__(cls, **kwargs)
        self.names = dict()
        self.yacc = yacc.yacc(module=self, debug=False,
                              optimize=True, write_tables=False,
                              outputdir=pyflwor_dir, **kwargs)
        return self.yacc

def make_parser(pyflwor_dir):
    if pyflwor_dir not in sys.path:
        sys.path.insert(0, pyflwor_dir)
    def execute(query, namespace):
        lexer = MonkeyPatchLexer(pyflwor_dir)
        parser = MonkeyPatchParser(pyflwor_dir)
        qbytes = bytes(query, "utf-8").decode("unicode_escape")
        qfunction = parser.parse(qbytes, lexer=lexer)
        return qfunction(namespace)
    return execute
```

Listing C.4: The *pyflwor_monkey_patch.py* file.

```yaml
%YAML 1.1
---
project: haros_tutorials
packages:
    - fictibot_drivers
    - fictibot_controller
    - fictibot_multiplex
    - minimal_example
configurations:
    missing_remap:
        launch:
            - fictibot_controller/launch/missing_remap.launch
    type_check:
        launch:
            - fictibot_controller/launch/type_check.launch
    multiplex:
        launch:
            - fictibot_controller/launch/multiplexer.launch
        plugin_data:
            haros_plugin_pyflwor:
                - name: Query 1 - No Global ROS Names
                  details: "Found {n} Resources using global names - {entities}"
                  query: "topics/publishers[self.rosname.is_global] |
                          topics/subscribers[self.rosname.is_global] |
                          services/servers[self.rosname.is_global] |
                          services/clients[self.rosname.is_global] |
                          parameters/reads[self.rosname.is_global] |
                          parameters/writes[self.rosname.is_global]"
                - name: Query 2 - No Conditional Communications
                  details: "Found {entities} under a conditional statement."
                  query: "topics/publishers[len(self.conditions) > 0] |
                          topics/subscribers[len(self.conditions) > 0] |
                          services/servers[len(self.conditions) > 0] |
                          services/clients[len(self.conditions) > 0]"
                - name: Query 3 - Message Types Must Match
                  details: "Found two participants on the same topic with
                            mismatching message types - {entities}"
                  query: "for p in <nodes/publishers | nodes/subscribers>,
                              q in <nodes/publishers | nodes/subscribers>
                          where p.topic_name == q.topic_name and p.type != q.type
                          return p, q"
                - name: Query 4 - No Unbounded Queues
                  details: "Found {n} topics with infinite queues - {entities}"
                  query: "topics/publishers[self.queue_size == 0] |
                          topics/subscribers[self.queue_size == 0]"
                - name: Query 5 - Queues of Size 1
                  details: "Found {n} topics with queues of size 1 - {entities}"
                  query: "topics/publishers[self.queue_size == 1] |
                          topics/subscribers[self.queue_size == 1]"
                - name: Query 6 - One Publisher Per Topic
                  details: "Found {n} publishers on a single topic - {entities}"
                  query: "topics[len(self.publishers) > 1]"
```

Listing C.5: Project file for Fictibot with Pyflwor queries (part 1).

```yaml
- name: Query 7 - No Disconnected Similar Topics
  details: "Found two topics with similar names and type,
            but they are not connected. It could be a missing
            remapping. {entities}"
  query: "for pub in <c/nodes/publishers[self.topic.is_disconnected]>,
              sub in <c/nodes/subscribers[self.topic.is_disconnected]>
          where pub.type == sub.type and
            (pub.topic.id.endswith(sub.topic.name)
              or sub.topic.id.endswith(pub.topic.name)
              or pub.rosname.full == sub.rosname.full
              or pub.rosname.full.endswith(sub.rosname.own)
              or sub.rosname.full.endswith(pub.rosname.own)
              or pub.rosname.given == sub.rosname.given)
          return 'pub':pub, 'sub':sub"
- name: Query 8 - Uses of std_msgs/Empty
  details: "Found {n} topics of type std_msgs/Empty - {entities}"
  query: "topics[self.type == 'std_msgs/Empty']"
```

Listing C.6: Project file for Fictibot with Pyflwor queries (part 2).

D

CATALOGUE OF PROPERTIES FOR TURTLEBOT2 AND AGROB V16

This appendix lists all the properties for the TurtleBot2 and the AgRob V16 systems that were considered for the evaluation of our Property-based Testing approach. The property specification language is defined in Chapter 4. The Property-based Testing approach is presented in Chapter 7. The results of the evaluation are presented in Chapter 8, Section 8.4.

d.1 TurtleBot2 Safety Controller Node

d.1.1 *Axioms*

```
1 globally: no /kobuki_safety_controller/reset
2 globally: no /kobuki_safety_controller/enable
3 globally: no /kobuki_safety_controller/disable
4 globally: no /mobile_base/events/bumper { not state in {PRESSED, RELEASED} }
5 globally: no /mobile_base/events/bumper { not bumper in {LEFT, CENTER, RIGHT} }
6 globally: /mobile_base/events/bumper forbids /mobile_base/events/bumper within 100 ms
7 globally: no /mobile_base/events/cliff { not state in {CLIFF, FLOOR} }
8 globally: no /mobile_base/events/cliff { not sensor in {LEFT, CENTER, RIGHT} }
9 globally: /mobile_base/events/cliff forbids /mobile_base/events/cliff within 100 ms
10 globally: no /mobile_base/events/wheel_drop { not state in {DROPPED, RAISED} }
11 globally: no /mobile_base/events/wheel_drop { not wheel in {LEFT, RIGHT} }
12 globally: /mobile_base/events/wheel_drop forbids /mobile_base/events/wheel_drop
       within 100 ms
```

d.1.2 *Properties*

```
1 globally: no /cmd_vel_mux/safety_controller { linear.x > 0.0 }
2 after /mobile_base/events/wheel_drop { state = DROPPED and wheel = LEFT } until
       /mobile_base/events/wheel_drop { state = RAISED and wheel = LEFT }: no
       /cmd_vel_mux/safety_controller { not linear.x = 0.0 }
3 globally: /cmd_vel_mux/safety_controller { linear.x = 0.0 } requires
       /mobile_base/events/wheel_drop { state = DROPPED }
```

```
4 globally: /mobile_base/events/wheel_drop { state = DROPPED } causes
     /cmd_vel_mux/safety_controller { linear.x = 0.0 and angular.z = 0.0 } within 100
     ms
5 until /mobile_base/events/wheel_drop { state = DROPPED }: /mobile_base/events/bumper
     { state = PRESSED } causes /cmd_vel_mux/safety_controller { linear.x < 0.0 }
     within 100 ms
6 globally: /mobile_base/events/wheel_drop { state = DROPPED } forbids
     /cmd_vel_mux/safety_controller { not linear.x = 0.0 } within 100 ms
```

d.2 TurtleBot2 Random Walker Controller Node

d.2.1 *Axioms*

```
1 globally: no /kobuki_random_walker_controller/enable
2 globally: no /kobuki_random_walker_controller/disable
3 globally: no /mobile_base/events/bumper { not state in {PRESSED, RELEASED} }
4 globally: no /mobile_base/events/bumper { not bumper in {LEFT, CENTER, RIGHT} }
5 globally: /mobile_base/events/bumper forbids /mobile_base/events/bumper within 100 ms
6 globally: no /mobile_base/events/cliff { not state in {CLIFF, FLOOR} }
7 globally: no /mobile_base/events/cliff { not sensor in {LEFT, CENTER, RIGHT} }
8 globally: /mobile_base/events/cliff forbids /mobile_base/events/cliff within 100 ms
9 globally: no /mobile_base/events/wheel_drop { not state in {DROPPED, RAISED} }
10 globally: no /mobile_base/events/wheel_drop { not wheel in {LEFT, RIGHT} }
11 globally: /mobile_base/events/wheel_drop forbids /mobile_base/events/wheel_drop
     within 100 ms
```

d.2.2 *Properties*

```
1 globally: no /cmd_vel_mux/random_walker { linear.x > 0.1 }
2 after /mobile_base/events/wheel_drop { state = DROPPED and wheel = LEFT } until
     /mobile_base/events/wheel_drop { state = RAISED and wheel = LEFT }: no
     /cmd_vel_mux/random_walker { not linear.x = 0.0 }
3 until /mobile_base/events/wheel_drop { state = DROPPED }: /mobile_base/events/bumper
     { state = PRESSED } causes /mobile_base/commands/led1 { value = ORANGE } within
     100 ms
4 globally: some /cmd_vel_mux/random_walker within 500 ms
```

d.3 TurtleBot2 Multiplexer Node

d.3.1 *Axioms*

```
1 globally: /cmd_vel_mux/safety_controller forbids /cmd_vel_mux/safety_controller
      within 100 ms
2 globally: /cmd_vel_mux/random_walker forbids /cmd_vel_mux/random_walker within 100 ms
```

d.3.2 *Properties*

```
1 globally: /cmd_vel_mux/safety_controller as A causes /mobile_base/commands/velocity {
      linear.x = @A.linear.x and angular.z = @A.angular.z } within 100 ms
2 until /cmd_vel_mux/safety_controller: /cmd_vel_mux/random_walker as A causes
      /mobile_base/commands/velocity { linear.x = @A.linear.x and angular.z =
      @A.angular.z } within 100 ms
```

d.4 TurtleBot2 Random Walker Configuration

d.4.1 *Axioms*

```
1 globally: no /kobuki_safety_controller/reset
2 globally: no /kobuki_safety_controller/enable
3 globally: no /kobuki_safety_controller/disable
4 globally: no /kobuki_random_walker_controller/enable
5 globally: no /kobuki_random_walker_controller/disable
6 globally: no /mobile_base/events/bumper {not state in {PRESSED, RELEASED}}
7 globally: no /mobile_base/events/bumper {not bumper in {LEFT, CENTER, RIGHT}}
8 globally: /mobile_base/events/bumper forbids /mobile_base/events/bumper within 100 ms
9 globally: no /mobile_base/events/cliff {not state in {CLIFF, FLOOR}}
10 globally: no /mobile_base/events/cliff {not sensor in {LEFT, CENTER, RIGHT}}
11 globally: /mobile_base/events/cliff forbids /mobile_base/events/cliff within 100 ms
12 globally: no /mobile_base/events/wheel_drop {not state in {DROPPED, RAISED}}
13 globally: no /mobile_base/events/wheel_drop {not wheel in {LEFT, RIGHT}}
14 globally: /mobile_base/events/wheel_drop forbids /mobile_base/events/wheel_drop
      within 100 ms
```

d.4.2 *Properties*

```
1 globally: no /mobile_base/commands/velocity { linear.x < -0.1 }
```

```
2 globally: /mobile_base/commands/velocity { linear.x = 0.0 } requires
      /mobile_base/events/wheel_drop { state = DROPPED }
3 after /mobile_base/events/wheel_drop { state = DROPPED and wheel = LEFT } until
      /mobile_base/events/wheel_drop { state = RAISED and wheel = LEFT }: no
      /mobile_base/commands/velocity { not linear.x = 0.0 }
4 globally: /mobile_base/events/wheel_drop { state = DROPPED } causes
      /mobile_base/commands/velocity { linear.x = 0.0 and angular.z = 0.0 } within 100
      ms
5 globally: some /mobile_base/commands/velocity within 500 ms
```

d.5 AgRob V16 Safety Controller Node

d.5.1 *Axioms*

```
1 globally: no /scan {not range_min = 0.01}
2 globally: no /scan {not range_max = 20.0}
3 globally: no /scan {not scan_time = 0.02}
4 globally: no /scan {not angle_min = -2.35619}
5 globally: no /scan {not angle_max = 2.35619}
6 globally: no /scan {not angle_increment = 0.785398}
7 globally: no /scan {not time_increment = 0.0028571428571429}
8 globally: no /scan {not len(ranges) = 7}
9 globally: no /scan {not len(intensities) = 7}
10 globally: no /scan {not ranges[3] in [0.01 to 20.0]}
11 globally: no /lidar1/scan {not range_min = 0.01}
12 globally: no /lidar1/scan {not range_max = 20.0}
13 globally: no /lidar1/scan {not scan_time = 0.02}
14 globally: no /lidar1/scan {not angle_min = -2.35619}
15 globally: no /lidar1/scan {not angle_max = 2.35619}
16 globally: no /lidar1/scan {not angle_increment = 0.785398}
17 globally: no /lidar1/scan {not time_increment = 0.0028571428571429}
18 globally: no /lidar1/scan {not len(ranges) = 7}
19 globally: no /lidar1/scan {not len(intensities) = 7}
20 globally: no /lidar1/scan {not ranges[3] in [0.01 to 20.0]}
21 globally: no /darknet_ros/bounding_boxes {not len(bounding_boxes) = 2}
22 globally: no /darknet_ros/bounding_boxes {not bounding_boxes[0].Class in {"person",
      "other"}}
23 globally: no /darknet_ros/bounding_boxes {not bounding_boxes[1].Class in {"other",
      "person"}}
24 globally: no /imu_um7/rpy {not vector.x in {0, 0.2181662, 0.436333}}
25 globally: no /imu_um7/rpy {not vector.y in {0, 0.1309, 0.2618}}
26 globally: no /imu_um7/rpy {not vector.z = 0.0}
```

d.5.2 *Properties*

```
1 globally: no /agrob/safe_mode {not safe_modes in {"000", "002", "003", "009", "010",
     "012", "013", "019", "090", "092", "093", "099", "100", "102", "103", "109",
     "110", "112", "113", "119", "190", "192", "193", "199"}}
2 until /imu_um7/rpy {vector.x < 0.436333 and vector.y < 0.2618}: /imu_um7/rpy forbids
     /agrob/safe_mode {not safe_modes in {"090", "092", "093", "099", "190", "192",
     "193", "199"}}
3 until /imu_um7/rpy: no /agrob/safe_mode {safe_modes in {"090", "092", "093", "099",
     "190", "192", "193", "199"}}
4 globally: some /agrob/safe_mode {safe_modes = "000"} within 3000 ms
5 after /imu_um7/rpy {not vector.x < 0.2181662 and not vector.y < 0.1309}:
     /agrob/safe_mode {safe_modes = "000"} requires /imu_um7/rpy {vector.x = 0 and
     vector.y = 0}
```

d.6 AgRob V16 Supervisor Node

d.6.1 *Axioms*

```
1 globally: no /astar_gps_goal
2 globally: no /agrob/target
3 globally: no /agrob/cmd_auto
4 globally: no /fix
5 globally: no /agrob_map/gps/filtered
6 globally: no /agrob_map/imu/rpy
7 globally: no /agrob/agrob_mode {not mode in {0, 1, 2, 3}}
8 globally: no /agrob/safe_mode {not safe_mode = 0}
9 globally: no /agrob/safe_mode {not safe_modes in {"000", "002", "003", "009", "010",
     "012", "013", "019", "090", "092", "093", "099", "100", "102", "103", "109",
     "110", "112", "113", "119", "190", "192", "193", "199"}}
10 globally: /agrob/safe_mode forbids /agrob/safe_mode within 25 ms
```

d.6.2 *Properties*

```
1 globally: no /husky_velocity_controller/cmd_vel {not linear.x in [-0.25 to 0.5]}
2 globally: no /husky_velocity_controller/cmd_vel {not angular.z in [-0.5 to 0.5]}
3 after /agrob/safe_mode {safe_modes in {"003", "009", "013", "019", "090", "092",
     "093", "099", "100", "102", "103", "109", "110", "112", "113", "119", "190",
     "192", "193", "199"}} until /agrob/safe_mode {safe_modes in {"000", "002", "010",
     "012"}}: no /husky_velocity_controller/cmd_vel {linear.x > 0.0}
4 after /agrob/safe_mode {safe_modes in {"003", "009", "013", "019", "090", "092",
     "093", "099", "100", "102", "103", "109", "110", "112", "113", "119", "190",
```

```
"192", "193", "199"}} until /agrob/safe_mode {safe_modes in {"000", "002", "010",
"012"}}: some /husky_velocity_controller/cmd_vel {not linear.x > 0.0} within 50 ms
```

d.7 AgRob V16 Joystick Controller Node

d.7.1 *Axioms*

```
1 globally: no /joy_teleop/joy {not buttons[0] in {0, 1}}
2 globally: no /joy_teleop/joy {not buttons[1] in {0, 1}}
3 globally: no /joy_teleop/joy {not buttons[2] in {0, 1}}
4 globally: no /joy_teleop/joy {not buttons[3] in {0, 1}}
5 globally: no /joy_teleop/joy {not buttons[4] in {0, 1}}
6 globally: no /joy_teleop/joy {not buttons[5] in {0, 1}}
7 globally: no /joy_teleop/joy {not buttons[6] in {0, 1}}
8 globally: no /joy_teleop/joy {not buttons[7] in {0, 1}}
9 globally: no /joy_teleop/joy {not buttons[8] in {0, 1}}
10 globally: no /joy_teleop/joy {not buttons[9] in {0, 1}}
11 globally: no /joy_teleop/joy {not buttons[10] in {0, 1}}
12 globally: no /joy_teleop/joy {not axes[0] in [-1.0 to 1.0]}
13 globally: no /joy_teleop/joy {not axes[1] in [-1.0 to 1.0]}
14 globally: no /joy_teleop/joy {not axes[2] in [-1.0 to 1.0]}
15 globally: no /joy_teleop/joy {not axes[3] in [-1.0 to 1.0]}
16 globally: no /joy_teleop/joy {not axes[4] in [-1.0 to 1.0]}
17 globally: no /joy_teleop/joy {not axes[5] in [-1.0 to 1.0]}
18 globally: no /joy_teleop/joy {not axes[6] in [-1.0 to 1.0]}
19 globally: no /joy_teleop/joy {not axes[7] in [-1.0 to 1.0]}
20 globally: no /joy_teleop/cmd_vel {not linear.x in [-0.4 to 0.4]}
21 globally: no /joy_teleop/cmd_vel {not angular.z in [-0.6 to 0.6]}
```

d.7.2 *Properties*

```
1 globally: no /agrob/agrob_mode {not mode in {0,1,2,3}}
2 globally: /agrob/agrob_mode {mode = 2} requires /joy_teleop/joy {axes[5] = -1.0 and
    axes[2] = -1.0}
3 after /joy_teleop/joy {axes[5] > -1.0} until /joy_teleop/joy {axes[5] = -1.0}: no
    /agrob/cmd_vel_joy {not twist.linear.x = 0.0}
```

d.8 AgRob V16 Basic Configuration

d.8.1 *Axioms*

```
1 globally: no /astar_gps_goal
2 globally: no /agrob/target
3 globally: no /agrob/cmd_auto
4 globally: no /fix
5 globally: no /agrob_map/gps/filtered
6 globally: no /agrob_map/imu/rpy
7 globally: no /velodyne_points
8 globally: no /scan {not range_min = 0.01}
9 globally: no /scan {not range_max = 20.0}
10 globally: no /scan {not scan_time = 0.02}
11 globally: no /scan {not angle_min = -2.35619}
12 globally: no /scan {not angle_max = 2.35619}
13 globally: no /scan {not angle_increment = 0.785398}
14 globally: no /scan {not time_increment = 0.0028571428571429}
15 globally: no /scan {not len(ranges) = 7}
16 globally: no /scan {not len(intensities) = 7}
17 globally: no /scan {not ranges[3] in [0.01 to 20.0]}
18 globally: no /lidar1/scan {not range_min = 0.01}
19 globally: no /lidar1/scan {not range_max = 20.0}
20 globally: no /lidar1/scan {not scan_time = 0.02}
21 globally: no /lidar1/scan {not angle_min = -2.35619}
22 globally: no /lidar1/scan {not angle_max = 2.35619}
23 globally: no /lidar1/scan {not angle_increment = 0.785398}
24 globally: no /lidar1/scan {not time_increment = 0.0028571428571429}
25 globally: no /lidar1/scan {not len(ranges) = 7}
26 globally: no /lidar1/scan {not len(intensities) = 7}
27 globally: no /lidar1/scan {not ranges[3] in [0.01 to 20.0]}
28 globally: no /darknet_ros/bounding_boxes {not len(bounding_boxes) = 2}
29 globally: no /darknet_ros/bounding_boxes {not bounding_boxes[0].Class in {"person",
       "other"}}
30 globally: no /darknet_ros/bounding_boxes {not bounding_boxes[1].Class in {"other",
       "person"}}
31 globally: no /imu_um7/rpy {not vector.x in {0, 0.2181662, 0.436333}}
32 globally: no /imu_um7/rpy {not vector.y in {0, 0.1309, 0.2618}}
33 globally: no /imu_um7/rpy {not vector.z = 0.0}
34 globally: no /joy_teleop/joy {not buttons[0] in {0, 1}}
35 globally: no /joy_teleop/joy {not buttons[1] in {0, 1}}
36 globally: no /joy_teleop/joy {not buttons[2] in {0, 1}}
37 globally: no /joy_teleop/joy {not buttons[3] in {0, 1}}
38 globally: no /joy_teleop/joy {not buttons[4] in {0, 1}}
39 globally: no /joy_teleop/joy {not buttons[5] in {0, 1}}
40 globally: no /joy_teleop/joy {not buttons[6] in {0, 1}}
```

```
41 globally: no /joy_teleop/joy {not buttons[7] in {0, 1}}
42 globally: no /joy_teleop/joy {not buttons[8] in {0, 1}}
43 globally: no /joy_teleop/joy {not buttons[9] in {0, 1}}
44 globally: no /joy_teleop/joy {not buttons[10] in {0, 1}}
45 globally: no /joy_teleop/joy {not axes[0] in [-1.0 to 1.0]}
46 globally: no /joy_teleop/joy {not axes[1] in [-1.0 to 1.0]}
47 globally: no /joy_teleop/joy {not axes[2] in [-1.0 to 1.0]}
48 globally: no /joy_teleop/joy {not axes[3] in [-1.0 to 1.0]}
49 globally: no /joy_teleop/joy {not axes[4] in [-1.0 to 1.0]}
50 globally: no /joy_teleop/joy {not axes[5] in [-1.0 to 1.0]}
51 globally: no /joy_teleop/joy {not axes[6] in [-1.0 to 1.0]}
52 globally: no /joy_teleop/joy {not axes[7] in [-1.0 to 1.0]}
```

d.8.2 *Properties*

```
1 globally: no /husky_velocity_controller/cmd_vel {not linear.x in [-0.25 to 0.5]}
2 globally: /darknet_ros/bounding_boxes {bounding_boxes[0].Class = "person"} forbids
    /husky_velocity_controller/cmd_vel {linear.x > 0.0} within 2000 ms
3 until /joy_teleop/joy: no /husky_velocity_controller/cmd_vel {not linear.x = 0.0}
4 after /imu_um7/rpy {not vector.x < 0.2181662 and not vector.y < 0.1309}:
    /husky_velocity_controller/cmd_vel {linear.x > 0.25} requires /imu_um7/rpy
    {vector.x = 0 and vector.y = 0}
```

E

PROPERTY-BASED TESTING EVALUATION RESULTS

e.1 TurtleBot2 Random Walker Controller Node

For the Random Walker Controller node, we wrote a specification consisting of 11 axioms and 4 properties, as shown in Appendix D.2. Out of these properties, we were able to generate 34 useful mutants, while 36 mutants were discarded. The following table shows the distribution of the various language features in the final specification.

Language Feature	Axioms	Properties	Mutants
globally	11	2	15
after	-	-	1
until	-	1	9
after ...**until**	-	1	9
no	8	2	14
some	-	1	4
causes	-	1	12
forbids	3	-	4
requires	-	-	-
within ...**ms**	3	2	22

There are no precedence mutants (**requires**) for this node because they are all trivial or true properties. Nine of the axioms are shared with the Safety Controller node.

In terms of test results, we achieved a near-perfect mutant score, failing to kill a single mutant with a limit of 200 examples. The mutant that we failed to kill is basically the same mutant that survived in the Safety Controller node, only for a different topic. It requires a specific timing and interaction of messages, in an input trace that is, at least, 4 messages long. In fact, the same counterexample trace suffices in this case. The tool was able to detect errors for other similar examples of the same counterexample length, so we believe that this instance is not beyond the tool's capabilities, but is, rather, a matter of allowing it a greater number of examples. Out of the 33 killed mutants, 13 were reported as flaky tests, i.e., the results were not deterministic. Once again, most of the time, flaky tests happen with mutants that require timing interactions (e.g., falsifying the property via time bounds). The table below summarises these results.

	Value
Mutants	34
Flaky Tests	13 (38.235%)
False Negatives	1 (2.941%)
Mutant Score	0.97059

Regarding other performance criteria, as per the following table, we did not observe anything out of the ordinary. On average, each mutant requires between 10 and 40 examples to falsify (including the shrinking phase), although the tool tends to find an error for the first time relatively early (with 11 or less examples). Each example takes about 3.8 seconds to execute, and most of this time (2.8 seconds) is spent on launching and tearing down the SUT between examples. In total, this means that the average time to kill a mutant and obtain a counterexample is between 38 seconds and three minutes. Despite the relatively long execution time, there are only 4 mutants for which the tool uses more messages than necessary. All of these are flaky tests that might have misled the Hypothesis search strategy.

Metric	Mean	Std. Deviation	Median
Number of examples	40	67	10
Examples to failure	11	41	1
Invalid examples	5	15	0
Shrink attempts	29	53	7
Time per example (s)	3.800	1.711	3.249
Trace duration (s)	0.989	1.223	0.520
Setup/Teardown time (s)	2.811	0.487	2.729
Input trace size (messages)	2	2	2
Excess Messages	0	1	0

e.2 TurtleBot2 Multiplexer Node

For the Multiplexer node, we wrote a shorter specification consisting of 2 axioms and 2 properties, as shown in Appendix D.3. Out of these properties, we were able to generate 17 useful mutants, while 9 mutants were discarded. One of the main reasons for the shorter specification of this node is due to language limitations. For instance, in some cases it would have helped to have a disjunction operator for topics, e.g., '`/b requires /a or /c`'. Although this is a relatively trivial extension to the language, in terms of syntax, semantics and implementation, it has been left for future work. The following table shows the distribution of the various language features in the final specification.

Language Feature	Axioms	Properties	Mutants
`globally`	2	1	10
`after`	-	-	-
`until`	-	1	7
`after` ...`until`	-	-	-
`no`	-	-	-
`some`	-	-	-
`causes`	-	2	14
`forbids`	2	-	2
`requires`	-	-	1
`within` ...`ms`	2	2	17

There are no unary pattern mutants (`no`, `some`) for this node because the original properties are response properties in which the behaviour message uses references to the stimulus message. If we dropped the stimulus message to produce an unary pattern, the message predicate would be left with an undefined reference. Removing all expressions with undefined references would lead to a mutation with multiple changes, which goes against common practice and against our ruleset.

In terms of test results, we achieved a perfect mutant score. Out of the 17 killed mutants, 4 were reported as flaky tests, i.e., the results were not deterministic. The table below summarises these results.

	Value
Mutants	17
Flaky Tests	4 (23.529%)
False Negatives	0
Mutant Score	1.0

Regarding other performance criteria, as per the following table, we did not observe anything out of the ordinary. On average, each mutant requires between 13 and 23 examples to falsify (including the shrinking phase), although the tool tends to find an error for the first time right away with the first example. Each example takes about 3.0 seconds to execute, and most of this time (2.6 seconds) is spent on launching and tearing down the SUT between examples. In total, this means that the average time to kill a mutant and obtain a counterexample is between 40 and 70 seconds. There are only 3 mutants for which the tool uses one message more than necessary, all of which are marked as flaky tests.

Metric	Mean	Std. Deviation	Median
Number of examples	23	17	13
Examples to failure	1	1	1
Invalid examples	1	5	0
Shrink attempts	21	17	12
Time per example (s)	2.965	0.352	2.935
Trace duration (s)	0.392	0.179	0.347
Setup/Teardown time (s)	2.574	0.173	2.588
Input trace size (messages)	2	1	2
Excess Messages	0	0	0

e.3 AgRob V16 Safety Controller Node

For the Safety Controller node, we wrote a specification consisting of 26 axioms and 5 properties, as shown in Appendix D.5. Out of these properties, we were able to generate 54 useful mutants, while 31 mutants were discarded. The following table shows the distribution of the various language features in the final specification.

Language Feature	Axioms	Properties	Mutants
globally	26	2	25
after	-	1	14
until	-	2	15
after ...**until**	-	-	-
no	26	2	9
some	-	1	6
causes	-	-	12
forbids	-	1	10
requires	-	1	17
within ...**ms**	-	1	28

There are no response properties (**causes**) for this node due to the fact that, in general, most of them are false, unless we specify an overwhelming amount of axioms that would make testing unfeasible.

In terms of test results, we achieved a perfect mutant score. Out of the 54 killed mutants, 3 were reported as flaky tests, i.e., the results were not deterministic. Sometimes the same example was able to falsify the property, other times it was not. We have checked that all three flaky tests happen with mutants that are dependent on timing interactions (e.g., falsifying the property via time bounds) as well as floating-point number exploration within a relatively narrow range (e.g., setting a value to exactly -1.0 for a specific index of an array). The table below summarises these results.

	Value
Mutants	54
Flaky Tests	3 (5.556%)
False Negatives	0
Mutant Score	1.0

Regarding other performance criteria, as per the following table, we did not observe anything out of the ordinary, other than the fact that, in comparison with TurtleBot2, we now require a larger number of examples until the final verdict is given (especially in the shrinking phase). On average, each mutant requires between 15 and 30 examples to falsify (including the shrinking phase), although the tool tends to find an error for the first time relatively early (with less than 10 examples). The mean number of examples is actually 75, but note that there is a substantial standard deviation. We do have 13 outliers, requiring about 150 examples or more, with one of them taking as long as 1045 examples – all shrink and reproduction attempts, the failure was found at the first try. Each example takes about 4 seconds to execute, and half of this time is spent on launching and tearing down the SUT between examples. In total, this means that the average time to kill a mutant and obtain a counterexample is less than two minutes; for the outliers, it takes about 15 minutes. Despite the relatively long execution time, there are only 4 mutants for which the tool uses more messages than necessary. One of these (the highest one, using 8 messages rather than the minimum of 2) is a flaky test, which might have misled the Hypothesis search strategy.

Metric	Mean	Std. Deviation	Median
Number of examples	75	153	19
Examples to failure	10	24	1
Invalid examples	0	0	0
Shrink attempts	66	148	18
Time per example (s)	4.775	1.876	3.959
Trace duration (s)	2.774	1.509	2.038
Setup/Teardown time (s)	2.001	0.368	1.921
Input trace size (messages)	1	1	1
Excess Messages	0	1	0

e.4 AgRob V16 Joystick Controller Node

For the Joystick Controller node, we wrote a specification consisting of 21 axioms and 3 properties, as shown in Appendix D.7. The large number of axioms is due to the lack of full support for quantifiers in our implementation, at the time of writing, and joystick messages involve arrays of values. With proper quantifier support, we could write the same specification with just 4 axioms. Out of the 3 properties, we were able to generate 29 useful mutants, while 21 mutants were discarded. The following table shows the distribution of the various language features in the final specification.

Language Feature	Axioms	Properties	Mutants
`globally`	21	2	18
`after`	-	-	1
`until`	-	-	-
`after` ...`until`	-	1	10
`no`	21	2	13
`some`	-	-	3
`causes`	-	-	5
`forbids`	-	-	1
`requires`	-	1	7
`within` ...`ms`	-	-	10

There are no relevant existence properties (`some`) for this node, because its behaviour is reactive. There are no response or prevention properties (`causes`, `forbids`) due to the fact that they are mostly false. There is no rate limit for incoming joystick messages, and most properties (as was the case with TurtleBot2's Safety Controller) have the obvious counterexample of sending a stimulus message and immediately after sending another message that cancels it or makes it irrelevant, before the node has any time to respond.

In terms of test results, we achieved a very high mutant score, failing to kill three mutants with a limit of 200 examples. The mutants that we failed to kill are not related to each other, but share the same underlying problem: they all require producing specific floating-point values (exactly -1.0) in two specific indexes of an array of size 8. These mutants are not different from many others that were successfully killed, in any measure of significance. Thus, we believe that the cause was randomness; it just so happened that Hypothesis did not explore these specific values for these mutants within the maximum number of examples, while it did for other mutants.

Out of the 26 killed mutants, 4 were reported as flaky tests, i.e., the results were not deterministic. In all cases, flaky tests are related to timing interactions. The table below summarises these results.

	Value
Mutants	29
Flaky Tests	4 (13.793%)
False Negatives	3 (10.345%)
Mutant Score	0.89655

Regarding other performance criteria, as per the following table, this node required a significantly larger number of examples to falsify properties. Only 12 mutants require less than 50 examples to kill (made obvious with the median value of 47). Among these 12 mutants, the mean is 18 examples. The remaining killed mutants require more than 150 examples, with 4 of them requiring more than 400 examples (with the vast majority belonging to the shrinking phase). Each example takes about 4 seconds to execute, and half of this time is spent on launching and tearing down the SUT between examples. This means that the average time to kill a mutant and obtain a counterexample is a minute or less for the mutants in the lower

range of examples, and 15 minutes or more for the larger ones. There are five mutants for which the tool uses more messages than necessary, and two of them are flaky tests.

Metric	Mean	Std. Deviation	Median
Number of examples	172	247	47
Examples to failure	25	49	2
Invalid examples	0	0	0
Shrink attempts	147	236	17
Time per example (s)	4.212	1.002	3.996
Trace duration (s)	2.280	0.789	2.123
Setup/Teardown time (s)	1.931	0.213	1.873
Input trace size (messages)	2	2	1
Excess Messages	0	1	0