



UNIVERSITY OF GENOVA

PHD PROGRAM IN BIOENGINEERING AND ROBOTICS

# Planning, Acting and Monitoring in Robotics Environments

by

**Stefano Bernagozzi**

Thesis submitted for the degree of *Doctor of Philosophy* (38° cycle)

July 2026

Lorenzo Natale  
Paolo Massobrio

Supervisor  
Head of the PhD program

***Thesis Jury:***

Marco Roveri, *Università di Trento*  
Sarah Keren, *Technion - Israel Institute of Technology*  
Luca Iocchi, *Università di Roma "La Sapienza"*

External examiner  
External examiner  
External examiner

**Dibris**

Department of Informatics, Bioengineering, Robotics and Systems Engineering

I would like to dedicate this thesis to Anna, Giulio, Giulia, Alberto and Marco.  
This is for all the support that you gave me.

## **Declaration**

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Stefano Bernagozzi

July 2026

## Acknowledgements

I would like to acknowledge firstly Lorenzo, which gave me the possibility to pursue this PhD.

Secondly I would like to thank all of my colleagues at IIT for the support and the collaboration during these years and the shared coffee breaks with a lot of fun. In particular, I would like to thank Federico, Gabriele and Davide who shared with me the whole journey here.

I would like also to thank Armando, Enrico and Karim for the support and the hospitality at the university of Genoa, full of interesting discussions and the possibility to work together on the museum use case.

Another special thanks goes to Iran and Charlie for giving me the possibility to work with them and spend a fantastic time in Birmingham, and to whom I am grateful for the support and the collaboration during the work on the flow-aware planning.

In Birmingham I also met Federico, with whom I shared a lot of fun and interesting discussions, and to whom I am grateful for the support and the shared time there.

To Nicholas and Gabriele I need to say a special thank, because he helped me when I was struggling at home with wine and pizza.

To Giulio, Giulia, Carlotta and Ilaria I would like to say a special thank you for the support and the fun we shared together.

To Daniele, Davide, Deiv, Mark, Michele, Chiara, Matteo and Lucia I would like to say a special thanks for the moments shared together, even if I could have not spent more time with them due to the distance, but I am grateful for the support and the fun we shared together.

I would also like to thank Alberto, Anca, Vittoria and Antonio who helped me during these years where I was far from home and to whom I am grateful for the support and the fun we shared together.

Finally, I would like to thank my family for the support and the patience during these years, and in particular my parents, my brother and my girlfriend for being always there for me.

## Abstract

As hardware and software costs continue to drop, consumer robotics has become easier to develop and increasingly attractive to industry. As a result, autonomous mobile robots are being deployed in unstructured public environments such as malls, hotels, and museums. In these settings, robots face unpredictable changes and failure modes and when they encounter situations outside their design assumptions, they may fail and require human intervention.

Mitigating failures in the field typically requires updating the software to address newly observed issues and re-testing previously working scenarios. This process is time-consuming and difficult to carry out exhaustively in practice, especially as users demand more human-like robots whose behavior must be updated at runtime or extended with new capabilities. A key challenge is therefore ensuring that the robot's executed behavior matches its specified behavior, particularly for complex, large-scale systems.

In this thesis I address these challenges by presenting (i) a model-based framework that generates executable code from deliberation models, enabling off-line correctness verification and on-line execution monitoring and (ii) a task-level planner for operation in crowded environments.

Together with my colleagues at IIT, I developed a model-based code generator that generates executable code from an XML-based language based on the interpretation of standard SCXML code. My contributions also include implementing the validation use cases, defining their verification and monitoring properties, and grounding MOON in concrete robotic scenarios. The robotics domain expertise gained from the robotic tour-guide use case guided the development of the monitoring approach, while feedback from the validation activities informed design decisions for the shared tools.

To address behavioral correctness, deliberation components are modeled at a conceptual level using Model-Based Development (MBD). Executable code is therefore automatically generated from these models, allowing the resulting behavior to be verified with existing model checkers and monitored at runtime using a monitoring tool.

For deployment in congested areas, I developed a probabilistic tour planner that leverages learned Circular Linear Flow Field (CLiFF) maps to predict pedestrian motion. The planner

formulates and solves a Markov Decision Process (MDP) on-line to route the robot while adapting to newly observed pedestrians.

We then integrated the generator and the software mentioned above into a robotic tour guide developed within the CONVINCe consortium and evaluated in simulation and real-world experiments.

## List of Publications

The work presented in this thesis has been carried out during my PhD in *Advanced and Humanoid Robotics* from November 2022 to January 2026. The presented outcome resulted in the following papers:

- **Bernagozzi, S.**, Ferrando, A., Ghiorzi, E., Natale, L., & Tacchella, A. “Modeling Robot Control Architectures for Verification and Monitoring”. In *2024 European Robotics Forum* (pp. 191-195). Cham: Springer Nature Switzerland.
- **Bernagozzi, S.**, Faraci, S., Ghiorzi, E., Pedemonte, K., Ferrando, A., Natale, L., & Tacchella, A. “Model-based verification and monitoring for safe and responsive robots”. In *2025 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAN)* (pp. 1-6). IEEE.
- **Bernagozzi, S.**, Faraci, S., Ghiorzi, E., Pedemonte, K., Natale, L., & Tacchella, A. “Code Generation and Monitoring for Deliberation Components in Autonomous Robots”. In *2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (pp. 16762-16767). IEEE.
- **Bernagozzi, S.**, Street, C., Mansouri, M., Natale, L. “Congestion-Aware Robot Tour Planning in Crowded Environments”. In *2026 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (to appear) IEEE.
- Street, C., Warsame, Y., Mansouri, M., Klauck, C., Henkel, C., Lampacrescia, M., Palmas, M., Lange, R., Ghiorzi, E., Tacchella, A., Azrou, R., Lallement, R., Morelli, M., Chen, G. I., Wallis, D., **Bernagozzi, S.**, Rosa, S., Randazzo, M., Faraci, S., Natale, L. “Towards a verifiable toolchain for robotics”. In *2024 Proceedings of the AAAI Symposium Series* (Vol. 4, No. 1, pp. 398-403).

This thesis builds upon the aforementioned papers to present a cohesive discussion that highlights the overall contributions of the PhD. Consequently, certain ideas and results have already been published in these works.

## List of Excluded Publications

The following publications were produced during my PhD but are not included among the core publications on which this thesis is based:

- Ghiorzi, E., Colledanchise, M., Piquet, G., **Bernagozzi, S.**, Tacchella, A., & Natale, L. “Learning Linear Temporal Properties for Autonomous Robotic Systems”. *IEEE Robotics and Automation Letters*, 8(5), 2930–2937, 2023.
- Rosa, S., Randazzo, M., Landini, E., **Bernagozzi, S.**, Sacco, G., Piccinino, M., & Natale, L. “Tour Guide Robot: A 5G-Enabled Robot Museum Guide”. *Frontiers in Robotics and AI*, 10, 2024.
- Calabrese, C., Gower, V., Randazzo, M., Randazzo, M., Brand, F., Mattos, F. G. M., Spinelli, F., Fussi, M., Landini, E., **Bernagozzi, S.**, et al. “Socially Assistive Robot to Administer and Assess the Timed Up and Go Test: A Feasibility Study”. *International Journal of Social Robotics*, 17(1), 133–146, 2025.



# Table of contents

|                                                  |               |
|--------------------------------------------------|---------------|
| <b>List of figures</b>                           | <b>xii</b>    |
| <b>List of tables</b>                            | <b>xv</b>     |
| <b>Nomenclature</b>                              | <b>xvi</b>    |
| <b>1 Introduction</b>                            | <b>1</b>      |
| 1.1 Motivation . . . . .                         | 1             |
| 1.2 Contribution . . . . .                       | 3             |
| 1.3 Structure of Thesis . . . . .                | 5             |
| <br><b>I Code Generation and Verification</b>    | <br><b>7</b>  |
| <b>2 Background</b>                              | <b>10</b>     |
| 2.1 Layered Architecture . . . . .               | 10            |
| 2.2 Behavior Trees . . . . .                     | 14            |
| 2.2.1 Control Flow Nodes . . . . .               | 16            |
| 2.2.2 Decorator Nodes . . . . .                  | 18            |
| 2.2.3 Plug-in Nodes . . . . .                    | 18            |
| 2.3 Finite State Machines . . . . .              | 19            |
| 2.4 Middleware . . . . .                         | 21            |
| 2.5 Past Metric Temporal Logic . . . . .         | 21            |
| 2.5.1 Patterns . . . . .                         | 22            |
| <br><b>3 Related Work</b>                        | <br><b>24</b> |
| 3.1 Code Generation . . . . .                    | 24            |
| 3.1.1 Model-driven robotics ecosystems . . . . . | 25            |

|           |                                                             |           |
|-----------|-------------------------------------------------------------|-----------|
| 3.1.2     | Generation from behavioral and formal models . . . . .      | 27        |
| 3.1.3     | Directly executable behavior models . . . . .               | 28        |
| 3.1.4     | Positioning of M2C . . . . .                                | 28        |
| 3.2       | Model Checking . . . . .                                    | 29        |
| 3.3       | Monitoring . . . . .                                        | 30        |
| 3.4       | Summary . . . . .                                           | 31        |
| <b>4</b>  | <b>Framework</b>                                            | <b>33</b> |
| 4.1       | Using the Framework . . . . .                               | 34        |
| 4.2       | Architecture . . . . .                                      | 36        |
| 4.3       | Modelling Language . . . . .                                | 38        |
| 4.3.1     | Semantic Continuity Across the Framework . . . . .          | 39        |
| 4.4       | Code Generation . . . . .                                   | 40        |
| 4.4.1     | How the Generated Model Works . . . . .                     | 41        |
| 4.5       | Verification . . . . .                                      | 42        |
| 4.5.1     | Model Translation for Verification and Monitoring . . . . . | 42        |
| 4.5.2     | Off-Line Verification . . . . .                             | 43        |
| 4.5.3     | On-Line Verification . . . . .                              | 44        |
| <b>5</b>  | <b>Validation of the Framework</b>                          | <b>47</b> |
| 5.1       | First Case Study: Simple Behavior Tree . . . . .            | 47        |
| 5.1.1     | Off-line Verification . . . . .                             | 48        |
| 5.1.2     | On-line Verification . . . . .                              | 48        |
| 5.2       | Second Case Study: Enhanced Behavior Tree . . . . .         | 49        |
| 5.2.1     | Off-line Verification . . . . .                             | 51        |
| 5.2.2     | On-line Verification . . . . .                              | 53        |
| 5.2.3     | Fault Injection and Monitor Overhead . . . . .              | 54        |
| 5.3       | Results . . . . .                                           | 58        |
| <b>6</b>  | <b>Discussion</b>                                           | <b>60</b> |
| 6.1       | Interpretation and Generalizability . . . . .               | 61        |
| 6.2       | Open Issues and Future Extensions . . . . .                 | 62        |
| <b>II</b> | <b>Flow-Aware Planning</b>                                  | <b>63</b> |
| <b>7</b>  | <b>Background</b>                                           | <b>67</b> |

|            |                                                         |            |
|------------|---------------------------------------------------------|------------|
| 7.1        | Topological Maps . . . . .                              | 67         |
| 7.2        | Hamiltonian Path Problem . . . . .                      | 67         |
| 7.3        | Circular Linear Flow Field (CLiFF) Maps . . . . .       | 69         |
| 7.4        | Poisson Binomial Distribution . . . . .                 | 70         |
| 7.5        | Markov Decision Processes . . . . .                     | 70         |
| <b>8</b>   | <b>Related Work</b>                                     | <b>72</b>  |
| 8.1        | Map Of Dynamics . . . . .                               | 72         |
| 8.2        | Social Navigation . . . . .                             | 74         |
| 8.3        | Task-Level Planning under Congestion . . . . .          | 75         |
| 8.4        | Research Gap . . . . .                                  | 77         |
| <b>9</b>   | <b>Framework</b>                                        | <b>78</b>  |
| 9.1        | Modeling Rationale and Action Abstraction . . . . .     | 79         |
| 9.1.1      | Action Representation in the Tour MDP . . . . .         | 80         |
| 9.2        | Solution Overview . . . . .                             | 81         |
| 9.3        | Modeling Congestion . . . . .                           | 81         |
| 9.4        | Tour Planning under Human Congestion . . . . .          | 84         |
| 9.4.1      | Why Receding-Horizon Replanning . . . . .               | 87         |
| <b>10</b>  | <b>Evaluation of Flow-Aware Planning</b>                | <b>88</b>  |
| 10.1       | Experimental Setup . . . . .                            | 89         |
| 10.2       | Results . . . . .                                       | 94         |
| 10.2.1     | Answers to the Evaluation Questions . . . . .           | 99         |
| <b>11</b>  | <b>Discussion</b>                                       | <b>101</b> |
| 11.1       | Uncertainty and Evaluation Scope . . . . .              | 102        |
| 11.2       | Limitations of the Model . . . . .                      | 103        |
| <b>III</b> | <b>Museum Guide Case Study</b>                          | <b>105</b> |
| <b>12</b>  | <b>From Dependable Execution to Adaptive Planning</b>   | <b>106</b> |
| 12.1       | Recap of Model-Based Execution and Monitoring . . . . . | 106        |
| 12.2       | Recap of Flow-Aware Tour Planning . . . . .             | 108        |
| 12.3       | How the Two Parts Meet in the Museum . . . . .          | 109        |
| <b>13</b>  | <b>Robotic Tour Guide Case Study Description</b>        | <b>111</b> |

---

|                                                                     |                |
|---------------------------------------------------------------------|----------------|
| <b>14 Verification</b>                                              | <b>120</b>     |
| 14.1 Off-Line Verification . . . . .                                | 123            |
| 14.2 On-line Verification . . . . .                                 | 124            |
| <b>15 Using Flow-Aware Planning in a Real-World Museum Scenario</b> | <b>128</b>     |
| 15.1 Changes for Museum Tour Guide Case Study . . . . .             | 128            |
| 15.1.1 Main Changes . . . . .                                       | 129            |
| 15.2 Data Collection and Processing . . . . .                       | 132            |
| 15.3 Integration with the Behavior Tree . . . . .                   | 134            |
| 15.4 Planner Execution Cycle . . . . .                              | 136            |
| 15.5 Evaluation . . . . .                                           | 136            |
| <b>16 Discussion</b>                                                | <b>141</b>     |
| 16.1 Failure Cases and Practical Limits . . . . .                   | 142            |
| <br><b>IV Conclusion and Open Research Directions</b>               | <br><b>143</b> |
| <b>17 Conclusion</b>                                                | <b>144</b>     |
| <b>18 Open Research Directions</b>                                  | <b>146</b>     |
| <b>References</b>                                                   | <b>148</b>     |
| <br><b>Appendix A ASCXML Language</b>                               | <br><b>156</b> |
| A.1 Models for Skills and Components . . . . .                      | 156            |
| A.2 SCXML Tags Excluded by ASCXML . . . . .                         | 156            |
| A.3 Event and Communication Semantics . . . . .                     | 157            |
| A.4 ROS Extensions . . . . .                                        | 158            |
| A.5 Behavior Tree Extensions . . . . .                              | 159            |

# List of figures

|     |                                                                                                                                                                                                                                                                                                                                                                                                           |    |
|-----|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 1.1 | R1 robot inside Palazzo Madama in Turin during a trial. . . . .                                                                                                                                                                                                                                                                                                                                           | 5  |
| 2.1 | Three layers of the robotic architecture (following the terminology of Colledanchise et al. [2021]) and their position with respect to hardware and environment. . . . .                                                                                                                                                                                                                                  | 11 |
| 2.2 | Deliberative layer functions — picture from Ingrand and Ghallab [2017] .                                                                                                                                                                                                                                                                                                                                  | 13 |
| 2.3 | Example of a Behavior Tree used in this thesis for the robotic museum guide (simplified). The BT controls the robot to move across different points of interest (POIs). Such behavior is obtained by orchestrating the navigation component (GoToPOI) and a shared blackboard (SetPOI, SetCurrentPOIDone). The BT also checks the level of the battery, raising an alarm if the battery depleted. . . . . | 15 |
| 4.1 | Architecture used in this part of the thesis. The deliberation layer is composed of the BT and its plugins, the skill layer contains FSM-based skill models and generated skill implementations, and the functional layer contains the components interacting with the environment. The green marks indicate the communications observed by the monitoring infrastructure. . . . .                        | 34 |
| 4.2 | Overview of the various framework components and their interactions. . . .                                                                                                                                                                                                                                                                                                                                | 35 |
| 4.3 | From mental models and system properties to formal models. . . . .                                                                                                                                                                                                                                                                                                                                        | 37 |
| 4.4 | Schematic view of the generated model. The SCXML state machine is the core logic of the skill, while the generated ROS 2 code acts as an adapter between the state machine and the environment. . . . .                                                                                                                                                                                                   | 42 |
| 5.1 | A simple BT that checks the battery level, if it is higher than 30% it drains the battery, otherwise it raises an alarm. . . . .                                                                                                                                                                                                                                                                          | 49 |
| 6.1 | Example map layout with three POIs and two possible routes to $P_2$ from $P_1$ .                                                                                                                                                                                                                                                                                                                          | 66 |

|      |                                                                                                                                                                                        |     |
|------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| 7.1  | Example of a topological map, where vertices represent POIs and edges represent paths between them. . . . .                                                                            | 68  |
| 9.1  | Approach for computing human congestion probabilities from a CLiFF map.                                                                                                                | 81  |
| 9.2  | Mapping human positions to topological edges. . . . .                                                                                                                                  | 83  |
| 10.1 | The topological maps for the ATC environment with 11 and 16 nodes. . . .                                                                                                               | 92  |
| 10.2 | The topological maps for the ATC environment with 21 and 26 nodes. . . .                                                                                                               | 93  |
| 10.3 | Execution time of the CPV2 solution, the CPV2 bounded solution , and the Hamiltonian-path solution for 11 and 16 maps sizes . . . . .                                                  | 96  |
| 10.4 | Execution time of the CPV2 solution, the CPV2 bounded solution , and the Hamiltonian-path solution for 21 and 26 maps sizes . . . . .                                                  | 97  |
| 10.5 | Planning time for the first CPV2 planning step under different numbers of congestion bands on different map sizes. . . . .                                                             | 98  |
| 13.1 | Schematic view of the top-level Behavior Tree of a museum tour-guide application. Sub-trees collapse lower-level BT fragments whose leaves are implemented by the skill layer. . . . . | 113 |
| 13.2 | Schematic view of the sub-tree responsible for checking the battery level and managing the charging process. . . . .                                                                   | 114 |
| 13.3 | Schematic view of the sub-tree responsible for checking if the museum is open and the robot is at start position. . . . .                                                              | 115 |
| 13.4 | Schematic view of the sub-tree responsible for setting the next POI. . . . .                                                                                                           | 115 |
| 13.5 | Schematic view of the sub-tree responsible for selecting the next POI to explain. . . . .                                                                                              | 116 |
| 13.6 | Schematic view of the sub-tree responsible for managing the tour duration. .                                                                                                           | 117 |
| 13.7 | Schematic view of the sub-tree responsible for going to the next POI. . . . .                                                                                                          | 117 |
| 13.8 | Schematic view of the sub-tree responsible for checking if people are following the robot. . . . .                                                                                     | 118 |
| 13.9 | Schematic view of the sub-tree responsible for checking the artwork, explaining it and answering questions. . . . .                                                                    | 119 |
| 14.1 | Wrong node inside a BT during SCAN validation. . . . .                                                                                                                                 | 124 |
| 14.2 | Traces of the battery level variable during wrong SCAN execution. . . . .                                                                                                              | 125 |
| 14.3 | Traces of the battery level variable during SCAN execution. . . . .                                                                                                                    | 125 |
| 14.4 | Traces of the POI selection and completion variables during SCAN execution.                                                                                                            | 125 |

---

|      |                                                                                                                                                 |     |
|------|-------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| 15.1 | Left: a LiDAR device that was placed inside the museum to track the visitors.<br>Right: one of the devices placed inside the museum. . . . .    | 132 |
| 15.2 | The predefined museum map with the scans of the eight LiDAR devices<br>placed in the rooms overlaid, together with the detected people. . . . . | 133 |
| 15.3 | The CLiFF map showing the visitors' flow across the rooms of the museum.                                                                        | 134 |
| 15.4 | The graph of the possible paths connecting the tour POIs and their alternative<br>positions. . . . .                                            | 135 |
| 15.5 | Screenshots of the museum environment simulated in Gazebo. . . . .                                                                              | 136 |
| 15.6 | Average tour total time, number of navigation aborts and recoveries during<br>the simulated tours. . . . .                                      | 139 |
| 15.7 | Average time, number of navigation aborts and recoveries for each POI. . .                                                                      | 140 |

# List of tables

|      |                                                                                                                                                       |     |
|------|-------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| 2.1  | Behavior Tree nodes used in this work divided by category. . . . .                                                                                    | 16  |
| 5.1  | Properties with the related pattern taken from Ulus [2019a], the text expressed in formal language, and the corresponding PAST-MTL formula. . . . .   | 52  |
| 5.2  | Model checking results with respect to temporal bounds . . . . .                                                                                      | 53  |
| 5.3  | Monitoring results with respect to temporal bounds . . . . .                                                                                          | 54  |
| 5.4  | Average tick time measured with different monitoring configurations. . . .                                                                            | 56  |
| 5.5  | Summary of the fault-injection experiments on the second case study. . . .                                                                            | 57  |
| 10.1 | The sizes of the maps created in the ATC domain. . . . .                                                                                              | 89  |
| 10.2 | The congestion bands used in the experiments. . . . .                                                                                                 | 90  |
| 10.3 | The Mann-Whitney U p-values comparing CPV2 and CPV2 bounded to the Hamiltonian path solver, and p-values comparing CPV2 against CPV2 bounded. . . . . | 95  |
| 14.1 | Monitoring properties considered in the case study. . . . .                                                                                           | 121 |
| 14.2 | Results from the tours in Palazzo Madama. . . . .                                                                                                     | 127 |
| 15.1 | Statistics on 30 simulated tours with and without congestion replanning . .                                                                           | 138 |
| 15.2 | Mann-Whitney U test results comparing the distributions of total time, recoveries and aborts across the 30 tours with and without CPV2. . . . .       | 138 |



# Nomenclature

## Roman Symbols

AS2FM Autonomous System to Formal Model

ASCXML Autonomous system's SCXML

BT Behavior Tree

CLiFF Circular-Linear Flow Field

CLiFF-LHMP Circular-Linear Flow Field for Long-term Human Motion Prediction

CONVINCE CONtext-aware Verifiable and adaptIve dyNamiC dELiberation

CPV2 Coverage Plan Version 2

DSL Domain-Specific Language

FIFO First In First Out

FSM Finite State Machine

LRTDP Labeled Real-Time Dynamic Programming

M2C Model to Code

MBD Model-Based Development

MDE Model-Driven Engineering

MDP Markov Decision Process

MOON MOnitoring On-line

MST Minimum Spanning Tree

MTL Metric Temporal Logic

past-MTL Past Metric Temporal Logic

POI Point Of Interest

RoAML Robotics Architecture Modeling Language

ROS Robot Operating System

SCAN StatistiCal ANalyzer

SCXML State Chart XML

SMC Statistical Model Checking

SSP Stochastic Shortest Path

SWGMM Semi-Wrapped Gaussian Mixture Model

TSP Traveling Salesman Problem

XML Extensible Markup Language

YARP Yet Another Robot Platform

# Chapter 1

## Introduction

### 1.1 Motivation

As hardware robotic platforms become more reliable and affordable, many companies develop custom software to control their robots. This software is often not completely tested or validated, and may therefore contain errors that are difficult to discover. Moreover, the design and implementation of such software is still a complex and error-prone task because there is an intrinsic difficulty in the process of integrating and orchestrating components so that they achieve a given behavior.

Many problems may arise during deployment because developers make incorrect assumptions about how a component works, mainly due to insufficient or misinterpreted documentation, or about how modules interact. Finally, a developer may make incorrect assumptions about how the robot functions, or how the environment responds to the robot's actions. For example, software that uses a Behavior Tree (BT) as a top-level deliberation component must also ensure that all BT skills interact correctly with the environment and that the resulting behavior is correct. This is not a trivial task, since the BT may be complex and may involve many skills, and thus it is easy to make mistakes in the design and implementation of the whole system, even if the individual components are correctly implemented.

This trend is leading to concentrated efforts in the design and implementation of deliberation layer control software. The wide adoption of the ROS 2 middleware in the robotics community has simplified the development of software components, and has made available a large number of off-the-shelf components that can be used and integrated. Yet, this is not sufficient to ensure that the resulting control software is correct and safe. The focus, therefore, is not only on the individual components, but also on the part of the system (the

*deliberation layer*) that at each given moment, decides what to do, by sending commands that configure, enable and disable individual components.

Because of the complexity of the task, and because of the need to ensure safe and secure operation, it is important to have methodologies that can help developers to design and implement control software for robots in a way that minimizes the risk of errors and their consequences.

One of these methodologies is Model-Based Development (MBD), a systematic approach to software development that relies on the use of models to represent the system being developed, while the generation of the code to be executed on the robot is automated. This allows developers to focus only on the high-level design of the system, rather than focusing on the implementation which should be then validated against the design and the requirements.

However, solely relying on MBD is still not enough to ensure that an entire control system is correct and safe. In particular, it is important to have a way to verify not only that the implementation of individual components does not introduce errors, but also that their integration and execution does not lead to unsafe behavior. This can be achieved through two different but complementary approaches: off-line verification and on-line monitoring.

Off-line verification is the process of checking the correctness of the design of the system while it is being designed. It can help to catch errors early in the design process, avoiding expensive tests with simulated environments or real-world operation, and to ensure that the design of the system is correct before moving on to the implementation stage. One of the most common techniques to achieve off-line verification is model checking, a formal method that allows to exhaustively explore the state space of a model. For large systems, however, model checking can be computationally expensive or even infeasible, and thus it is often necessary to use statistical model checking, a technique that relies on sampling to estimate the probability that a property is satisfied.

On-line monitoring, on the other hand, is the process of checking the behavior of the system during its execution. It is meant to detect errors that may have been missed during the design and implementation stages, and to ensure that the system behaves as expected in real-world scenarios. On-line monitoring can be achieved through the use of monitors, software components that observe the messages exchanged over the system and check whether those messages satisfy certain properties or are compliant to a certain model given to the monitor. Monitors are implemented in a way that they do not interfere with the normal operation of the system, and they can be used to trigger alarms or to take corrective actions when an error is detected.

To enhance the strength of MBD, it is also important to have a clear and strong architecture for the control software, that can help to modularize the design and implementation of the system, and to make it easier to verify and monitor.

Moreover, robots should also be able to dynamically adapt their behavior depending on environmental changes, since they are often deployed in non-structured environments or public places. This task can be achieved via the integration of planning strategies that enable the robot to perform actions which mitigate or avoid disruptions and interferences. The main downside of planning is that it can be computationally expensive, and thus can be infeasible if it has to be used on-line. To overcome this problem, real-time planning techniques can be used to generate plans in a short amount of time, often by using heuristics or by limiting the search space.

This thesis addresses the design and implementation of complete control software for robots using MBD in the first part, with a particular emphasis on code generation. In the second part it focuses on the integration of planning techniques to enable dynamic adaptation of the robot's behavior in response to environmental changes, mainly in crowded environments. Finally, all these components are validated in a real-world scenario which is a robotic tour guide.

This thesis was carried out within the CONVINCe project<sup>1</sup>. Some tools and software components discussed in the following chapters were developed by project partners and are reported here because they are necessary to understand the complete workflow. Whenever a chapter relies on shared project infrastructure, the text separates the parts that are individual contributions of this thesis from the surrounding CONVINCe components.

## 1.2 Contribution

The first part of the thesis focuses on the design and implementation of a framework for model-based development of control software for robots, developed together with other partners within the CONVINCe project. In this part I will present the creation of a behavior for a robot, starting from a simple case and then moving to a more complex one. I will also show how the various tools we developed can be used to design and implement the behavior and to validate it using off-line verification and on-line monitoring. My main contributions in this part are the development of software for generating executable code from models, the integration of the various tools into a common workflow, and the implementation of the validation use cases. For these use cases, I defined the verification and monitoring properties

---

<sup>1</sup><https://convince-project.eu>

and grounded MOON in concrete robotic scenarios. I also provided the robotics domain expertise needed to guide the development of the monitoring approach, and used feedback from the validation activities to contribute to design decisions for the shared tools.

The second part of the thesis focuses on the design and implementation of a real-time robot tour planner for crowded environments. In this part I will present the design and implementation of the planner, and I will show how it can be used to enable a robot to dynamically adapt its behavior in response to environmental changes, by generating plans in a short amount of time. My main contribution of this part is a flow-aware planning formulation that lifts short-horizon predictions of human motion to a task-level stochastic shortest-path problem. The planner represents navigation choices as actions over a topological abstraction, uses predicted congestion distributions to define stochastic transition costs, and solves the resulting MDP on-line in a receding-horizon loop.

Subsequently, in the third part I will present the integration of all the techniques developed in the previous parts, and I will show how they can be used together to design and implement a behavior for a robot that can be deployed in a real-world scenario. In particular, I will present a case study of a robotic tour guide operating in a museum environment that we developed at the Italian Institute of Technology (IIT) on the R1 robot (Figure 1.1), where the robot must interact with visitors, navigate between points of interest, and adapt to crowd dynamics. This case study will allow me to validate the proposed ideas on the complete robotic tour-guide application, showing how runtime verification and flow-aware planning play different but complementary roles in the overall architecture. Verification and monitoring increase confidence in the correctness and health of the system, while the planner improves the efficiency and robustness of the robot's behavior in the presence of visitors. Taken together, these elements illustrate how the thesis contributions can be combined within a single deployed system, rather than being considered in isolation. My main contributions in this part include the design and implementation of the robotic tour-guide behavior, the definition of the properties used to verify and monitor it, and the provision of the robotics domain expertise that guided both behavior and monitoring development. They also include validating the design and implementation using off-line verification and on-line monitoring, feeding the resulting domain and validation feedback into tool-design decisions, deploying the framework on the R1 robot in a real-world scenario, and validating the planner in a real-world scenario.

The thesis should therefore be read with three ownership levels in mind. First, M2C, the integration workflow, the validation scenarios, the flow-aware planning formulation, and the museum planner adaptation are individual thesis contributions. Second, the deployed

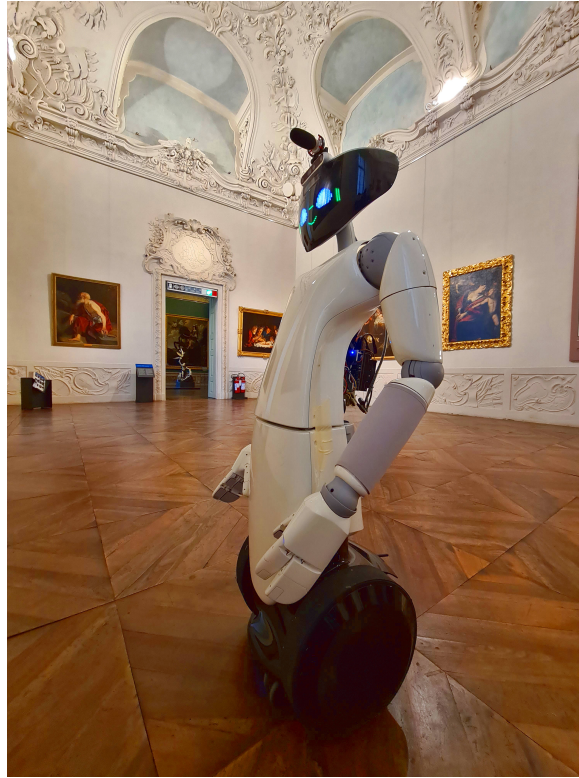


Figure 1.1 R1 robot inside Palazzo Madama in Turin during a trial.

museum demonstrator and some enabling software were developed collaboratively within IIT and the CONVINC consortium. Third, partner tools such as AS2FM, SCAN, MOON, and the broader RoAML specification layer are used as shared project infrastructure: they are described when needed for scientific completeness. My contribution to these shared tools was use-case driven: I implemented the validation scenarios, defined the relevant properties, grounded MOON in robotic applications, and provided robotics domain expertise and validation feedback that informed their design decisions.

## 1.3 Structure of Thesis

The thesis is organized into four parts. Part I focuses on model-based development, code generation, off-line verification, and on-line monitoring for robotic control software. It first introduces the background concepts needed throughout the thesis, including Behavior Trees, finite-state machines, ROS 2 communication, temporal properties, model checking, and runtime monitoring. It then presents the framework, showing how design models are translated into executable skill code and how the same models can be used to support

verification and monitoring. The part concludes with the validation of the framework on progressively richer robotic behaviors.

Part II introduces the flow-aware planning contribution. In the first chapter it introduces some background notions, such as topological maps, Markov decision processes, Circular Linear Flow Field maps, the Hamiltonian path problem and Poisson binomial distribution. Then, after reviewing the relevant literature on human-aware planning, congestion modeling, and tour planning, it defines the planning framework used in the thesis. It then defines how the planner was implemented and evaluates the planner on synthetic data and compares it with a baseline that do not reason over future congestion.

Part III brings the previous contributions together in the museum guide case study. It describes a robotic tour-guide application, the Behavior Tree and skill architecture, the verification and monitoring properties considered for the system, and the integration of the flow-aware planner into a complete tour-guide behavior. This part shows how the methods developed in the previous parts interact in a single robotic application: verification and monitoring support correctness and diagnosability, while planning improves the robustness and efficiency of execution in the presence of visitors.

Finally, Part IV summarizes the main contributions of the thesis and discusses the open research directions that this thesis brings up.



# **Part I**

## **Code Generation and Verification**

---

The development of robotic control software often requires balancing rapid implementation with the need for structure, traceability, and dependability. When these concerns are addressed separately, the resulting software can be difficult to maintain, verify, and relate back to its design models. This part presents a framework for code generation, off-line verification and execution monitoring that addresses this problem through a common model-based workflow. The framework was developed in the context of the CONVINCe project, and some of its tools were developed collaboratively with project partners. For this reason, this part describes the complete pipeline, but it also distinguishes my individual contributions from the surrounding project infrastructure.

My main contributions in this part are:

- The M2C code generator, which automatically generates executable ROS 2 C++ skill packages from high-level ASCXML specifications while preserving the FSM as the decision-making core of the skill.
- The integration of the various tools in a common workflow, which allows developers to easily transition from design to implementation and testing, while maintaining a clear connection between the different stages of development.
- The implementation of two use cases of different complexity and the definition of their verification and monitoring properties, providing an application-level grounding for MOON and validating the framework through off-line verification and on-line monitoring.
- The provision of robotics domain expertise, particularly from the robotic tour-guide use case, to guide monitoring development and provide validation feedback that informed design decisions for the shared tools.

From the perspective of the overall thesis, this part addresses the dependability side of autonomous robotic systems. Its main goal is to show how a common model-based representation can support the full development chain, from specification to execution, while preserving a meaningful connection with verification and runtime monitoring. This perspective is complementary to the planning part that follows: before making a robot more adaptive, it is necessary to ensure that its deliberative software can be generated, analyzed, and monitored in a principled way.

The specific contribution of this part is not limited to the presentation of individual tools. Rather, it lies in showing how a single modeling pipeline can support four development needs

that are often treated separately: specification, code generation, off-line verification, and runtime monitoring.

Taken together, these elements provide a concrete methodology for reducing the gap between the system that is designed, the one that is verified, and the one that is finally executed.

# Chapter 2

## Background

This chapter introduces the main concepts and tools used throughout this part. In particular, it describes the architectural organization of robotic software, the behavioral modeling formalisms adopted in the thesis, the middleware used for implementation, and the logical formalism used for specifying system properties.

### 2.1 Layered Architecture

As discussed in [Siciliano et al., 2008, Chapter 12, Robotic Systems Architectures and Programming], robotic control software is commonly organized in *layers*, arranged as a stack of increasing abstraction from the hardware to the mission-level behavior. In this thesis, this organization is used as a conceptual separation of responsibilities rather than as a strict prescription on processes, packages, or deployment units. Following the terminology of Colledanchise et al. [2021], the three main layers are (Figure 2.1):

- The *control layer* which contains the low-level software for actuation and sensing, developed in close connection with the hardware it operates.
- The *functional layer* which contains software components that expose robot capabilities, either by interfacing with the control layer or by processing data from other components. Examples include navigation, perception, inverse kinematics, and object detection.
- The *deliberative layer* which coordinates functional capabilities in order to achieve task-level objectives, such as reaching a destination, executing a tour, or responding to a low-battery condition.

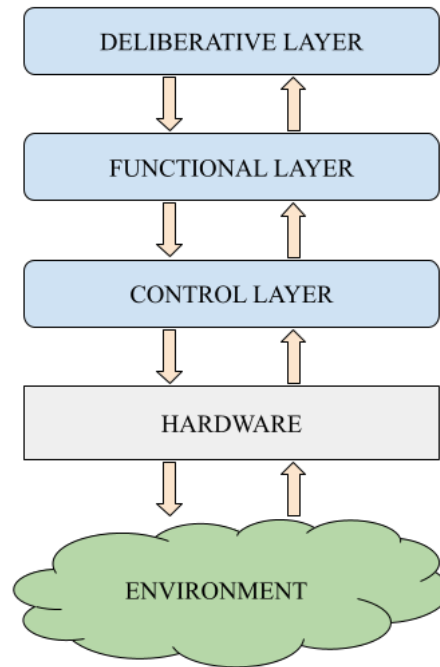


Figure 2.1 Three layers of the robotic architecture (following the terminology of Colledanchise et al. [2021]) and their position with respect to hardware and environment.

The control layer is closest to the *environment*, which includes both the physical robot and the physical space in which it operates. The functional and deliberative layers usually communicate through a middleware, such as ROS 2 (Macenski et al. [2022]) or YARP (Metta et al. [2006]), which provides the infrastructure for distributing computation among modules and machines.

The main focus of this part of the thesis is the deliberative layer, because it is the layer in which robot behavior is structured, refined, and coordinated.

To clarify the terminology used in the following chapters, I recall a few definitions from Ingrand and Ghallab [2017].

**Action.** An *action* is a process that changes the state of the robot and its environment; this includes perception and information gathering actions. An action can be viewed at different levels of abstraction hierarchy. It is primitive at some level and compound at the next level. E.g., `open(door)` is a primitive action at an abstract (task) level and a compound task to be refined, when needed, into a collection of concrete (motion) level actions.

**Command.** A *command* is a primitive action at the lowest level of the hierarchy. Commands can be directly executed by the robot platform.

**Platform.** The *platform* is a collection of devices and sensory-motor functions integrating sensing and closed-loop actuation. Choosing the appropriate command and monitoring that the context remains within a command functioning envelope is part of deliberation. The execution of a command does not need deliberation capabilities, which means that commands are the interface between the deliberative layer and the functional layer and that the platform is the integration of functional and control layers with the robot hardware. For example, `grasp(handle)` is a command which, under appropriate conditions, can be directly executed by a robot platform.

**Plan.** A *plan* is an organized collection of actions.

**Policy.** A *policy* is a universal plan, i.e., a mapping from states to actions.

**Skill.** A *skill* is also an organized collection of actions. It differs from a plan in the following:

- its actions are at a lower level;
- its primitives are commands;
- its control structure can be richer than that of a plan.

A skill can be retrieved from a library of skills (hand-programmed or learned) while a plan is synthesized by a planner.

**Task.** A *task* is the specification of a mission or objectives to be achieved by the robot, in possibly different forms and abstraction levels, e.g., as goal states.

**Event.** An *event* is an occurrence of a change in the state of the robot and its environment; it can be controlled or exogenous, expected or unexpected.

Here, the state of a robot can be defined as:

**State.** A *state* of a robot is defined by 3 different mathematical spaces:

1. The **Configuration Space (C-space)**, representing all valid physical positions and orientations the robot can assume;
2. The **Tangent Space**, containing the velocities and directional derivatives corresponding to the current configuration;
3. The **Belief Space**, representing the probability distribution over the internal states when operating under sensor uncertainty.

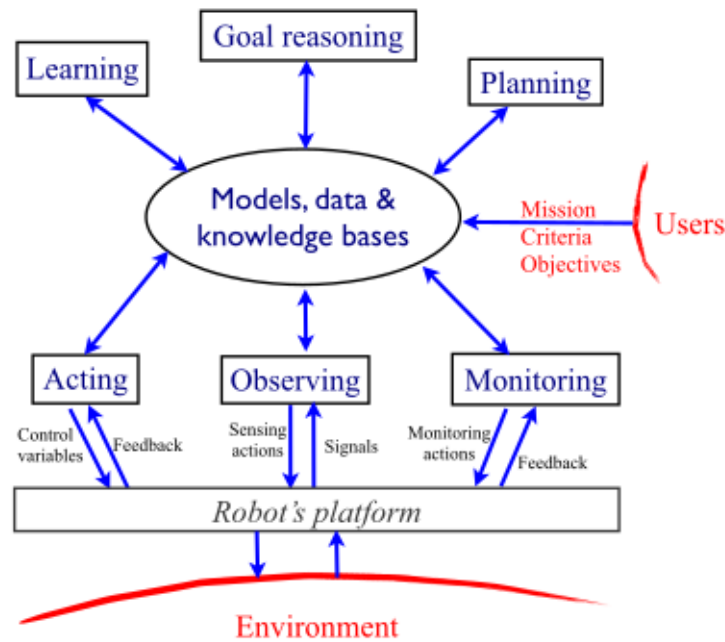


Figure 2.2 Deliberative layer functions — picture from Ingrand and Ghallab [2017]

Still with reference to Ingrand and Ghallab [2017], the deliberative layer can be understood as comprising at least six functions, shown in Figure 2.2:

**Planning.** Planning combines prediction and search to synthesize a task plan in some abstract action space based on predictive models of the environment and feasible actions in order to achieve some purpose.

**Acting.** Acting refines planned actions into commands appropriate for the current context and reacts to events; both refinement and reaction may rely on skills, i.e., a collection of closed-loop functions. A skill processes a sequence (or a continuous stream) of stimulus input from sensors and generates output to actuators, in order to trigger motor forces and control the correct achievement of chosen actions.

**Observing.** The observation detects and recognizes features and relations characterizing the state of the world, as well as events, actions and plans relevant for the task. Observing combines bottom-up processes, from sensors to meaningful data, with top-down activities such as focus mechanisms, sensing actions and planning for information gathering.

**Monitoring.** The monitoring compares what is predicted regarding the robot activities to what is observed in the world. It detects and interprets discrepancies, performs diagnosis and triggers recovery actions when needed.

**Goal reasoning.** Goal reasoning performs monitoring at the mission level. It keeps commitments and goals in perspective; it assesses their relevance from the observed evolutions from new opportunities, constraints or failures; it entails from this assessment whether some commitments should be abandoned, and when and how to update the current goals.

**Learning.** The learning part allows a robot to acquire, adapt and improve through experience the models needed for deliberation. Active learning induces a robot to act deliberately in order to better learn.

Because the deliberative layer is responsible for high-level behavior, it is the layer where behavioral formalisms such as Behavior Trees (BTs) and Finite State Machines (FSMs) are most often used. A comparative study by Ghzouli et al. [2023] shows that both are widely used in robotics applications, with BTs gaining increasing adoption in ROS-based systems due to their flexibility and reactivity. The following sections provide a brief overview of both formalisms.

## 2.2 Behavior Trees

Behavior Trees (BTs) are a widely adopted formalism for modeling behavior in both computer games and robotics (Colledanchise and Ögren [2018]; Sekhavat [2017]). In robotics, they are increasingly used for the design of deliberation policies due to their modularity, composability, and support for reactive behaviors (Iovino et al. [2022]).

They are tree-shaped structures where internal nodes control the flow of execution, and leaf nodes represent actions or conditions. The execution of a BT is driven by a ticking mechanism, where the root node receives ticks at regular intervals and propagates them down the tree according to the semantics of the internal nodes. The status returned by the leaf nodes (success, failure, or running) determines how the internal nodes propagate ticks and how the overall behavior unfolds. The reactivity of the BT allows also the interruption of ongoing actions when the context changes, which is particularly useful in dynamic environments. This is achieved by sending an halt command to the currently running action, which can then perform cleanup operations before terminating.



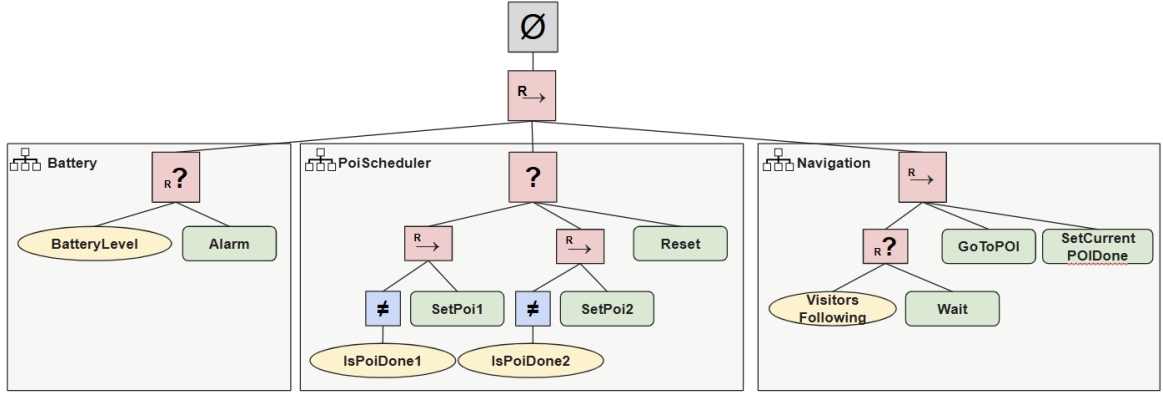


Figure 2.3 Example of a Behavior Tree used in this thesis for the robotic museum guide (simplified). The BT controls the robot to move across different points of interest (POIs). Such behavior is obtained by orchestrating the navigation component (GoToPOI) and a shared blackboard (SetPOI, SetCurrentPOIDone). The BT also checks the level of the battery, raising an alarm if the battery depleted.

An example of a simple BT is shown in Figure 2.3.

Despite their popularity, their semantics is not yet fully standardized. Different implementations and publications often adopt slightly different interpretations of execution and halting behavior (Colledanchise and Natale [2021]). Two recent efforts to formalize BT semantics are presented in Ghiorzi and Tacchella [2024] and Ingrand [2025]. The former aims to define semantics consistent with existing implementations, while the latter proposes a formal translation into FIACRE for verification purposes. However, no existing BT library is fully compliant with a single formal semantics, which makes it difficult to use BTs directly as a universal formal reference.

In this thesis, the semantics adopted is the one from BehaviorTree.CPP version 3.8<sup>1</sup>, as it is one of the most widely used BT libraries in robotics. This choice allows us to remain close to practical implementations while maintaining a clear operational interpretation for monitoring.

All the BT nodes that used in this work can be seen in Table 2.1. They are divided into three main categories: control flow nodes, decorator nodes, and plug-in nodes. In the following will be described all the nodes used in this work.

<sup>1</sup><https://github.com/BehaviorTree/BehaviorTree.CPP>

| Category           | Name                 | Symbol                                                                              |
|--------------------|----------------------|-------------------------------------------------------------------------------------|
| Control Flow Nodes | Root                 |  |
|                    | Sequence             |  |
|                    | Reactive Sequence    |  |
|                    | Sequence With Memory |  |
|                    | Fallback             |  |
|                    | Reactive Fallback    |  |
|                    | Parallel             |  |
|                    | If Then Else         |  |
| Decorator Nodes    | Inverter             |  |
| Plug-in Nodes      | Condition Node       |  |
|                    | Action Node          |  |
|                    | Sub-Tree             |  |

Table 2.1 Behavior Tree nodes used in this work divided by category.

### 2.2.1 Control Flow Nodes

Control flow nodes define the structure of the tree and determine how execution propagates among children.

**Root Node.** The root node is the entry point of the BT. It has a single child and propagates the tick signal to it. The root node returns the same status as its child.

**Sequence Node.** A sequence node ticks its children from left to right. It returns:

- *success* if all children return success,
- *failure* if any child returns failure,
- *running* if a child returns running and no child returns failure.

If a child returns running, then at the following tick the sequence is recovered from that node instead of beginning.

**Reactive Sequence Node.** A reactive sequence node behaves similarly to a sequence node, but it re-ticks all its children at every tick. This allows it to react to changes in the state of previously successful children.

**Sequence With Memory Node.** A sequence with memory node also behaves like a sequence node, ticking its children from left to right, but if it is interrupted it remembers the last child ticked and starts ticking from it in the next tick (only in case of *failure* and *running*). This allows it to avoid re-ticking successful children when interrupted,

which saves time if the computation takes many time, but it does not react to changes in their state.

**Fallback Node.** A fallback node ticks its children from left to right. It returns:

- *success* if any child returns success,
- *failure* if all children return failure,
- *running* if a child returns running and no child returns success.

If a child returns running, then at the following tick the fallback is recovered from that node instead of beginning.

**Reactive Fallback Node.** Like the reactive sequence node, a reactive fallback node re-evaluates all children at every tick, allowing it to react to changes in their state.

**Parallel Node.** A parallel node ticks all its children simultaneously. It has two parameters: a success threshold and a failure threshold. It returns:

- *success* if the number of children that return success is greater than or equal to the success threshold,
- *failure* if the number of children that return failure is greater than or equal to the failure threshold,
- *running* otherwise.

**If-Then-Else Node.** An if-then-else node evaluates a condition and executes one of two branches based on the result. It can have two or three children: the first child is the condition, the second child is the "then" branch, and the optional third child is the "else" branch. It can return:

- *success* if the condition returns success and the "then" branch returns success, or if the condition returns failure and the "else" branch returns success,
- *failure* if the condition returns success and the "then" branch returns failure, or if the condition returns failure and the "else" branch returns failure,
- *running* if the condition returns running, or if the condition returns success and the "then" branch returns running, or if the condition returns failure and the "else" branch returns running.

### 2.2.2 Decorator Nodes

Decorator nodes modify the behavior of their child. The only decorator node used in this thesis is the inverter node.

**Inverter.** The inverter node reverses the result of its child:

- success becomes failure,
- failure becomes success,
- running remains unchanged.

### 2.2.3 Plug-in Nodes

Plug-in nodes are the BT leaves whose behavior is defined outside the BT control-flow semantics. In general, they may interact with the external world, allowing the execution of actions and the evaluation of conditions based on the current state of the system. Their behavior is defined by user-provided code, which can be implemented in various programming languages and can interact with the robot's sensors, actuators, and other software components, typically through a middleware that allows distributing computation across different machines. In this thesis, however, plug-in nodes are used in a more restricted way: they act as ROS 2 adapters that forward BT ticks to the skill layer and return the corresponding status to the BT. They are divided into actions and conditions, which can be implemented as user needs.

**Condition Node.** A condition node evaluates a predicate and returns success if the condition holds, and failure otherwise. It cannot return running.

**Action Node.** An action node performs an operation and returns:

- *success* if the action completes successfully,
- *failure* if the action fails,
- *running* if the action is still in progress.

If halted, the action performs cleanup operations before terminating.

**Sub-Tree Node.** A sub-tree node encapsulates another BT and forwards the tick to its root. It returns the status of the sub-tree. This allows for modularity and reuse of behavior across different parts of the system.

## 2.3 Finite State Machines

Finite State Machines are a well-established formalism in model-based development for representing system behavior. They are widely used in robotics to describe both control logic and abstract models of system components.

In this thesis, FSMs are specified using a subset of the SCXML standard (Barnett et al. [2015]). An SCXML model consists of states and transitions, where transitions may be triggered by events. Events can be internal or external and are processed asynchronously through event queues. They can also have parameters, which are passed along with the event. This naturally supports the concurrent execution of multiple interacting state machines.

To ensure a well-defined and analyzable semantics, hierarchical states and embedded scripting are excluded (for an exhaustive list see Appendix A). These restrictions are necessary because hierarchical state machines often admit multiple interpretations, and scripting introduces arbitrary code that is difficult to reason about formally.

The choice of SCXML is motivated by its status as a W3C standard, providing a well-defined syntax and execution model. This makes it particularly suitable as a common language for integrating code generation, verification, and monitoring.

Here I provide a brief overview of the main SCXML elements used in this thesis, which are `<scxml>`, `<state>`, `<transition>`, `<send>`, `<datamodel>` and `<assign>`. For the full semantics of SCXML, the reader can refer to W3C website<sup>2</sup>.

**<scxml>**. The `<scxml>` tag is the root element of an SCXML document. It defines the overall structure of the state machine and contains all other elements. It can include attributes such as the initial state, the name of the state machine, and other metadata. The `<scxml>` element serves as the container for the entire state machine definition.

**<state>**. The `<state>` tag represents a state in the state machine. It defines the behavior of the system when it is in that state. Each state can have entry and exit actions, which are executed when the state is entered or exited, respectively. They can also have on-event handlers that specify how the system should respond to events while in that state (see `<transition>` tag). In a state, the system is also able to send events, which can trigger transitions in the same or other state machines. This allows for complex interactions and coordination between different parts of the system (see `<send>` tag).

**<transition>**. The `<transition>` tag defines a transition between states. It specifies the source state, the target state, and the event that triggers the transition. Transitions can

---

<sup>2</sup><https://www.w3.org/TR/scxml/>

also include conditions and actions that are executed when the transition occurs. When an event is processed, the state machine checks for transitions that are triggered by that event and evaluates their conditions. If a transition is taken, the system executes the exit actions of the source state, the actions associated with the transition, and then the entry actions of the target state. This allows for dynamic changes in the system's behavior based on events and conditions. The user can read event parameters through the `event.data` variable in transition actions, which allows for more flexible and context-aware behavior.

**<send>.** The `<send>` tag represents an event that can trigger transitions. Events can be generated by the system or received from the environment. They can also have parameters, which are passed along with the event through the `<param>` tag. The `<send>` tag can be used within states to generate events based on the current state of the system, allowing for complex interactions and coordination between different parts of the system. For example, a state can send an event to itself to trigger a transition after a certain delay, or it can send an event to another state machine to coordinate behavior across multiple components.

**<datamodel>.** The `<datamodel>` tag is used to define variables and data structures that can be manipulated by the state machine. This allows for more complex behavior and decision-making based on the current state of the system. Variables defined in the datamodel can be accessed and modified in state entry and exit actions, as well as in transition actions. This enables the state machine to maintain internal state and make decisions based on that state, which is essential for modeling complex behaviors in robotics applications. Variables values can be set through the `<assign>` tag.

**<assign>.** The `<assign>` tag is used to assign values to variables defined in the datamodel. This allows the state machine to update its internal state based on events and conditions. The `<assign>` tag can be used within state entry and exit actions, as well as in transition actions, to modify the values of variables and influence the behavior of the state machine. For example, a transition action might use an `<assign>` tag to update a variable that tracks the number of times a certain event has occurred, which can then be used in conditions for future transitions.

## 2.4 Middleware

A middleware is a software layer that enables communication and coordination among system components. In robotics, middleware provides the infrastructure required to integrate sensors, actuators, and control algorithms. This thesis uses ROS 2 (Macenski et al. [2022]), which is currently one of the most widely adopted middleware platforms in robotics. ROS 2 support for various communication paradigms allows for flexible and efficient integration of various components, enabling the developer to distribute its software across multiple nodes without worrying about the underlying communication details. It provides three types of interfaces: topics, services, and actions.

**Topics.** Topics are a publish–subscribe communication mechanism that allows nodes to exchange messages asynchronously. A node can publish messages to a topic, and other nodes can subscribe to that topic to receive the messages. This decouples the sender and receiver, allowing for flexible communication patterns. Topics are typically used for streaming data, such as sensor readings or control commands.

**Services.** Services provide a synchronous request–response communication mechanism. A node can offer a service, and other nodes can call that service by sending a request and waiting for a response. Services are typically used for operations that require immediate feedback or that need to be executed atomically, such as querying the state of a component or performing a specific action.

**Actions.** Actions are an extension of services that allow for asynchronous request–response interactions with feedback. An action server can handle long-running operations that may provide feedback during execution and can be preempted if necessary. Clients can send goals to an action server and receive updates on the progress of the goal, as well as the final result once the goal is completed. Actions are typically used for tasks that may take a significant amount of time to complete, such as navigation or manipulation tasks.

## 2.5 Past Metric Temporal Logic

To specify system requirements and behavioral properties, I will use past Metric Temporal Logic (past MTL). past MTL (Brzoska [1993]) is a fragment of Metric Temporal Logic (Alur and Henzinger [1993]; Koymans [1990]) that includes only past temporal operators. In this thesis, formulas are defined as follows:

$$\varphi := \top \mid p \mid \neg\alpha \mid \alpha \wedge \beta \mid \alpha \mathbf{S}[a : b] \beta \mid \mathbf{P}[a : b] \alpha \mid \mathbf{H}[a : b] \alpha$$

The operators are defined as follows:

- $\top$  is the tautology, which is always true.
- $p$  is a predicate that can be evaluated on the current state of the system.
- $\neg\alpha$  is the negation of  $\alpha$ .
- $\alpha \wedge \beta$  is the conjunction of  $\alpha$  and  $\beta$ .
- $\alpha \vee \beta$  is the disjunction of  $\alpha$  and  $\beta$ .
- $\alpha \mathbf{S}[a : b] \beta$  (*since* operator) means that  $\beta$  held at some time in the interval  $[t - b, t - a]$ , and that  $\alpha$  has held continuously since then.
- $\mathbf{P}[a : b] \alpha$  (*past eventually* operator) means that  $\alpha$  held at some time in the interval  $[t - b, t - a]$ . It can be derived also from the *since* operator as  $\mathbf{P}[a : b] \alpha := \top \mathbf{S}[a : b] \alpha$ .
- $\mathbf{H}[a : b] \alpha$  (*historically* operator) means that  $\alpha$  has held continuously in the interval  $[t - b, t - a]$ . it can be derived from the *past eventually* operator as  $\mathbf{H}[a : b] \alpha := \neg \mathbf{P}[a : b] \neg\alpha$ .

For each bounded operator, if one of the bounds is missing, it is assumed to be infinite. For example,  $\alpha \mathbf{S}[a : ] \beta$  means that  $\beta$  held at some time in the interval  $(-\infty, t - a]$ , and that  $\alpha$  has held continuously since then.

Formulas are interpreted over timed traces. Given a trace  $w$  and a time  $t$ , the satisfaction relation  $w, t \models \varphi$  is defined in the standard way.

These operators allow expressing temporal properties over system executions in a concise and formal manner.

### 2.5.1 Patterns

Following the approach proposed in Ulus [2019b], pre-defined patterns can be used to write the logical formulae. These patterns can be used as building blocks to express more complex properties, and they can be easily instantiated with specific events and conditions relevant to the system being monitored. They exploits the expressiveness of past MTL to capture common temporal properties, such as:



- *Absence* of an event  $e$  in the last  $t$  time units:  $\neg \mathbf{P}[0 : t] e$ .
- *Existence* of an event  $e$  in the last  $t$  time units:  $\mathbf{P}[0 : t] e$ .
- *Universality* of a condition  $p$  in the last  $t$  time units:  $\mathbf{H}[0 : t] p$ .
- *Response* of an event  $e_2$  to an event  $e_1$  within  $t$  time units:  $\mathbf{H}[0 : t] (e_1 \rightarrow \mathbf{P}[0 : t] e_2)$ .

Moreover, each property defined above can be bounded or unbounded, allowing for more flexibility in expressing temporal requirements. The bounded version can either be before an event or between two events. For more details on the patterns and their formal definitions, the reader can refer to Ulus [2019b].

# Chapter 3

## Related Work

In recent years, many approaches have been proposed for the design, verification, and monitoring of robotic control software (Dal Zilio et al. [2023]). However, most of them focus only on one of these aspects, and only a few attempt to integrate them within a single workflow.

A particularly close reference point is Dal Zilio et al. [2023], which combines code generation, validation, and verification of robotic software within a unified toolchain. That framework starts from a domain-specific language (DSL), derives an implementation from the high-level specification, and supports both model checking and on-line monitoring.

However, compared to the framework presented in this thesis, their approach executes the resulting formal model directly in place of a traditional implementation, using it as the runtime artifact to enable verification and monitoring. By contrast, the workflow adopted here starts from a model, generates an implementation, and then applies verification and monitoring techniques to the generated system. Moreover, their framework targets ROS 1, while the present work supports ROS 2, which is currently the most widely adopted middleware in robotics research.

The following sections review the state of the art in code generation, model checking, and runtime monitoring, focusing on the approaches most relevant to this thesis.

### 3.1 Code Generation

Model-based development and code generation, more generally referred to as Model-Driven Engineering (MDE) (Bézivin [2005]), are well-established approaches in software engineering and have been widely adopted in robotics. The main idea is to describe the relevant parts of the system through models and automatically derive executable artifacts from them.

In robotics this is especially attractive because software systems are heterogeneous: they combine deliberative logic, middleware interfaces, sensors, actuators, navigation components, and platform-specific deployment constraints. Code generation can reduce handwritten boilerplate, make architectural assumptions explicit, and help developers maintain a connection between a design-level description and the executed system. However, the benefit depends strongly on what is preserved after generation. If the generated code is disconnected from the models used for analysis, then code generation alone does not solve the verification and monitoring problem.

### 3.1.1 Model-driven robotics ecosystems

The survey by de Araújo Silva et al. [2021] shows that model-driven robotics has produced many specialized languages and toolchains for architecture modeling, component composition, deployment, and code generation. This confirms the relevance of MDE for robotics, but also highlights a recurring fragmentation problem: different tools often use different modeling languages, target different middleware assumptions, and provide different levels of behavioral semantics. For the contribution of this thesis, this point is central. M2C is not intended as a general-purpose robotics DSL or as a complete replacement for existing modeling ecosystems. Instead, it focuses on executable deliberation-layer skills and uses ASCXML, a ROS-flavoured extension of SCXML, so that the same behavioral artifact can be used in the code-generation path and remain meaningful for later verification and monitoring.

Several established robotics MDE ecosystems concentrate on component interfaces, system assembly, and deployment. The SmartMDSD toolchain (Stampfer et al. [2016]), for example, combines component-based software engineering with domain-specific languages and generator-based development. It separates the concerns of component developers, system builders, and domain experts and generates platform-specific artifacts from platform-independent component and system models. This separation improves reuse and supports additional design-time analyses, including resource and performance analysis. Its primary unit of abstraction, however, is the component and its composition within a system, rather than a standard executable state-machine model shared with runtime conformance monitoring.

BRIDE (Bubeck et al. [2014]) follows a related role-based philosophy. It models capabilities, coordinators, and system configurations and uses model-to-code transformations to produce framework-specific artifacts, including C++ code for ROS. Its emphasis is on reducing integration effort and separating robot-domain logic from the middleware frame-

work. M2C addresses a more specific problem: generating the complete ROS 2 wrapper of a deliberative skill while retaining its ASCXML state machine as the explicit execution core.

The work of Ringert et al. provides another important comparison point. It combines component-and-connector architecture models with automata-based component behavior (Ringert et al. [2014]), and its generator-composition approach allows different behavior languages and target platforms to be integrated modularly (Ringert et al. [2015]). This line of work demonstrates that robotics code generation must accommodate heterogeneous modeling languages rather than assume a single universal formalism. Its contribution is chiefly the composition of architecture and code generators; in contrast, M2C fixes a narrower ASCXML-to-ROS 2 translation so that the generated skill can participate in a specific verification and monitoring pipeline.

One of the most relevant recent ecosystem-level efforts is the work of the RobMoSys consortium (Schlegel et al. [2021]), which promotes separation of roles, component composition, and model-driven development as foundations for a robotics software ecosystem. RobMoSys is important because it addresses the engineering scale of robotic systems and recognizes that different stakeholders need different modeling views. The framework in this thesis is compatible with this architectural view, but it has a narrower and more formal focus: it targets the behavioral models of BT plugins, skills, and components, and it requires those models to be executable or translatable for verification. In other words, RobMoSys motivates the separation between task, skill, and functional concerns, while M2C addresses the concrete problem of turning an ASCXML skill model into a ROS 2 executable without losing the model needed by the rest of the assurance pipeline.

Papyrus for Robotics instantiates this approach in an Eclipse- and UML-based environment and generates ROS 2 artifacts from structural and behavioral models (Radermacher et al. [2021]). It supports component definitions, system assembly, skill and high-level behavior modeling, and safety-related information. This makes it broader than M2C and particularly strong for architecture-level, multi-stakeholder development. The distinction is again one of scope and semantic role: Papyrus for Robotics generates a mixture of implementation skeletons, configuration, build files, and behavior artifacts, whereas M2C specifically compiles an ASCXML skill protocol into a runnable package and preserves that protocol as the reference used by off-line and on-line analyses.

### 3.1.2 Generation from behavioral and formal models

Closer to M2C are approaches that generate robot control software from state machines or other formal behavioral models. SMACHA (Ridge et al. [2017]) uses compact YAML scripts, reusable Jinja2 templates, and meta-scripting to generate ROS SMACH state-machine programs. It is relevant because it treats code generation as a way to make task-level state machines easier to assemble and reuse, and because its template mechanism can in principle target different languages and state-machine frameworks. However, the generated artifact is tied to the semantics of the selected runtime framework, and the source script is not intended to serve as a formal model for both statistical verification and runtime conformance checking.

Godart et al. [2017] generate real-time robot control software from hierarchical state machines and hardware configurations modeled in SysML. Their workflow exports the behavioral model to SCXML before generating control code, making it especially relevant to the representation used in this thesis. The approach separates generated coordination logic from reusable, manually implemented primitive device commands. Its target is hardware-oriented real-time control, whereas M2C uses an SCXML-derived skill model to generate ROS 2 communication adapters at the deliberation layer and integrates the same model with system-level analysis.

A particularly close formal-methods approach is presented by Wang et al. [2019]. The authors model a robotic system as a network of timed automata, verify safety requirements, and automatically generate executable C++ code with mappings to ROS instructions. This work shares two important objectives with M2C: executable software is derived from the analyzed behavioral model, and middleware interactions are handled by the generator rather than reimplemented manually. The main differences are that their workflow targets ROS 1 and timed-automata models for a specific verification-and-generation chain, while the present framework targets ROS 2 skills and reuses the ASCXML artifact not only for off-line verification but also for property and model-conformance monitoring after deployment.

More recently, Onozawa et al. [2023] use model-based development to semi-generate C code for ROS 2-based autonomous-driving software. Their work is close in its ROS 2 orientation, but it targets executable software in a specific autonomous-driving context and does not focus on preserving a shared behavioral model for off-line statistical verification and on-line monitoring. Taken together, these works show a spectrum between generating reusable software skeletons, generating framework-specific state-machine programs, and compiling verified formal behavior into executable control code. M2C lies closest to the last category, but uses a standardized state-machine interchange language and makes post-generation monitoring part of the intended workflow.

### 3.1.3 Directly executable behavior models

Another important line of work in robotics adopts BTs as the main representation of deliberative behavior. A recent comparative study by Ghzouli et al. [2023] shows that both BTs and state machines are extensively used due to their modularity and reactivity in robotics applications (Colledanchise and Ögren [2018]; Iovino et al. [2022]), with BTs gaining increasing adoption in ROS-based systems. BT surveys and empirical studies are particularly relevant to this thesis because they show why BTs are attractive for deliberation: their hierarchical structure supports reuse, reactivity, and readable task decomposition. Even though code is not necessarily generated in the case of BTs, they remain relevant because they are often executed directly by the software, thereby reducing the distance between behavioral model and runtime artifact. However, BT semantics is not yet fully standardized across libraries and frameworks, although recent work has made progress in this direction (Ghiorzi and Tacchella [2024]; Ingrand [2025]). For this reason, BTs are used here at the architectural level, while ASCXML/SCXML serves as the common representation for the lower-level skill behavior that must be generated, verified, and monitored. This division is deliberate: the BT specifies how skills are orchestrated, while the ASCXML model specifies the execution protocol of each skill and its ROS 2 interactions. M2C therefore does not generate an entire deliberative architecture from scratch; it generates the executable skill packages that let BT leaves communicate with the rest of the robotic system through well-defined ROS 2 interfaces.

The formal toolchain of Dal Zilio et al. [2023] offers a second example of direct model execution. It synthesizes formal Fiacre models from component specifications and executes those models in place of conventional component implementations, thereby making the runtime faithful to the verified semantics. This is a strong solution to the model–implementation consistency problem. M2C instead retains a conventional ROS 2 deployment artifact: it generates C++ middleware integration around an executable state machine and then checks the resulting system at runtime. The trade-off is that a generated implementation still introduces an implementation boundary, but it can be deployed and integrated as an ordinary ROS 2 package and monitored against its source model.

### 3.1.4 Positioning of M2C

The closest robotics code-generation approaches can be compared along four dimensions: the abstraction from which generation starts, the artifact that is produced, the middleware targeted by generation, and the role retained by the model after deployment. Architecture-

oriented environments such as SmartMDS, BRIDE, MontiArcAutomaton, and Papyrus for Robotics provide broad support for component definition, composition, and integration. Behavior-oriented generators such as SMACHA and the SysML-to-SCXML workflow focus more directly on control flow, but do not connect the source behavior to both off-line system verification and on-line model monitoring. The timed-automata approach of Wang et al. [2019] connects verification and generation closely, but does not extend the connection to runtime monitoring, while Dal Zilio et al. [2023] avoids generation of a conventional implementation by executing the formal model directly. Other approaches focus on verification of existing ROS 2 systems, such as Dust et al. [2024], but reconstruct models from implementation traces instead of using the model as the source from which the implementation is generated. M2C occupies a different point in this design space: it generates deployable ROS 2 C++ packages from ASCXML models, while keeping those models available for the rest of the toolchain. The contribution is therefore not code generation in isolation, nor a new general-purpose robotics modeling ecosystem. It is the preservation of a behavioral artifact across three uses: producing the implementation, constructing the off-line verification model, and checking temporal properties and behavioral conformance during execution.

## 3.2 Model Checking

Model checking (Clarke [1997]) is a formal verification technique used to determine whether a system model satisfies a given property, typically expressed in temporal logic. Classical model checking relies on the exhaustive exploration of the system state space, ensuring that all possible executions are analyzed. While this approach provides strong correctness guarantees, it suffers from the well-known state explosion problem, which limits its applicability to complex systems.

This limitation is particularly relevant in robotics, where systems are inherently concurrent, interact with dynamic environments, and often exhibit nondeterministic or stochastic behavior. As a result, applying classical model checking directly to realistic robotic systems is often infeasible.

To address these challenges, statistical model checking (SMC) (Legay et al. [2019]) has been proposed as an alternative approach. Instead of exploring all possible executions, SMC relies on simulation-based sampling to estimate the probability that a property holds. By analyzing a finite number of execution traces, SMC can provide probabilistic guarantees about system correctness, making it applicable to larger and more complex models. This approach has been implemented in tools such as UPPAAL SMC (David et al. [2011]) and

PRISM (Kwiatkowska et al. [2011]), and has proven effective for systems with stochastic and real-time characteristics.

In model-driven approaches, verification is typically performed on formal models derived from high-level specifications. Here system behavior is described using SCXML, which provides a standardized representation of finite state machines. This enables the automatic derivation of formal models suitable for statistical verification, ensuring that properties can be analyzed consistently with the structure of the generated implementation.

A recent robotics approach is presented in Dust et al. [2024], where formal models are extracted from ROS 2 execution traces and then used for pattern-based verification. Unlike the workflow adopted here, that approach is bottom-up: models are reconstructed from implementations rather than used to generate them. Moreover, while Dust et al. [2024] relies on pattern-based specifications, we adopt past Metric Temporal Logic, which provides a more expressive and compositional framework for specifying temporal properties and enables a tighter integration between off-line verification and runtime monitoring.

### 3.3 Monitoring

Model checking and runtime verification are often considered complementary techniques, where the former provides design-time guarantees and the latter evaluates properties over system executions. Runtime verification has been widely studied as a lightweight alternative to full model checking (Leucker and Schallhart [2009]), and is particularly relevant in robotics, where systems must operate in dynamic and uncertain environments.

Execution monitoring is defined as “*the continuous real-time process of observing a system, detecting anomalies, and identifying deviations from expected behavior*” (Pettersson [2005]). In robotics, this corresponds to checking whether system requirements are satisfied during execution. Monitoring improves reliability and safety, but introduces computational overhead and engineering complexity.

Monitoring tasks are typically divided into fault detection, fault isolation, and fault identification (Gertler [2017]). In this thesis I focus mainly on fault detection.

Most recent monitoring approaches in robotics are built on ROS or ROS 2. A representative example is Ferrando et al. [2020], extended to ROS 2 in Saadat et al. [2024], which uses past MTL formulas evaluated by the Reelay engine (Ulus [2026]). A similar approach is used by NASA in Perez et al. [2022], where requirements are specified using FRET and monitored using Ogma (Perez [2023]) and Copilot (Perez et al. [2020]). These approaches where they combine temporal logic specifications with runtime observation are conceptually similar



to the one in Ferrando et al. [2020], but they differ in the choice of specification language and monitoring engine. The present framework adopts past MTL for its expressiveness and compositionality, and integrates it with the SCXML-based model representation to enable a seamless transition from off-line verification to runtime monitoring.

Beyond temporal-logic-based monitoring, alternative paradigms have also been explored. For instance, the TeSSLa-ROS-Bridge (Begemann et al. [2023]) enables runtime verification using the temporal stream language TeSSLa (Convent et al. [2018]), where properties are evaluated through stream transformations rather than logical formula evaluation. This highlights the diversity of runtime verification techniques in robotics. Different from the past MTL-based approach, TeSSLa focuses on processing and analyzing event streams, which can be more efficient for certain types of properties but may require a different modeling approach. Compared to the present framework, which uses a unified SCXML-based representation for both code generation and monitoring, the TeSSLa-ROS-Bridge operates as a separate layer that processes ROS messages, potentially leading to a less integrated workflow.

A different approach is proposed in Lim and Noh [2024], where fault detection is based on redundancy. Multiple components perform the same task, and their outputs are compared to detect inconsistencies. While this improves robustness, it increases resource usage and system complexity. A related idea appears in the present framework, where model monitoring effectively compares implementation behavior against a reference model.

## 3.4 Summary

The literature reviewed above shows that the individual building blocks required to implement our approach already exist in different forms. There are mature studies on model-driven engineering, a broad body of work on design-time verification, and several effective approaches to runtime monitoring in robotics. However, what is still largely missing is a development pipeline in which these elements are not merely co-located, but are linked through a shared behavioral representation and a coherent semantic interpretation.

This gap is especially relevant for deliberation software. In practice, BTs and FSMs are often used to guide implementation, but the artifacts employed for code execution, model checking, and runtime monitoring are usually different, partially manual, or only loosely connected. As a result, developers may verify one artifact, execute another, and monitor only a subset of the final system behavior.

The contribution of this thesis is to reduce this fragmentation. In particular, the proposed framework uses a common model-based representation to support code generation, off-line

---

statistical verification, and runtime monitoring in a unified ROS 2-oriented workflow. The key novelty is therefore not only the presence of these tools, but the semantic continuity between them, which helps limit mismatches between design-time assumptions and runtime behavior.

# Chapter 4

## Framework

This chapter presents the framework used in this part of the thesis. The framework connects model-based development, off-line verification, code generation, and runtime monitoring around a common representation of deliberative robotic software. Starting from BTs, FSMs, and system-level properties, the same modeling effort can be used to analyze the design, generate executable ROS 2 code, and monitor whether the running implementation remains consistent with the analyzed artifacts. The main components of the framework are the following tools:

- A **modeling language** based on BTs and FSMs, which allows the user to design the behavior of the robot in a modular and hierarchical way.
- A **code generator** that generates code from the models specified in the modeling language, which can be executed on the robot.
- A **statistical model checker** that verifies properties of the models, which can be used to ensure that the models satisfy the desired properties before generating code.
- An **execution monitor** that checks if the properties verified on the models hold at runtime on the corresponding implementations, which can be used to detect and handle errors at runtime.

These components have also been used in experimental sections to model complete systems, and demonstrate how the framework supports not only on-line but also off-line verification.

Applicability is demonstrated on two use cases of different complexity.

## 4.1 Using the Framework

The objective of the framework is to preserve semantic continuity across design, verification, code generation, and runtime monitoring, so that the same behavioral specification can support all phases of development.

As mentioned in Section 2.1 and reprised in Section 4.2, we assume a three-layer architecture instantiated through four modeling elements (Figure 4.1): task plan, plugins, skills, and components. The task plan is represented as a Behavior Tree, whose leaves correspond to action and condition plugins. These plugins interact with the skill layer by forwarding ticks and receiving status responses. The skills implement the high-level robot capabilities and orchestrate the components, which in turn interact with the modeled environment. From

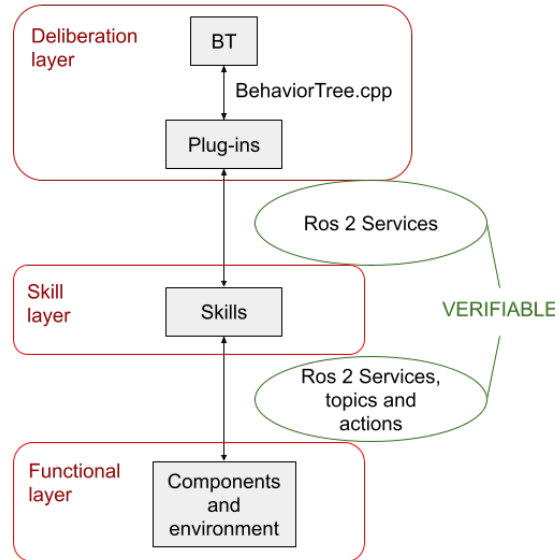


Figure 4.1 Architecture used in this part of the thesis. The deliberation layer is composed of the BT and its plugins, the skill layer contains FSM-based skill models and generated skill implementations, and the functional layer contains the components interacting with the environment. The green marks indicate the communications observed by the monitoring infrastructure.

the methodological viewpoint, the same modeling effort, as will be described in Section 4.3, supports two complementary branches of the framework, as summarized in Figure 4.2. The first branch is concerned with off-line verification (Section 4.5.2). Since our models include custom tags to represent ROS 2 interfaces, they must first be translated into standard SCXML before they can be analyzed by SCAN. This translation is performed by AS2FM, described

in Section 4.5.1, which produces a set of SCXML files compliant with the standard and suitable for statistical model checking. The second branch is concerned with execution and runtime monitoring (Section 4.5.3). The Behavior Tree can be executed directly through BehaviorTree.CPP (Section 2.2), while the plugins and components are provided as executable ROS 2 nodes. For the skills, instead, an implementation must be generated from the corresponding FSM. This step is carried out by `model2code` (Section 4.4), which translates the skill model into executable ROS 2 C++ code. The resulting implementation can then be monitored at runtime through MOON, which checks either whether relevant temporal properties hold on the observed communications or whether the generated skill behaves consistently with its reference FSM.

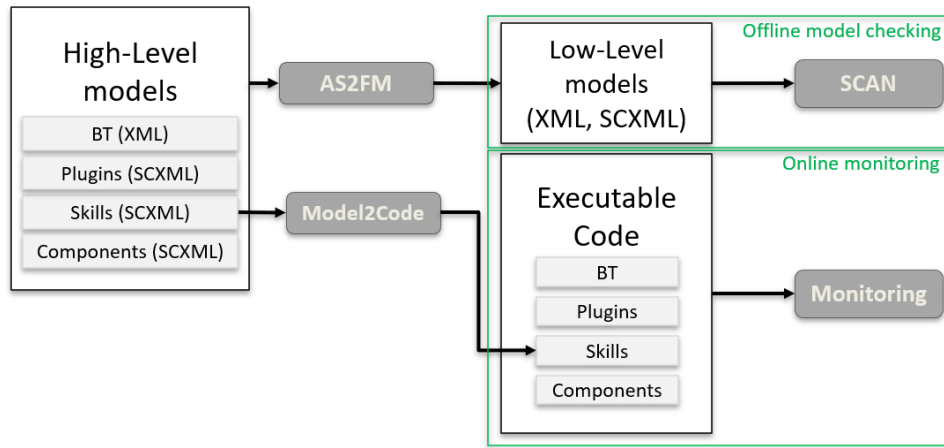


Figure 4.2 Overview of the various framework components and their interactions: on the left are the common modelling inputs, while on the right are the outputs produced for verification, execution, and monitoring.

This organization gives rise to a workflow in which the same model is reused for multiple purposes: it is translated into plain SCXML for verification, compiled into executable code for deployment, and exploited again to derive runtime monitors. As a result, off-line verification and on-line monitoring are not disconnected activities, but two coordinated analyses performed on artifacts that originate from the same behavioral specification.

Finally, system-level requirements are expressed as temporal properties over the monitored events. For each requirement, we proceed in three steps: first, we select the appropriate specification pattern as described in Section 2.5; second, we formulate the requirement in constrained natural language so that it can be mapped unambiguously to the chosen pattern; third, we derive the corresponding past-MTL formula. This process provides the properties that are later verified off-line by SCAN and monitored at runtime by MOON.

## 4.2 Architecture

The general layered view introduced in Section 2.1 is instantiated in this framework as the architecture shown in Figure 4.1. The architecture is organized around four modeling elements: the task plan, the plugins, the skills, and the components. The task plan and plugins realize the deliberation layer, the skills form an intermediate skill layer between deliberation and functional execution, and the components realize the functional layer and the environment. The required properties are also modeled when performing off-line verification, but they are not treated as deployed software layers.

**Task plan.** The task plan is the highest-level behavioral model of the robot. In the framework it is represented as a Behavior Tree, encoded in the BehaviorTree.CPP XML format. The BT defines the control structure of the behavior, but its leaves do not contain arbitrary robot logic.

**Plugins.** Plugins are the executable interface between BT leaves and the rest of the architecture. The framework uses ROS 2 action and condition plugins that forward ticks to the corresponding skill and return the resulting BT status. Keeping this interaction explicit is important for monitoring, because communication between the BT and the rest of the system remains visible instead of being hidden inside node-local code or shared variables.

**Skills.** Skills implement high-level robot capabilities, such as navigation, interaction, scheduling, or battery checking. They are specified as FSMs in ASCXML language (see Appendix A). A skill model can be concrete, when it is compiled into executable ROS 2 code by the code generator, or abstract, when it describes an externally implemented behavior used only for analysis and monitoring.

**Components.** Components implement the lower-level functional capabilities used by the skills. They are ROS 2 nodes exposing topics, services, or actions, and they may wrap perception, navigation, actuation, blackboard storage, or other functional modules. They are also responsible to interact with the hardware and the environment. Also components are specified in ASCXML language.

**Environment and properties.** For off-line verification, the architecture is closed by abstract models of the environment and by temporal properties over the events exchanged by the system. These elements make it possible to analyze the interaction between the task plan, skills, components, and environment before deployment.

This organization deliberately separates the control logic of the BT from the computation performed by skills and components. Alternative BT implementations often rely on shared blackboards or arbitrary code inside leaves; this is convenient for implementation, but it hides communication and computation steps that are important for runtime observation. In our architecture, shared information is represented explicitly through components, such as a blackboard component implemented as a ROS 2 node. This introduces some communication overhead, because ticks and statuses must cross the plugin-skill boundary, but it improves modularity and makes the relevant interactions observable by MOON.

The same decomposition is reflected in the modeling workflow summarized in Figure 4.3. Developers start from informal mental models of the task plan, skills, functional components, environment, and required properties. These models are then encoded in the formats accepted by the toolchain: BT XML for task plans, ASCXML for skills and component abstractions, and past-MTL formulas for properties. The resulting set of models is the operational model used by the verification, code generation, and monitoring tools.

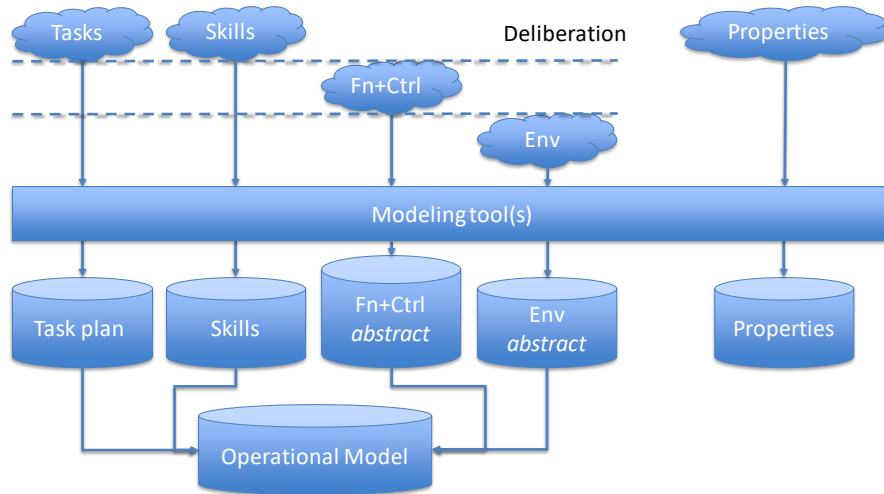


Figure 4.3 From mental models and system properties to formal models.

The architecture is also compatible with the separation of functions promoted by RobMoSys (Schlegel et al. [2021]). There is no one-to-one correspondence between the terminology used here and the RobMoSys levels, but the mapping is natural: mission and task concerns are represented by the task plan, skill concerns by the FSM-based skill layer, and service or function concerns by the components of the functional layer. The operating system and middleware are treated as infrastructure that connects these elements rather than as behavioral models in their own right.

## 4.3 Modelling Language

Within the CONVINCe consortium we have developed a high level specification and a modeling language that we called RoAML to specify all the components of the system and some properties over them. The specification is based on the idea of having a main file that references all the models of the system, including the BT, the FSMs and the properties. The main advantage of using a modeling language is that it allows us to have a better overview of the system without having to read the code, and it also allows us to have a better understanding of the system, by reasoning on the models of its components, as opposed to their implementation. Moreover, it also enables the integration with the previously mentioned tools, since the models can be used as a common language between these tools and the implementation, as well as a better modularity and re-usability of the models, since we can easily replace one model with another one without having to change the rest of the system. For example, we can easily replace the BT with another one that has the same nodes but different structure, or we can easily replace the FSMs with other ones that have the same events but different states and transitions. This becomes useful if some components are changed, for example a different grasping pipeline which requires different checks. This allows us to have a better flexibility in the design of the system and it also allows us to have a better adaptability to changes in the requirements or in the environment. We assume that all the communications that we monitor occurs through the ROS 2middleware. This file have four main sections, which are the following:

**Parameters.** In this section the user can specify parameters that can affect the execution of the model when it is executed by the model checker. they parameters can be used to configure the behavior of the model checker and to specify some assumptions about the system. The parameters that we currently support are the following:

- **max\_time:** the maximum time that the model checker will run before halting the execution. This parameter is useful to prevent the model checker from running indefinitely in case of a model that has a large state space or that has a lot of non-determinism.
- **bt\_tick\_rate:** the tick rate of the BT, which is the rate at which the BT will be ticked during the execution of the model. This parameter is useful to control the execution of the BT and to ensure that it is executed at a reasonable rate, which can be important for the performance of the model checker and for the realism of the model.



- **bt\_tick\_if\_not\_running:** a boolean parameter that specifies whether the BT should be ticked even if it is not running. This parameter is useful to control the execution of the BT and to ensure that it is ticked even if it is not running, which can be important for the performance of the model checker and for the realism of the model. For example, if the BT is not running, it might be because it is waiting for some event to occur, and in this case it might be useful to tick it anyway to ensure that it is executed at a reasonable rate.

**Behavior Tree.** In this section the user can specify the BT that defines the behavior of the robot. The BT is specified through a Behaviortree.CPP compliant XML file. It also contains a list of plug-ins specified in ASCXML format that specify how the BT interfaces with the skills and components (including the communication protocols involved when talking to ROS 2 nodes, e.g., actions, services or topics),

**Skills and Components Models.** In this section the user can specify the models of the skills, components and environment that are used in the BT. These models are also specified in the format of the ASCXML language.

**Properties.** In this section the user can specify the properties that should be verified on the complete model. These properties are specified through an XML file that contains a list of past-MTL formulae over the events exchanged by the model.

To specify the various state machine we have defined the ASCXML language, which is based on the SCXML standard, but it has some extensions to support the specific needs of our framework. In particular, we added the following features:

- the ability to specify ROS 2 interfaces
- the ability to specify timers

For a detailed description of the ASCXML language and its features, see Appendix A.

### 4.3.1 Semantic Continuity Across the Framework

A central point of the proposed framework is that the same behavioral description should remain meaningful across the different stages of development. In many robotics workflows, the model used during design is only an informal guide for the implementation, while verification and monitoring are applied to further derived artifacts with partially different

assumptions. This weakens the connection between evidence gathered at design time and guarantees that can be trusted at runtime.

In our approach, semantic continuity is pursued by keeping BTs, FSMs, and RoAML specifications at the core of the entire pipeline. The same models are used to describe the system, to derive SCXML artifacts for verification, to generate executable skills, and to configure or derive runtime monitors. This does not eliminate all abstraction gaps, but it significantly reduces the risk that verification, code generation, and monitoring refer to incompatible views of the same system.

## 4.4 Code Generation

Although SCXML state machines can be compiled and executed, they cannot directly interface with the functional layer, because communication details such as topics, services, actions, message types, and node configuration are external to standard SCXML. This creates a gap between the behavioral model and the concrete software artifact that must interact with the rest of the ROS 2 system. Model2code<sup>1</sup> (M2C) is the tool that bridges this gap. M2C translates an ASCXML skill model into an executable ROS 2 C++ package. Conceptually, the translation preserves the FSM as the decision-making core of the skill and generates the surrounding software infrastructure required to run it as a ROS 2 node. In other words, the state machine remains responsible for deciding *when* an event should be consumed or produced, while the generated code is responsible for translating between ASCXML events and concrete ROS 2 communications.

The workflow can be understood in three broad stages.

**Model Translation.** The first stage translates the high-level ASCXML into plain SCXML.

This step rewrites custom tags and communication annotations into a more regular state-machine representation that can be used by downstream tools and by the code generator itself.

**Metadata and Interface Extraction.** Once the translated model is available, M2C identifies the key information needed for generation. It extracts the identity of the skill, recognizes the events used in the state machine, and links those events to the ROS 2 interfaces they represent. In this way, the tool understands whether the skill needs to react to service requests, publish or subscribe to topics, or interact through actions.

---

<sup>1</sup><https://github.com/convince-project/model2code>

**Code Generation.** Finally, the tool fills a set of templates with the extracted information. It generates the source files, headers, build files, and package metadata needed to obtain a compilable ROS 2 package. The exact output depends on the type of skill and on the interfaces used in the model.

The generated package typically includes:

- a translated SCXML file,
- C++ source and header files for the skill,
- a `main.cpp` entry point,
- a `CMakeLists.txt` file,
- a `package.xml` file,
- optional extra files related to the skill datamodel.

#### 4.4.1 How the Generated Model Works

Once generated and executed, the skill behaves as a runtime bridge between the SCXML state machine and the ROS 2 system around it.

At startup, the generated `main.cpp` creates the application, instantiates the generated skill class, and starts the state machine. The generated class also creates the ROS 2 node and sets up the interfaces required by the skill, such as tick and halt services, service clients, topic subscriptions, and action clients.

From that point on, the SCXML model becomes the internal logic of the skill. Incoming ROS 2 requests or messages are translated into SCXML events and submitted to the state machine. The state machine then changes state according to the model and may emit new events that represent outgoing calls or responses.

When the SCXML model requests an interaction with the outside world, the generated C++ code handles that request by calling the corresponding ROS 2 service, publishing or receiving topic data, or interacting with an action server. In other words, ROS 2 communication is handled by the generated wrapper code, while the decision logic remains inside the SCXML model.

This means that, at runtime, the generated package works as an event-driven loop:

- ROS 2 inputs are converted into SCXML events,
- the SCXML state machine processes those events,

- the resulting state transitions trigger ROS 2 outputs or responses.

In this way, the generated model is not just a static artifact: it becomes the executable control logic of the skill, with the generated ROS 2 code acting as the adapter between the abstract state machine and the concrete robotic software interfaces. A schematic view of the generated model is shown in Figure 4.4.

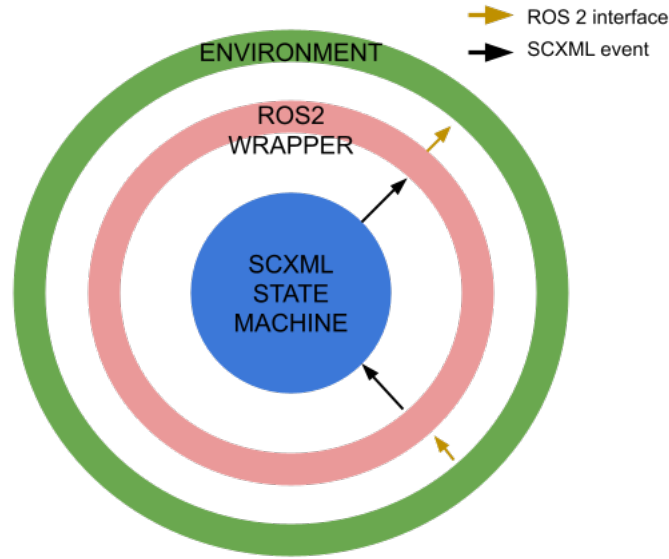


Figure 4.4 Schematic view of the generated model. The SCXML state machine is the core logic of the skill, while the generated ROS 2 code acts as an adapter between the state machine and the environment.

## 4.5 Verification

Both off-line and on-line model verification shares the same models, but in order to be used by the respective tools, the models must be translated into different formats. In the following sections I will describe the translation steps required to use the same models for both off-line verification with SCAN and on-line monitoring with MOON, then I will describe the two verification tools and how they are used in the framework.

### 4.5.1 Model Translation for Verification and Monitoring

While M2C translates skill models into executable ROS 2 code, off-line verification requires a different translation step. In particular, the models used in our framework contain high-

level constructs related to ROS 2 communications and Behavior Tree integration, which are convenient for development but are not directly suitable as input to a formal verification backend or an on-line model monitor. To bridge this gap, we use AS2FM<sup>2</sup> (Autonomous Systems to Formal Model), another tool developed in the CONVINCe project to translate high-level behavioral models into formal artifacts for verification. AS2FM takes RoAML models and translates them into low-level, standard-compliant SCXML by expanding those domain-specific constructs into plain state-machine mechanisms. This includes, for example, the treatment of ROS 2 topics, services, actions, timers, BT ticks, BT responses, and blackboard-related interactions. This translation step is important because it provides a formal representation of the same behavioral model used elsewhere in the toolchain, but in a form that can be consumed by analysis tools. We rely on AS2FM to obtain the plain SCXML models that are then given as input to SCAN for statistical model checking and to MOON for on-line verification.

### 4.5.2 Off-Line Verification

To perform off-line verification we utilize SCAN<sup>3</sup> (StatistiCal ANalyzer) a statistical model checker designed to verify models of asynchronous processes exchanging data over FIFO buffers — a *channel system* in the parlance of Baier and Katoen [2008], where each buffer is a *channel* and data items are *messages*. Moreover, the underlying model features also timed transitions and invariants. SCAN accepts SCXML-specified FSMs running asynchronously (see Section 2.3) which are internally converted to a channel system, where event queues are modeled by channels, and SCXML events are modeled by messages.

The staple feature of SCAN is to sample concrete execution of channel systems to find the probability that a property is satisfied, within specific confidence bounds. In practice, SCAN runs sufficiently many Monte Carlo simulations of the model to prove a statistical result characterized by precision and confidence parameters. To determine the number of tests required, it adopts the Adaptive criterion from Budde et al. [2020] which updates the halting threshold dynamically, based on the test results as they are obtained. Other than providing statistically sound success rate results, SCAN can save the execution traces it produces in terms of events being sent and received by SCXML machines, thus allowing developers to directly relate the execution traces to the specified model and properties.

In particular, SCAN deals with temporal properties expressed in past-MTL (Section 2.5) with super-dense time semantics (Sarjoughian and Sundaramoorthi [2015]) and state-event

<sup>2</sup><https://convince-project.github.io/AS2FM/>

<sup>3</sup><https://github.com/convince-project/scan>

support (Chaki et al. [2004]), which is meant to align it with the monitoring system described later so that the same properties verified on the models can be monitored at runtime on the corresponding implementations. In SCAN, predicates are restricted to the messages sent along the channels corresponding to messages that can be monitored in the real system. We choose not to verify the internal state of the processes, as models might be an abstraction over the actual components in the system, and thus their actual state neither corresponds to some state in the actual component, nor it can be monitored.

### 4.5.3 On-Line Verification

On-line verification is the runtime process of observing a system execution, reconstructing a trace of relevant events and states, and checking this trace against formal or semi-formal requirements in order to detect violations as early as possible.

In this thesis, monitoring is performed through MOON<sup>4</sup> (MONitoring On-line), a runtime monitoring framework developed in the CONVINCe project and presented in Bernagozzi et al. [2025b]. MOON is implemented on top of ROSMonitoring (Ferrando et al. [2020]) and acts as the on-line counterpart of the off-line verification carried out by SCAN. While verification establishes evidence on the model at design time, MOON checks at runtime whether the running ROS 2 system remains consistent with the verified requirements and with the behavior prescribed by the models.

It can provide two different monitors: *property monitors* and *model monitors*. Property monitors check whether the observed behavior satisfies temporal properties over the monitored channels, while model monitors check whether the observed behavior of a skill is compliant with its design-time FSM specification. Both types of monitors are generated from the same RoAML specification, which ensures that the runtime checks are consistent with the design models and with the properties verified off-line by SCAN. In this way, MOON is used to reduce the reality gap between model analysis and actual execution. My contribution to MOON was its application-level grounding rather than the standalone implementation of the tool. I defined monitoring properties for the robotic use cases, provided robotics domain expertise to identify the interactions and failure conditions that the monitor needed to observe, and validated the resulting monitoring workflow. Feedback from this process was used to inform design decisions for MOON and its integration with the other tools in the framework. MOON accepts the same model specification described in Section 4.3 and exploits the information already available in that specification to generate the monitoring

<sup>4</sup><https://github.com/convince-project/MOON>

infrastructure automatically. This is an important design choice, because it reduces the amount of manual configuration required from the developer and preserves a tight connection between design models, generated code, and runtime verification. Its architecture is organized around three dedicated command-line tools, namely `moon_config`, `moon_generator`, and `moon_oracle`.

**moon\_config** `moon_config` is responsible for parsing the RoAML specification and extracting the relevant information to configure the monitor. It identifies the channels to be monitored, the properties to be checked, and the logging options. It also prepares the configuration files needed for the next stage.

**moon\_generator.** `moon_generator` takes the configuration produced by `moon_config` and generates a ROS 2 package that implements the monitoring logic. This includes the necessary nodes to subscribe to the monitored channels, collect the observed data, and forward it to the oracle for evaluation. The generated package can be built and deployed alongside the main system.

**moon\_oracle.** `moon_oracle` is the component responsible for evaluating the observed behavior against the specified requirements. It can be configured to check either temporal properties or model compliance, depending on the monitoring objectives. The oracle receives the observed events and states from the monitor nodes and performs the necessary checks to determine if any violations or inconsistencies are present.

### Property Monitoring

MOON supports property monitoring by generating monitors from temporal requirements over the communications exchanged by the running system. The approach, described in Bernagozzi et al. [2025a], is the following: the monitored communication is first acquired by the instrumentation layer, then forwarded by the monitor to the oracle, which incrementally evaluates the past-MTL formula over the resulting stream of timestamped observations. For topics, this is done by passively subscribing to the channel and logging the incoming messages without altering the original communication. For services and actions, instead, the monitor reconstructs the request–response interaction from built-in ROS 2 service introspection so that the oracle can evaluate properties over both the call and its outcome. This makes MOON consistent with the off-line verification stage, where the same class of properties is checked on the design models. Property monitoring is useful when the objective is to verify explicit requirements over the observed channels, for example that a message is eventually

produced after a given request or that a topic is published with a minimum frequency. MOON evaluates these requirements directly on the messages observed at runtime. In the current implementation, the oracle is launched explicitly through the `moon_oracle` command and can be configured for either dense or discrete time. The monitored events are then sent to the oracle, which evaluates the property on-line.

### **Model Monitoring**

Model monitoring is the feature that most clearly distinguishes MOON from ROSMonitoring. Instead of checking a temporal formula over a trace, MOON checks whether the observed behavior of a skill is compliant with its design-time FSM specification. This is particularly meaningful here because the monitored skills are generated by M2C from the same models used to construct the monitor, so the comparison is performed between an implementation and its own reference model. At the conceptual level, model monitoring relies on executing or simulating a reference version of the design model and comparing its expected evolution with the interactions observed on the running system. The monitored inputs and outputs of the skill are forwarded to the oracle, which checks whether the implementation remains compatible with the behavior allowed by the FSM. Whenever a discrepancy is detected, MOON reports a model-level inconsistency. In this sense, the oracle acts as an on-line reference execution of the design model and continuously checks whether the implementation remains observationally equivalent to it with respect to the monitored interfaces.



# Chapter 5

## Validation of the Framework

The following chapter presents two case studies of increasing complexity, both based on the architectural pattern described in Section 4.2. Together, they show how the framework can be used to design robot behavior, verify that the resulting models satisfy the desired properties through statistical model checking, generate executable code from those models, and monitor at runtime whether the verified properties continue to hold on the implementation.

The first case study is a simple BT that checks the battery level of the robot and raises an alarm if the battery level is low, while the second one is an enhanced version that adds a scheduler for selecting the next Point of Interest (POI) and a navigation sub-tree that checks whether people are following the robot before navigating to the selected POI.

To validate the framework, off-line verification and on-line monitoring were tested separately in order to answer four main questions:

- i. are results of SCAN reliable and can the monitoring find flaws due to the reality gap?
- ii. can the use of both frameworks enhance the reliability of the system?
- iii. is monitoring overhead feasible for real-time applications?
- iv. is model monitoring useful to detect faults in the generated code?

Finally, the results of the whole validation process are discussed in Section 6.

### 5.1 First Case Study: Simple Behavior Tree

The first case study is a simple example based on the BT shown in Figure 5.1. In this scenario, the robot monitors its battery level: if the battery level is above 30%, the BT ticks

a drain-battery action; otherwise, it raises an alarm. This case study is useful for testing the basic functionalities of the framework, such as code generation, model checking, and execution monitoring, and it also helps clarify which properties can be verified on the models and monitored at runtime. The main components of this case study are the following:

- A BT that checks the battery level of the robot, drains the battery when the level is above 30%, and raises an alarm otherwise.
- Three FSMs for the skill layer, which are the following: a battery level skill which returns success if the battery level is higher than 30% and failure otherwise, an alarm skill which sends an alarm message to the alarm component when triggered and a battery drain skill which decreases the battery level when called.
- Two components for the functional layer, which are the following: a battery component which simulates the battery level of the robot and an alarm component which simulates the alarm system of the robot.
- A property that states that if the battery level is low, then the alarm should be raised. This property can be expressed in past-MTL as follows:

$$\mathbf{H}(BatteryLevelLow \implies AlarmRaised)$$

This states that "it is always the case that if the battery level is low, then the alarm is raised".

### 5.1.1 Off-line Verification

To verify the model off-line, SCAN was executed on the system model. The results of the verification are consistent with the expected behavior of the system, which means that the property holds on the model. This is a good indication that the model is correct and that the property is satisfied by the system. However, model verification does not guarantee that the implementation will behave as expected, due to the reality gap between the model and the implementation. This is why property monitoring is needed.

### 5.1.2 On-line Verification

For the on-line verification, the skills were compiled with M2C and the resulting code was executed on the system. The results of the monitoring are consistent with the expected behavior of the system, which means that the property holds at runtime on the implementation.

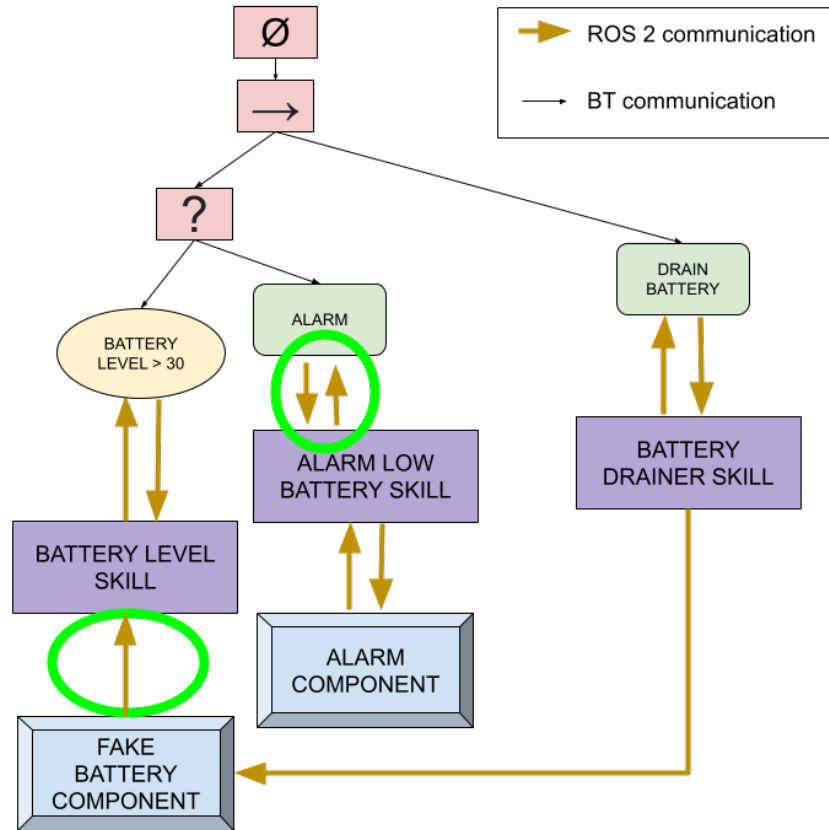


Figure 5.1 A simple BT that checks the battery level, if it is higher than 30% it drains the battery, otherwise it raises an alarm.

This is a good indication that the implementation behaves as expected and that the property is satisfied by the system. Moreover, it allows us to detect any deviation from the expected behavior at runtime, which is crucial for ensuring the reliability of the system, which can happen in more sophisticated systems also subject to hardware malfunctions.

## 5.2 Second Case Study: Enhanced Behavior Tree

Now consider a scenario where a humanoid robot guides visitors on a simple tour. The robot has to navigate to two Points of Interest (POIs), while ensuring that visitors are following it and that its battery level is sufficient. To simplify the simulation, speech-based interaction is not considered. The robot should also be able to raise an alarm if the battery level is low. This scenario is representative of a real robotic application, and it allows us to test the capabilities of our framework in handling more complex systems, and it gives an example of the properties that can be verified and monitored on more complex systems.

Starting from the BT of Figure 5.1, the drain-battery action has been replaced with two sub-BTs: “PoiScheduler”, which selects the appropriate POI and resets the tour once all POIs have been completed, and “Navigation”, which checks whether visitors are following the robot and, if so, navigates toward the selected POI. The full BT is shown in Figure 2.3.

This BT simulates a small tour, although some actions are intentionally abstracted; for example, navigation does not check whether the robot is already at the correct location, and battery management does not model charging. The case study is nevertheless useful for assessing how the framework handles more complex systems, which properties can be verified and monitored, and which limitations remain. The main components of this case study are the following:

- A BT that simulates a small tour, where the robot selects a POI to go to and if people are following it, it will navigate to the selected POI.
- Eleven FSMs for the skill layer, including a battery level skill which returns success if the battery level is higher than 30% and failure otherwise, an alarm skill which sends an alarm message to the alarm component when triggered, two skills that select the POI (one for POI 1 and one for POI 2), two skills that check whether a POI has already been explained (again one for each POI), one skill for resetting the POIs to not done, one skill that checks whether people are following or not, one skill that waits for the others to change their state (it always returns RUNNING), one skill that tells the robot to reach the currently selected POI, a navigation skill which simulates the navigation of the robot to the selected POI, and a people-following skill which simulates the detection of people following the robot.
- Six components for the functional layer: The battery component which simulates the battery of the robot, the notify user component which simulates the alarm system of the robot, the scheduler component which is responsible for the selection of the POI, the blackboard component which is used to share data among skills and components, the navigation component which simulates the navigation of the robot to the selected POI, and the people detector filter component which simulates the detection of people following the robot.
- Five properties listed in Table 5.1, which also reports the property pattern described in Section 2.5.1, the natural-language requirement, and the corresponding past-MTL formula. These properties are useful to ensure that the system behaves as expected and to detect deviations at runtime.

To evaluate this case study, SCAN and MOON were tested separately using a simulation, which can be found in the GitHub repository<sup>1</sup>. For SCAN, there is a model of the whole system that can be simulated, while for MOON there is a simulation with all components (Navigator, Alarm, Battery, and Visitors Following) that can be executed stand-alone.

### 5.2.1 Off-line Verification

The system models translated into plain SCXML using AS2FM and the properties written in XML were given as input to the SCAN model checker. The selected SCAN parameters are the default ones: 0.95 for confidence, 0.01 for precision, and the maximum length of execution trace is set to 500000 steps. SCAN takes a maximum of two minutes to check each property, and the resulting precision decreases with the number of runs until it reaches  $\pm 0.01$ .

Table 5.2 shows the success rate for each property considering different temporal bounds (multiples of 100ms). The first property has a success rate equal to one and does not have a time bound. The bound of Property 2 was found manually through trials: in this case, the bound depends on the interaction among ticks and the scheduling of different processes. The exact value is not relevant per se, but it shows that a bound exists. The time bounds for Properties 3 and 5 are correct because they correspond to the rate at which the components publish the corresponding values (i.e., 1Hz), and therefore to a period of 10 time units in both cases. The properties are always true given a bound greater than or equal to 10, and false otherwise. The bound of property 4 is two time units since the system needs to be initialized and, at the beginning, the POI 1 is not yet selected. The bound corresponds to the time required by the Behavior Tree to tick the first leaf nodes and retrieve their values from the components before setting the POI 1 as selected.

---

<sup>1</sup><https://github.com/convince-project/MBVM-SIMPAR2025>

| Property No. | Pattern                | Text                                                                                          | PAST-MTL formulae                                                                                                                                            |
|--------------|------------------------|-----------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1            | ABSENCE<br>BEFORE      | It is always the case that the “alarm” does not occur before the “battery_level < 30%” occurs | $\mathbf{H}(\text{battery\_level} \geq 30\% \implies \mathbf{H}(\neg \text{alarm}))$                                                                         |
| 2            | RESPONSE<br>GLOBALLY   | It is always the case that “alarm” responds to “battery_level < 30%” within $t_1$             | $\mathbf{H}(\text{alarm} \implies \mathbf{P}(\text{battery\_level} < 30\%)) \wedge \neg(\neg \text{alarm} \mathbf{S}_{[t_1:]} \text{battery\_level} < 30\%)$ |
| 3            | RECURRENCE<br>GLOBALLY | It is always the case that “battery_published” occurs at least every $t_2$ time units         | $\mathbf{H}(\mathbf{P}_{[t_2:]} \text{True}) \implies (\mathbf{P}_{[t_2:]} \text{battery\_pub})$                                                             |
| 4            | RESPONSE<br>GLOBALLY   | It is always the case that “POI_1_selected” responds to not “POI_1_completed” within $t_3$    | $\mathbf{H}(\text{POI1\_sel} \implies \mathbf{P}(\neg \text{POI1\_comp})) \wedge \neg(\neg \text{POI1\_sel} \mathbf{S}_{[t_3:]} \neg \text{POI1\_comp})$     |
| 5            | RECURRENCE<br>GLOBALLY | It is always the case that “visitors_published” occurs at least every $t_4$ seconds           | $\mathbf{H}(\mathbf{P}_{[t_4:]} \text{True}) \implies (\mathbf{P}_{[t_4:]} \text{visitors\_pub})$                                                            |

Table 5.1 Properties with the related pattern taken from Ulus [2019a], the text expressed in formal language, and the corresponding PAST-MTL formula.

| Property ID | Temporal Bound | Success Rate |
|-------------|----------------|--------------|
| 1           | -              | 1.00         |
| 2           | $t_1 \geq 62$  | 1.00         |
|             | $t_1 < 62$     | 0.00         |
| 3           | $t_2 \geq 10$  | 1.00         |
|             | $t_2 < 10$     | 0.00         |
| 4           | $t_3 \geq 2$   | 1.00         |
|             | $t_3 < 2$      | 0.00         |
| 5           | $t_4 \geq 10$  | 1.00         |
|             | $t_4 < 10$     | 0.00         |

Table 5.2 Model checking results with respect to temporal bounds (Temporal bounds are in multiples of 100ms)

### 5.2.2 On-line Verification

For on-line verification, all the properties listed in Table 5.1 were tested for various time bounds in order to identify the bounds that lead the properties to fail when running on actual code. Results can be found in Table 5.3.

Similarly to off-line verification, the first property has no time bound and is verified correctly by the monitor. Time bounds for Properties 3 and 5 are correct because they correspond to the rate at which the “Battery” and “Visitors Following” components publish their messages through topics (i.e., 1Hz). For the last two properties, the relevant bound is independent of publication frequency and depends instead on the response of the system. The value for Property 2 was found through trials and corresponds to 2.5 seconds. By looking at the traces, this value corresponds, as expected, to the average time required by the alarm to become active after the battery drops below the threshold. This is equivalent to approximately two ticks of the Behavior Tree. For property number 4 instead, the bound value is related to the time that the behavior tree needs to reach the next action. Also in this case the bound corresponds to the time required by the Behavior Tree to tick the first leaf nodes and retrieve their values from the components before setting the POI 1 as selected.

The different time bounds for property 2 obtained from off-line and on-line verification are due to the fact that the first one is based on the model, which abstracts away from the actual implementation, while the second one is based on the actual code, which includes all the details of the implementation. This highlights the importance of on-line monitoring, as it allows us to detect any deviation from the expected behavior at runtime, which can happen in more sophisticated systems also subject to hardware malfunctions.

| Property ID | Temporal Bound [s] | Success Rate |
|-------------|--------------------|--------------|
| 1           | -                  | 1.00         |
| 2           | $t_1 \geq 2.5$     | 1.00         |
|             | $t_1 < 2.5$        | 0.00         |
| 3           | $t_2 \geq 1$       | 1.00         |
|             | $t_2 < 1$          | 0.00         |
| 4           | $t_3 \geq 0.2$     | 1.00         |
|             | $t_3 < 0.2$        | 0.00         |
| 5           | $t_4 \geq 1$       | 1.00         |
|             | $t_4 < 1$          | 0.00         |

Table 5.3 Monitoring results with respect to temporal bounds

### 5.2.3 Fault Injection and Monitor Overhead

The off-line and on-line validation discussed so far shows that the second case study can be verified and monitored successfully under nominal conditions. A complementary perspective is provided by the experiments reported in Bernagozzi et al. [2025b], where the same case study is used to evaluate whether generated code and runtime monitors can detect concrete implementation faults and whether the added runtime overhead remains acceptable. The objective here is to assess whether the combined use of the framework components improves system reliability by detecting faults that are not visible at the level of the selected properties, but are instead revealed by model monitoring, while keeping monitoring overhead compatible with realistic robotic software stacks. We therefore need to inject faults in the implementation and check whether they are detected by the monitors. In these experiments we considered the skills “BatteryLevel”, “Alarm”, “IsPoiDone1”, and “SetPoi1”, because together they cover different interaction patterns with the functional layer. For property monitoring, rather than introducing a new set of requirements, we focused on the properties already listed in Table 5.1, in particular Property 1 on the absence of alarms before a low-battery condition, Property 2 on the bounded response of the alarm, and Property 4 on the bounded selection of the first POI.

#### Runtime Detection of Injected Faults

To evaluate the ability of the generated monitors to detect anomalous behavior, we injected one fault at a time in the monitored skills. The injected bugs belong to three categories: unresponsiveness, incorrect response, and incorrect parameter names. These categories represent typical problems that can occur in robotic software, and they are also useful to evaluate the ability of the monitors to detect different types of faults. In particular,



unresponsiveness can happen when there is a connection problem or a timeout, incorrect response can happen when the skill has an implementation defect or a logical error, while incorrect parameters may prevent correct connection between skills and the functional layer, so that the response is not received as expected. This setup makes it possible to compare what can be detected by property monitoring alone and what instead becomes visible only when the generated code is checked against the original skill model.

Table 5.5 summarizes the results. Two observations are particularly important. First, model monitoring detects all injected faults, although in the unresponsive cases it needs a timeout before reporting the anomaly. Second, the property monitors only detect the faults that eventually manifest themselves on the selected Case Study 2 requirements. For example, when “SetPoi1” becomes unresponsive or when “IsPoiDone1” uses a wrong parameter name, the behavior appears correct from the standpoint of Properties 1, 2, and 4, while the model monitor still detects the inconsistency. This confirms that model monitoring and property monitoring are complementary: the first checks conformance with the expected internal behavior of a skill, while the second checks whether explicit requirements are violated at system level.

### Monitoring Overhead

The same study also quantified the runtime cost of monitoring. We measured the average tick time of the monitored skills and of the BT root under four configurations: no monitors, only property monitors, only model monitors, and both classes of monitors active. Results are reported in Table 5.4.

The measurements show that property monitoring adds only a small overhead, since it relies on asynchronous observation of ROS 2 communications. Model monitoring is more expensive because, in the current implementation, it relies on a synchronous man-in-the-middle for service interactions. Even so, the resulting delay remains compatible with deliberative components, whose execution times are typically much larger than those of low-level control loops. This result is important because it shows that the additional guarantees provided by model monitoring can be obtained without preventing execution on realistic robotic software stacks.

| Skill Name   | Average Tick Time ( $\mu s$ ) |                   |                |                                      |
|--------------|-------------------------------|-------------------|----------------|--------------------------------------|
|              | No monitors                   | Property Monitors | Model Monitors | Property Monitors and Model Monitors |
| BatteryLevel | 3,877                         | 4,103             | 56,943         | 65,670                               |
| IsPoiDone1   | 6,906                         | 7,030             | 314,621        | 327,816                              |
| SetPoi1      | 6,094                         | 7,663             | 308,686        | 333,764                              |
| Alarm        | 4,699                         | 8,151             | 284,138        | 298,117                              |
| ROOT         | 29,904                        | 37,036            | 810,121        | 864,467                              |

Table 5.4 Average tick time measured with different monitoring configurations.

| Test ID | Type of fault       | Skill        | Model Monitor                                     | Property Monitor                                                       |
|---------|---------------------|--------------|---------------------------------------------------|------------------------------------------------------------------------|
| T1      | Unresponsiveness    | BatteryLevel | Detects after timeout                             | Property 1: no detection; Property 2: detects when battery_level < 30% |
| T2      | Unresponsiveness    | SetPoi1      | Detects after timeout                             | Property 4: no detection                                               |
| T3      | Incorrect response  | BatteryLevel | Detects at first occurrence                       | Property 1: detects at first occurrence                                |
| T4      | Incorrect response  | Alarm        | Detects at first occurrence                       | Properties 1 and 2: no detection                                       |
| T5      | Incorrect parameter | BatteryLevel | Detects when the faulty value affects the outputs | Property 2: detects when battery_level < 30%                           |
| T6      | Incorrect parameter | IsPoiDone1   | Detects at first occurrence                       | Property 4: no detection                                               |

Table 5.5 Summary of the fault-injection experiments on the second case study.

## 5.3 Results

The results of this chapter show that the proposed framework can support the full development chain of deliberation components, from design-time modeling to execution-time monitoring. The first case study demonstrates that the framework already works end-to-end on a minimal example: a BT can be modeled together with the associated skills and components, translated for off-line verification, compiled into executable code, and monitored at runtime using the same design artifacts. Although simple, this example is important because it isolates the basic mechanisms of the framework and shows that they can be exercised consistently without introducing ambiguities in the interpretation of the models.

The second case study confirms that the same approach remains effective on a richer deliberative structure. In this scenario, the framework supports a BT with multiple sub-trees, a larger set of skills and components, and several temporal properties concerning battery management, POI selection, and periodic topic publication. The off-line results obtained with SCAN and the on-line results obtained with MOON are consistent with each other: the same properties that hold on the model also hold on the generated implementation when the temporal bounds are chosen according to the timing behavior of the system. This is a significant result because it shows that the transition from models to executable code does not break the validity of the analyzed requirements, at least for the considered class of deliberation components.

Additionally, results are also validated through a comparison between model monitor and properties monitor. Instead of considering only nominal executions, these experiments inject faults into the implementation of selected skills in the second case study. The resulting evidence shows that runtime monitoring is not limited to confirming expected behavior, but can also expose concrete deviations in generated code. In particular, the experiments highlight that property monitors and model monitors provide different but complementary guarantees: property monitors detect faults only when they propagate to explicitly monitored requirements, whereas model monitors can reveal inconsistencies with the expected skill behavior even when no property becomes false. The overhead analysis further shows that this additional monitoring capability remains compatible with deliberative execution times, especially for property monitoring and, with a higher but still acceptable cost, also for model monitoring.

With respect to the four research questions stated at the beginning of the chapter, the results show that:

- RQ1: The framework can be used to generate code for deliberation components, and the generated code can be executed on a real robot.
- RQ2: The same models used for code generation can be translated into a form suitable for off-line verification, and the results of off-line verification are consistent with the observed behavior of the generated code when the temporal bounds are chosen according to the timing characteristics of the system.
- RQ3: The generated code can be monitored at runtime using the same design artifacts, and the monitors can detect both expected behavior and deviations from the model, including faults that do not necessarily violate explicitly monitored properties.
- RQ4: The overhead introduced by runtime monitoring is acceptable for deliberation-layer components, especially for property monitoring, and remains compatible with the execution times of the considered case studies.

Taken together, these results indicate that the main value of the framework is not only in the individual tools, but in their integration. Modeling, code generation, off-line verification, and runtime monitoring are all grounded in a common representation, which reduces the gap between what is designed, what is verified, and what is finally executed. The validation was also formative: implementing the scenarios and defining their properties exposed application-level requirements and integration constraints that would not have emerged from isolated tool testing. The resulting robotics-domain feedback informed design decisions for the monitoring workflow and the interfaces between the shared tools. This interpretation is deliberately narrower than claiming complete robotic correctness, but it is stronger than showing that each tool runs in isolation: the evidence supports the claim that the same design artifact can survive a complete development cycle and remain useful at each stage.

# Chapter 6

## Discussion

The main outcome of this part is methodological. It is not simply a presentation of a code generator, a model checker, and a runtime monitor as isolated tools; rather, it shows that they can operate coherently on a shared representation of deliberative robotic software. This coherence is what makes the approach meaningful: generated code remains structurally close enough to the original models to support monitoring, while the same models can also be translated into a form suitable for off-line statistical verification. In this sense, this part contributes a unified workflow rather than a collection of disconnected techniques. Compared with the model-driven robotics approaches reviewed earlier in this part, the main distinction is not only that executable ROS 2 code is generated, but that the generated implementation remains connected to verification and monitoring artifacts after generation.

The validation results also clarify the role of the two forms of runtime monitoring. Property monitoring is particularly useful when the developer knows in advance which requirements should be stated explicitly and wishes to check them on-line. Model monitoring addresses a different need: instead of checking a specific requirement, it verifies that an implemented skill still behaves according to the model from which it was generated. The fault-injection experiments of the second case study make this distinction concrete, because they show that some faults remain invisible at the level of the selected properties while still being detected by model monitoring. For this reason, the two monitoring strategies should be seen as complementary rather than interchangeable.

Another important discussion point concerns the reality gap between verified models and running software. Off-line verification provides confidence that the model satisfies a set of properties under the assumed semantics and timing abstraction. On-line monitoring cannot replace that role, but it provides a safeguard once the model has been compiled and deployed. The combination of the two therefore addresses a fundamental limitation of purely

model-based verification: even if the model is correct, the implementation can still deviate due to translation issues, integration faults, or unexpected timing effects. The chapter results show that using SCAN and MOON together is a practical way to reduce this gap.

## 6.1 Interpretation and Generalizability

The results should be interpreted as evidence about semantic continuity across the framework, rather than as a proof that all possible robotic failures are covered. The strongest claim supported by the experiments is that, for deliberation-layer skills expressible in the supported ASCXML subset, the same behavioral model can drive code generation, off-line statistical verification, and runtime monitoring without forcing the developer to maintain three unrelated artifacts. This is the main difference with workflows in which a design model, an implementation, and a monitor configuration evolve independently. The experiments also show that the benefit is practical: the fault-injection campaign reveals errors that would otherwise require manual log inspection or ad-hoc debugging.

The two monitoring modes offer different levels of coverage. Property monitoring is effective when the desired guarantees can be stated explicitly at the communication level, but it still requires the developer to formulate the relevant requirements and to define how observed messages are abstracted into the predicates used by the property. Model monitoring provides a complementary check once the component model is available, because the developer does not need to formalize additional properties manually. This is not only a convenience benefit: in the reported experiments, model monitors were able to detect certain injected faults immediately and, in some cases, to reveal anomalies that were not captured by the available property monitors. At the same time, the current implementation may introduce non-negligible overhead, especially for monitored interactions that require richer runtime coordination than passive topic observation.

The generality of the results is strongest for robotic systems that share the architectural assumptions used in this thesis: a BT-based deliberation layer, FSM-based skills, and ROS 2 interfaces that expose the relevant interactions. The conclusions are weaker for perception-heavy or control-level components whose behavior cannot be summarized well by finite-state models, or for systems whose critical behavior is hidden inside node-local computation. This limitation is not accidental; it is the tradeoff that makes the framework analyzable. The current contribution is therefore best understood as a dependable-software workflow for deliberative robotic behavior, rather than as a complete verification solution for every layer of an autonomous robot.

## 6.2 Open Issues and Future Extensions

Several natural extensions emerge from the work presented in this chapter. A first one concerns monitoring support for a broader subset of ROS 2 communication mechanisms. At present, the monitoring pipeline is more mature for services and topics than for actions, and model monitoring still relies on an invasive man-in-the-middle approach for part of the communication. Extending support to richer ROS 2 interaction patterns and exploiting introspection mechanisms more systematically would reduce runtime overhead and make deployment easier in larger systems.

Another extension concerns the boundary between the deliberative layer and the functional layer. In the current framework, monitoring and generation focus mainly on deliberation components and on their interfaces with the rest of the architecture. Extending model-based support deeper into the functional layer would further reduce the abstraction gap between the verified models and the deployed system, and would make it possible to reason about a wider portion of the robotic stack within the same methodological framework.

There is also room to improve the treatment of timing. In the chapter, temporal bounds are identified empirically and justified by the execution characteristics of the case studies. This is sufficient for the present validation, but a more systematic derivation of such bounds from the architecture, the middleware, or the execution traces would make the verification workflow more robust and less dependent on manual tuning.

Finally, another research direction can be the preservation of the semantics between the model used by the model checker and the executed model, with tools such as CompCert<sup>1</sup> or a more formal standardization of BTs and FSMs execution.

More broadly, the framework suggests a research direction in which modeling, code generation, verification, and runtime monitoring are treated as phases of a single development life-cycle rather than as separate activities.

---

<sup>1</sup><https://compcert.org/>



## **Part II**

# **Flow-Aware Planning**

Robots are increasingly being deployed in human-populated environments, such as museums, shopping malls, and airports, where they must navigate around people to perform almost any kind of task, such as delivery, cleaning, or guiding. In these environments, the presence of people can cause congestion, which can significantly impact the robot's ability to perform its tasks efficiently, by either delaying the robot or even preventing it from reaching its goal.

This part presents a flow-aware planning approach that enables robots to anticipate and avoid congestion by using predictions of human motion. In particular, the framework allows the robot to plan a tour of a set of points of interest (POIs) in the environment, while avoiding congested areas that may arise due to the presence of visitors.

From the perspective of the overall thesis, this part complements the previous work on model-based development and runtime assurance by addressing adaptive decision making under crowd dynamics. Where the previous part focused on making deliberative robotic software more dependable, this part focuses on improving the quality of the robot's task-level decisions when operating in public environments.

The main contributions of this part are as follows:

- A task-level abstraction that converts sampled human-motion predictions into probability distributions over future congestion on the edges of a topological map.
- A stochastic shortest-path MDP formulation for tour planning under time-varying human congestion, where actions are navigation or waiting decisions and transition probabilities are induced by predicted congestion bands.
- An on-line receding-horizon planning algorithm based on LRTDP, which repeatedly solves the current MDP approximation, executes the first action, and replans after new observations are available.
- An experimental evaluation showing that the proposed abstraction improves tour execution time compared with a replanning Hamiltonian-path baseline that reacts to current congestion but does not model its stochastic future evolution.

The scientific novelty of this part is therefore not the isolated use of CLiFF maps, MDPs, or LRTDP, which are established tools. Rather, it lies in the way these elements are combined into a planning model that connects learned pedestrian-flow predictions with long-horizon task sequencing. The planner explicitly separates three levels of representation: continuous human trajectories, probabilistic congestion over topological edges, and abstract tour actions. This separation is what makes the approach compatible with the concrete BT-based execution

architecture developed in the previous part. In other words, the contribution is a bridge between prediction and deliberation: human-flow forecasts are not used only to bias a local path planner, but are transformed into stochastic task-level consequences that can change which POI the robot should visit next, whether it should wait, and how it should trade a longer geometric route against a less congested future one. This is the point at which the work differs from both classical tour construction with fixed costs and most social-navigation approaches, which usually optimize immediate motion around people rather than the sequencing of several goals under predicted crowd evolution.

**Key novelty highlights.** For quick orientation, the key novelties of this part are:

- *Prediction-to-planning lift*: sampled human trajectories are converted into probability distributions over future congestion on topological edges, rather than being used only as local obstacle forecasts.
- *Task-level stochastic action model*: navigation and waiting choices are represented as actions whose outcomes depend on predicted congestion bands, allowing the robot to compare route order, detours, and waiting under uncertainty.
- *On-line receding-horizon use*: the planner repeatedly rebuilds the flow-aware MDP from new observations, executes only the first abstract action, and then replans, making the method compatible with dynamic public environments.

**Example.** To make the planning problem concrete, consider a museum guide robot that must explain three POIs,  $P_1$ ,  $P_2$ , and  $P_3$ , as Figure 6.1.

The robot is currently at  $P_1$  and must decide how to reach  $P_2$ . From its current position,  $P_2$  can be reached either through a short corridor or through a longer side passage.

At the current instant the short corridor may still appear free, but the observed visitors are moving toward it; CLiFF-LHMP samples therefore predict that the corridor is likely to become congested when the robot would arrive. A planner that only reacts to current congestion may choose the short route, while the flow-aware planner can decide to wait, use the side passage, or visit  $P_3$  first if that makes the expected tour duration lower. This example will be used in the framework chapter to illustrate how continuous human-flow predictions become edge-level congestion probabilities, how those probabilities affect MDP transitions and costs, and how the resulting abstract decision is later executed by the BT-based architecture.

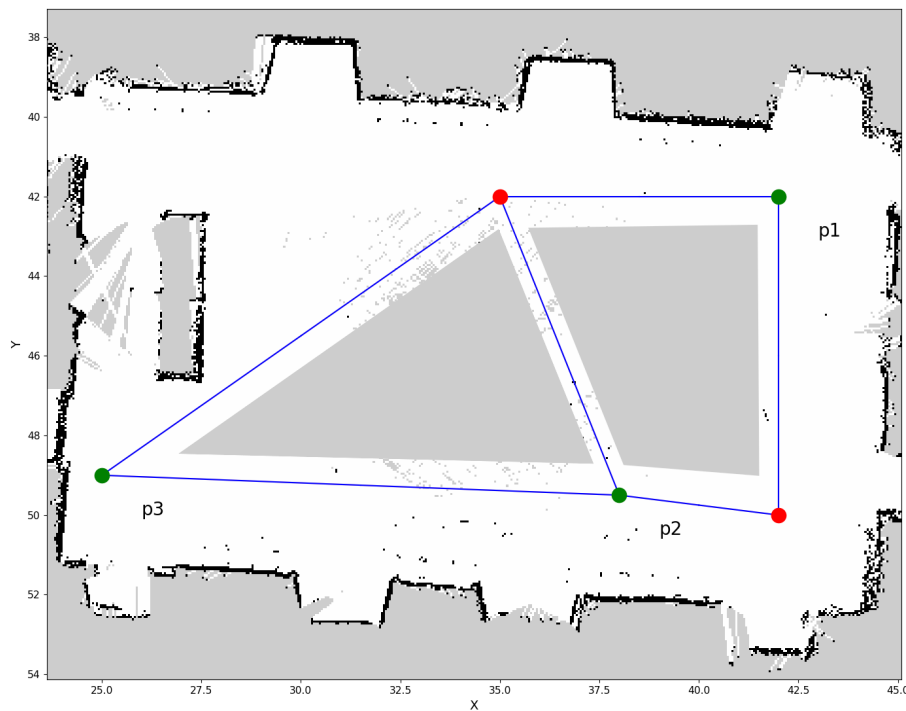


Figure 6.1 Example map layout with three POIs and two possible routes to  $P_2$  from  $P_1$ . To explain all the POIs starting from  $P_1$  the robot has 3 options: explain first  $P_2$  through the shortest route and then  $P_3$ , explain first  $P_2$  through the longest route and then  $P_3$ , explain first  $P_3$  and then  $P_2$ .

# Chapter 7

## Background

### 7.1 Topological Maps

One of the problem that rises in robotics and computer science is how knowledge is represented (Markman [2013]), and how it can be used to solve problems. Modeling the environment is a key aspect of this problem, and different representations have been proposed to capture different aspects of the environment. In this thesis the environment is represented using a topological map to capture the various points of interest (POIs) in the environment and the paths between them.

**Definition 1** A topological map is a tuple  $\mathcal{T} = \langle \mathcal{V}, \mathcal{E}, \rho \rangle$ , where  $\mathcal{V}$  is a finite set of nodes representing POIs in the environment;  $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$  is a set of edges the robot can navigate along; and  $\rho : \mathcal{E} \times \mathbb{N} \rightarrow \text{Dist}(\mathbb{R}_{\geq 0})$  is a function that takes an edge and the number of humans on that edge, and returns a distribution over how long it takes the robot to traverse that edge. These distributions can be learned from data.

The topological map captures the spatial structure of the environment, and is used to plan tours for the robot. It abstracts away from the low-level details of the environment, such as the geometry and the dynamics of the robot, and focuses on the high-level structure of the environment. An example of a topological map can be seen in Figure 7.1.

### 7.2 Hamiltonian Path Problem

Since the robot has to visit all the POIs in the environment, the tour planning problem can be seen as a combinatorial problem, where the robot has to find a sequence of vertices to visit that minimizes the expected time to complete the tour. The problem of finding a

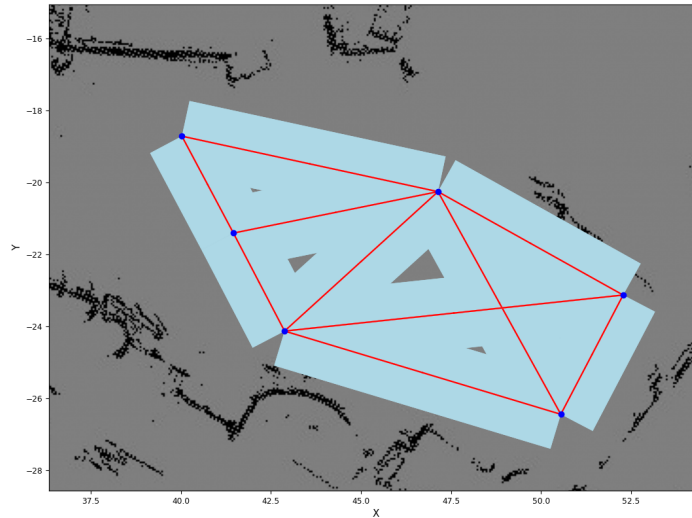


Figure 7.1 Example of a topological map, where vertices represent POIs and edges represent paths between them. It has 6 vertices and 10 edges. In light blue is represented the area that an edge takes up in the environment.

sequence of vertices to visit that minimizes the expected time to complete the tour is known as the Hamiltonian path problem, which is a well-known combinatorial problem in computer science. It is among the most fundamental NP-complete problems, and is often used as a benchmark for evaluating the computational complexity of algorithms. In its general form, the Hamiltonian path problem asks whether there exists a path in a graph that visits each vertex exactly once, without returning to the starting point (Karp [2009]). In the context of tour planning, the graph is represented by the topological map, as described before, where vertices represent POIs and edges represent paths between them.

The robot has to find a sequence of vertices to visit that minimizes the expected time to complete the tour, which can be seen as finding a Hamiltonian path in the graph. An Hamiltonian path over a graph can be defined as follows:

**Definition 2** *Given a graph  $G = (V, E)$ , a Hamiltonian path is a path that visits each vertex in  $V$  exactly once.*

The Hamiltonian path problem can be also reduced to a standard Traveling Salesman Problem (TSP) (Papadimitriou [1977]), which asks for the shortest possible route that visits each vertex exactly once and returns to the starting point. To reduce the Hamiltonian path problem to TSP, the graph is transformed into a complete graph by introducing a dummy vertex  $v_d$  whose only outbound vertex is connected only to the starting vertex  $v_s$  of the Hamiltonian path, with edges of zero cost, and has inbound vertices from any other vertex

with zero cost. In this way, the optimal solution to the TSP will correspond to a Hamiltonian path in the original graph. Many solvers are available for the TSP; in this part of the thesis, the LKH-3 solver<sup>1</sup> (Helsgaun [2017]) is used.

### 7.3 Circular Linear Flow Field (CLiFF) Maps

A third key aspect of knowledge representation is how to capture the dynamics of the environment, and in particular the dynamics of human motion. To solve this problem, CLiFF maps are used to capture human motion dynamics in terms of their velocity at discrete locations in an environment (Kucner et al. [2017]). In this chapter, they are used to predict human congestion during tour planning.

**Definition 3** A CLiFF map (Kucner et al. [2017]) is a set  $\Xi = \{\Xi_1, \dots, \Xi_n\}$ , where  $\Xi_i = \langle l_i, \xi_i, p_i, q_i \rangle$ .

Each  $\Xi_i$  corresponds to a discrete location  $l_i \in \mathbb{R}$  in the environment. Locations  $l_i$  are often arranged in a grid.  $\xi_i$  is an SWGMM that describes the distribution over human velocity at  $l_i$ . Here, velocity is represented in terms of its orientation (the circumference of the SWGMM) and speed (the height of the SWGMM). CLiFF maps are data-driven models. Value  $p_i$  describes the probability of motion occurring at  $l_i$ , i.e. how often was there motion when we observed  $l_i$ ? Value  $q_i$  is the proportion of the data collection time we spent observing  $l_i$ , and acts as a confidence measure for  $\Xi_i$ .

To learn a CLiFF map from continuous data, observations are assigned to any  $l_i$  they fall within a fixed radius of.

Given a CLiFF map, Zhu et al. [2023] proposes an approach for predicting human trajectories given a history of observations of their position and velocity. This history is used to estimate the human's current state, i.e. their position and velocity. The predicted trajectory is built iteratively along each timestep in the prediction horizon. For each iteration, they first apply the human's current velocity to obtain their next position. They then sample a direction and speed from the SWGMM for the nearest location  $l_i$  in the CLiFF map. The human's next direction and speed are then computed by biasing their current direction and speed with those sampled.

**Definition 4** Given a CLiFF map  $\Xi$  and human observation history  $\mathcal{H}$ , we can predict a trajectory  $\Psi : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^2$  for the human, where  $\Psi(t)$  is the human's predicted location at time  $t$ .

<sup>1</sup><http://webhotel4.ruc.dk/~keld/research/LKH-3/>

Trajectories  $\Psi$  predicted by CLiFF map  $\Xi$  are sampled from a wider distribution of trajectories for the human. In Section 9.3, these distributions are estimated by sampling multiple trajectories for each human. This is used to build a probabilistic model of human congestion. Further, CLiFF maps discretize the environment in a more granular way than topological maps. Section 9.3 then explains how predicted trajectories are mapped onto topological edges for congestion modeling.

## 7.4 Poisson Binomial Distribution

To model the probability of human congestion on topological edges, the Poisson binomial distribution is used to model the number of humans on an edge at a given time. The Poisson binomial distribution is a generalization of the binomial distribution that models the number of successes in a sequence of independent Bernoulli trials with different success probabilities. It can be defined as follows:

**Definition 5** *Given  $n$  independent Bernoulli trials with success probabilities  $p_1, p_2, \dots, p_n$ , the Poisson binomial distribution models the probability of observing  $k$  successes as:*

$$P(X = k) = \sum_{A \subseteq \{1, 2, \dots, n\}, |A|=k} \prod_{i \in A} p_i \prod_{j \in A^c} (1 - p_j)$$

where  $A^c$  is the complement of  $A$  in  $\{1, 2, \dots, n\}$ .

## 7.5 Markov Decision Processes

In computer science, to solve a non-deterministic problem, is often useful to model it as a Markov Decision Process. A Markov Decision Processes (MDPs) is a mathematical framework for modeling decision-making in situations where outcomes are partly random and partly under the control of a decision maker (Puterman [1994]). It model systems with non-deterministic action choices and stochastic action outcomes and is widely used in various fields, including robotics, artificial intelligence, and operations research. Here, MDPs are used as base model for tour planning in crowded environments, where the robot has to make decisions about which path to take and how to navigate through the environment while considering the presence of humans and their movements. An MDP can be defined as follows:

**Definition 6** *An MDP is a tuple  $\mathcal{M} = \langle S, \bar{s}, A, T, R \rangle$ , where:*



- $S$  is a finite set of states;
- $\bar{s}$  is the initial state;
- $A$  is a finite set of actions;
- $T : S \times A \times S \rightarrow [0, 1]$  is a probabilistic transition function, where  $T(s, a, s')$  denotes the probability of reaching state  $s'$  after executing action  $a$  in state  $s$ ;
- $R : S \times A \rightarrow \mathbb{R}$  is a reward function, where  $R(s, a)$  is the reward of executing action  $a$  in state  $s$ .

The objective of an MDP is to find a policy that maximizes the expected cumulative reward over time.

In this thesis, however, the objective is to minimize the expected cumulative cost to reach a goal state, which can be modeled by negating the reward function. To do this, the tour planning problem is modeled as a Stochastic Shortest Path (SSP) MDP, which is a special case of MDPs where the objective is to reach a goal state with minimum expected cumulative cost. It differs from a standard MDP in that it has a set of goal states and the reward function is defined as a cost function that is non-negative and zero for goal states.

An SSP MDP can be defined as follows:

**Definition 7** An SSP MDP (Puterman [1994]) is a tuple  $\mathcal{M} = \langle S, \bar{s}, A, T, C, G \rangle$ , where:

- $S, \bar{s}, A$  and  $T$  are defined as Def. 6;
- $C : S \times A \rightarrow \mathbb{R}_{\geq 0}$  is a cost function, where  $C(s, a)$  is the cost of executing action  $a$  in state  $s$ ;
- $G \subseteq S$  is a set of goal states.

The optimal solution to an SSP MDP can be represented as a deterministic memoryless policy, which maps each state to an action without considering the history of states or actions taken.

**Definition 8** A deterministic memoryless policy is a mapping  $\pi : S \rightarrow A$ , where  $\pi(s)$  denotes the action to execute in state  $s$ .

# Chapter 8

## Related Work

Flow-aware planning for robots in human-populated environments sits at the intersection of several research areas. The most relevant for this part of the thesis are:

- i. maps of dynamics for modeling and predicting human motion,
- ii. social navigation for moving safely and efficiently around people,
- iii. task-level planning methods that reason about congestion and time-dependent travel costs.

This chapter reviews these strands of literature and positions the contribution of this part with respect to them. The positioning is organized around two distinctions that are important for the thesis. The first distinction is between predicting human motion and using that prediction for decision making. The second is between motion-level planning, where the output is a local path or control policy, and task-level planning, where the output is a sequence of goals or route choices over a longer horizon. The contribution of this part lies in the intersection of these two dimensions: using short-horizon stochastic human-flow predictions as input to on-line task-level tour planning.

### 8.1 Map Of Dynamics

A map of dynamics captures environment features that vary over time and affect robot behavior, such as human motion for ground robots, wind for aerial robots, or currents for underwater robots.

One of the most known approaches to model dynamics is FreME<sub>n</sub> (Krajník et al. [2017]), which models periodic temporal dynamics using Fourier transforms. This is well suited

to environments where recurrent patterns dominate, such as offices in which occupancy follows daily rhythms. However, FreMEN models Bernoulli variables such as occupancy and is therefore not well suited to scenarios in which the robot must reason about how many people may be present in an area at a given time. This limitation is mitigated in Vintr et al. [2019], which extends FreMEN through a warped hypertime representation that can model continuous variables. Even so, both FreMEN and the warped hypertime model are primarily aimed at longer-term temporal regularities, such as patterns over days or weeks, rather than predictions on the scale of seconds, which makes them not ideal for the short-horizon congestion forecasting required in this part of the thesis. Moreover, they do not explicitly model current motion and only weakly exploit the present state of the environment. Modelling motion more explicitly is partially addressed in Molina et al. [2018], where FreMEN is extended to reason about the direction of human motion, although the resulting model still does not explicitly condition on the current occupancy in the way required here.

An alternative line of work models human motion directly through local flow patterns. CLiFF (Circular Linear Flow Field) maps represent the speed and direction of human motion at discrete locations in an environment using semi-wrapped Gaussian mixture models (SWGMMs) (Kucner et al. [2017]). These maps can then be used to predict human trajectories from a short observation history. In particular, Zhu et al. [2023] predicts trajectories up to 50 seconds into the future by combining recent human observations with the local motion patterns encoded in the CLiFF map. This makes CLiFF especially relevant to the present work, where short-horizon predictions are used to estimate future congestion. In this part, CLiFF maps are not used only to forecast individual motion, but also as the basis for estimating distributions over crowd levels on topological edges.

Related flow-based representations include directional grid maps (Senanayake et al. [2020]), which also capture the speed and direction of human motion over the environment. Similarly, Saraydaryan et al. [2023] predicts human trajectories using von Mises distributions rather than SWGMMs, providing an alternative probabilistic representation of directional uncertainty. More recently, deep-learning approaches have also been proposed for modeling crowd motion. For example, Kazemi Eskeri et al. [2025] uses an encoder-forecaster architecture to predict crowd dynamics for social navigation. For a broader overview of maps of dynamics, the survey in Kucner et al. [2023] provides a useful taxonomy and discussion. That survey is useful here because it separates several roles that maps of dynamics can play: they may describe where people are likely to be, how they are likely to move, or how environmental dynamics should affect robot decision making. The distinction matters for this thesis. Long-term occupancy models are valuable for deployment analysis and

route design, but the robot tour considered here requires short-horizon predictions that are conditioned on the current distribution and motion of visitors. Conversely, local flow models can support immediate motion planning, but they do not by themselves define how a robot should sequence several POIs over a longer task horizon.

Several works have already shown that maps of dynamics can improve motion-level planning. For instance, Swaminathan et al. [2018] uses CLiFF maps for flow-aware trajectory planning under motion-pattern uncertainty, while Swaminathan et al. [2022] studies the utility of maps of dynamics for human-aware motion planning. These works are close to the present thesis in their use of learned motion patterns, but they operate primarily at the trajectory or path-planning level. The contribution developed here is complementary: the CLiFF-based prediction layer is lifted to a topological task model, where the robot reasons about which POI to visit next and when waiting may be useful.

## 8.2 Social Navigation

Robots deployed in public environments must navigate in the presence of people whose goals, motion patterns, and willingness to interact with the robot may all be uncertain. This has led to a large body of work on social navigation, where the objective is not only collision avoidance but also motion that is safe, legible, and socially acceptable. In most of this literature, the main focus is on motion-level decision making rather than on long-horizon task sequencing.

Crowd navigation is commonly addressed through either planning-based approaches (Banisetty et al. [2021]; Feyzabadi and Carpin [2014]; Singamaneni et al. [2021]) or reinforcement-learning-based approaches (Chen et al. [2017]; Kästner et al. [2022]; Pérez-Higueras et al. [2018]; Wang et al. [2022]). Both families of methods have shown strong results, although planning-based approaches remain attractive because of their lower data requirements and their clearer control over safety constraints. More recent work has also explored hybrid approaches. For example, Matsumoto et al. [2024] combines learning-based and rule-based elements for crowd-aware navigation, while Debnath et al. [2025] combines proactive global planning with reinforcement-learning-based local avoidance.

From the perspective of this thesis, the key limitation of most social-navigation methods is not that they ignore human behavior, but that they typically use human information to shape local motion generation or short-horizon path selection. By contrast, the problem addressed in this part is task-level: the robot must decide which point of interest (POI) to visit next, and possibly whether to wait, while reasoning about how future human flows

will affect travel times across the topological map. The survey literature helps clarify this difference. Early work on human-aware robot navigation, such as the survey by Kruse et al. [2013], organizes the field around navigation behaviors that respect human comfort, safety, and social conventions. This perspective is still central: it treats people not merely as moving obstacles, but as social agents whose space, motion, and expectations constrain robot motion. Later reviews, including Chik et al. [2016] and Charalampous et al. [2017], broaden this view to service robots in dynamic human environments and discuss frameworks that combine perception, path planning, social constraints, and local collision avoidance. These surveys show that social navigation is a system-level problem, but the planning component is still usually evaluated in terms of local path quality or immediate interaction with nearby people.

More recent surveys refine this picture. Möller et al. [2021] emphasizes the pipeline from human perception and tracking to human-aware navigation decisions, making clear that the quality of navigation depends strongly on the available human-state representation. Singamaneni et al. [2024] provides a taxonomy of socially aware navigation methods and highlights recurring challenges such as benchmarking, generalization across environments, and balancing efficiency with social acceptability. Mavrogiannis et al. [2023] similarly identifies core open challenges in social robot navigation, including reasoning about multi-agent interaction, human-robot legibility, and evaluation in realistic crowds. These concerns are essential for the lower navigation layers used by a museum guide, but they do not directly solve the task-level sequencing problem considered in this part. Taken together, these surveys show that the dominant evaluation criteria in human-aware navigation are safety, comfort, legibility, proxemic compliance, and local navigation efficiency. These criteria are essential for the navigation layer, but they do not directly answer the task-level question considered here: how the robot should order or postpone visits to multiple POIs when predicted crowd flow changes the expected cost of future route segments.

### 8.3 Task-Level Planning under Congestion

The planning problem considered in this part of the thesis is also related to routing problems such as Hamiltonian path and travelling salesperson formulations (Papadimitriou [1977]; Sevaux et al. [2008]). These formulations provide useful algorithmic background for tour construction, and state-of-the-art solvers are strong baselines when travel costs are static. However, classical routing formulations usually assume fixed edge costs and do not model how travel times evolve stochastically as a result of human congestion. They are therefore

informative as a computational reference point, but insufficient on their own for the present setting.

Few works consider high-level planning in crowded environments. Dugas et al. [2020] presents a Monte-Carlo tree search planner that selects high-level navigation behaviors according to the crowdedness of the environment and the estimated willingness of people to let the robot pass. These behaviors include standard motion planning, verbal communication, and nudging forward to escape crowded areas. This is closely related in spirit, but the objective is still to navigate through the crowd by selecting interaction strategies, whereas the present work reasons about how to route around congestion at the level of a topological tour.

Debnath et al. [2025] proposes a combined global and local planner for human-populated environments. As in the present work, the cost of navigation is increased in crowded areas. However, this is modeled through hand-designed Gaussian costs centered on currently observed human locations. In contrast, the approach developed here uses a learned map of human motion to predict how congestion evolves over time and then lifts those predictions to a task-level planning model.

Planning under congestion has also been studied in multi-robot settings. Street et al. [2022] models congestion caused by other robots using continuous-time Markov chains and incorporates this information into MDP-based planning. This is particularly relevant because it demonstrates how stochastic congestion estimates can be integrated into a high-level planning formalism. The approach in this part adopts a similar MDP-based perspective, but replaces robot-induced congestion with human congestion predicted from observed visitor motion. This replacement is not only a change of application domain. Robot-induced congestion is generated by agents whose policies may be known, coordinated, or controlled by the same system, whereas visitor congestion is exogenous, only partially observed, and inferred from noisy motion patterns. As a consequence, the planner developed here must connect a learned crowd-prediction layer to the task model, rather than deriving congestion dynamics from a controlled multi-robot process.

The baseline used in the evaluation, a Hamiltonian-path solver with replanning, should be understood in this context. It represents a strong deterministic tour-construction approach once current edge costs are available: it can react to observed congestion and replan after each move. What it does not represent is the future stochastic evolution of those costs. This makes it a suitable comparison for the central claim of the thesis: if two task-level planners replan on the same observations, but only one of them reasons over predicted congestion distributions, then the performance gap can be attributed to the flow-aware stochastic abstraction rather than to replanning itself.

## 8.4 Research Gap

The literature reviewed above provides several of the ingredients required for planning in crowded public environments, but not their full combination. Maps-of-dynamics methods can model and predict human motion, yet they often stop at the prediction layer and do not explain how these forecasts should be exploited for long-horizon task decisions. Social-navigation methods, by contrast, typically use human information at the level of local motion generation or short-horizon path adaptation rather than for deciding the order in which a robot should visit multiple goals. Finally, existing task-level approaches that reason about congestion either focus on different sources of congestion, such as other robots, or rely on simplified occupancy models that do not capture the stochastic temporal evolution of human crowds. The resulting gap is not simply that previous work lacks one particular algorithmic ingredient. Rather, the missing element is an abstraction pipeline that preserves enough uncertainty from human-flow prediction to influence task-level decisions, while remaining compact enough for repeated on-line planning.

Within the overall thesis, this part addresses that gap from the planning side. Where Part I established the model-based and runtime-assurance foundation for deliberative robotic software, Part II contributes a flow-aware planning framework that connects crowd prediction with on-line task-level decision making. More specifically, it:

- i. uses a learned map of human motion to predict how congestion evolves over time,
- ii. abstracts these predictions into a stochastic planning model suitable for on-line tour planning,
- iii. re-plans during execution so that the robot can adapt its route to the current and anticipated flow of visitors.

Compared with deterministic tour construction, the contribution is the explicit treatment of future congestion as stochastic action outcomes. Compared with social-navigation approaches, the contribution is not a new local collision-avoidance policy, but a task-level mechanism for deciding which POI or route should be attempted next. Compared with map-of-dynamics work, the contribution is to use human-flow predictions as decision-making input rather than only as an environmental forecast. The resulting contribution is therefore a task-level planning framework that bridges crowd prediction, probabilistic reasoning, and long-horizon decision making in public environments, while remaining consistent with the broader thesis objective of dependable adaptive autonomy.

# Chapter 9

## Framework

This chapter presents the probabilistic planning framework used for tour planning in crowded environments. A formal definition of the tour-planning problem is introduced first, followed by an overview of the proposed solution and an evaluation of the framework on synthetic data. The tour-planning problem is defined similarly to a Hamiltonian path problem (Section 7.2), but the POIs may be visited more than once.

**Problem 1** *Given a topological map  $\mathcal{T}$  describing the POIs in the environment as in Def. 1, synthesize a policy  $\pi$  that minimizes the expected duration required to visit each node/POI  $v \in \mathcal{V}$ . POIs may be revisited during the tour.*

The permission to revisit POIs is a deliberate modeling choice. In a physical environment, reaching a new POI may require passing through a previously visited region, and avoiding a congested edge may require a temporary detour through nodes that have already been completed. A pure Hamiltonian-path formulation would forbid this behavior and would therefore conflate task completion with graph traversal constraints. The formulation used here instead treats the visited set as the task-progress variable and keeps the topological position as the physical state of the robot. This distinction is important because it allows the robot to reason about routes that are longer geometrically but better under predicted congestion. In the running example introduced at the start of this part, the robot may have already explained  $P_1$  but still pass near it again to reach  $P_3$  through a less congested passage. The goal is not to avoid revisiting physical regions; it is to complete the set of explanations efficiently under changing crowd flow.



## 9.1 Modeling Rationale and Action Abstraction

The proposed formulation uses an MDP because the robot faces two coupled decisions that are not captured by a deterministic routing model. First, it must choose which topological action to execute next, for example which edge to traverse or whether to wait. Second, the duration and usefulness of that action depend on uncertain future congestion, which is only partially observable through current human observations and the learned flow model. An MDP is therefore a suitable abstraction: it keeps the controllable choices explicit as actions, while representing the predicted effect of human motion as stochastic outcomes. Since the objective is to reach a goal condition with minimum expected duration, the formulation is a stochastic shortest-path MDP rather than an infinite-horizon reward-maximization problem. Alternative formulations are possible, but they would emphasize different aspects of the problem. A deterministic time-dependent shortest-path or travelling-salesperson model could use predicted mean costs, but it would collapse the distribution of future congestion into a single value before planning. A POMDP would represent the unobserved human state more explicitly, but it would be substantially harder to solve on-line at the scale required for repeated tour replanning. The MDP abstraction adopted here is therefore a middle ground: uncertainty enters through stochastic action outcomes induced by predicted congestion bands, while the state remains compact enough to support receding-horizon execution.

The action abstraction is deliberately coarser than the concrete execution actions used by the robot. In the MDP, an action such as traversing edge  $(v_i, v_j)$  means that the robot commits to moving from one topological region or POI to another. In the implemented system, the same decision is realized by the BT and skill architecture described in Part I: the scheduler selects the next POI, the GoToPoi sub-tree invokes navigation-related plugins, and the corresponding skills and components call the concrete ROS 2 navigation interfaces. Thus, the MDP action is an abstract task-level decision, while the BT/plugin/skill chain is the concrete execution mechanism. Keeping this separation makes the planning model small enough for on-line use while preserving a clear path from planned actions to executable robot behavior. It also clarifies what the planner does not attempt to solve. The planner does not compute socially compliant trajectories, avoid local collisions, or reason about the exact pose of each visitor during navigation. Those responsibilities remain with the navigation and perception layers. The planner's role is to decide which topological commitment is most promising before those lower layers are invoked. In the running example, this means that the planner decides whether the next abstract commitment should be “go toward  $P_2$  through the short corridor”, “go toward  $P_2$  through the side passage”, “wait”, or “visit  $P_3$  first”. The

concrete navigation controller still decides the detailed trajectory once that commitment is selected.

### 9.1.1 Action Representation in the Tour MDP

In the flow-aware planner, actions are represented as controllable choices in the MDP state-transition system, not as executable middleware calls. Given a topological map  $\mathcal{T}$ , the action set is  $A = \mathcal{E} \cup \{\text{wait}\}$ . An edge action  $e = (v_i, v_j)$  represents the decision to move from node  $v_i$  to node  $v_j$ . It abstracts away the geometric path, local obstacle avoidance, controller commands, and the concrete ROS 2 interactions used to execute navigation. At a state  $s = (v, t, \omega)$ , only outgoing edges from the current node  $v$  are enabled, together with the waiting action when the time bound still allows it.

The effect of an edge action is stochastic because the future congestion on the selected edge is uncertain. For each congestion band  $c^j$  with non-zero probability at time  $t$ , executing  $e = (v, v')$  leads to an outcome in which the robot reaches  $v'$ , the visited set is updated with  $v'$ , and time advances by the expected traversal duration  $\mathbb{E}[\rho(e, c^j)]$ . The probability of that outcome is the predicted congestion probability  $\Pr(c^j \mid e@t)$ . The cost of the same action is the expected traversal duration over all congestion bands. The waiting action `wait` has a deterministic effect: it leaves the robot at the same node, keeps the visited set unchanged, and advances time by  $d_{\text{wait}}$ .

A policy  $\pi$  therefore maps each abstract planning state to either an edge traversal or a wait decision. During execution, only the first selected action is committed: the robot executes the corresponding topological move, observes the updated crowd state, rebuilds the congestion model, and replans. This receding-horizon use of the policy is what connects the abstract action representation to the BT-based execution architecture. An edge action becomes a selected destination for the scheduler and the `GoToPoi` behavior, while the lower layers remain responsible for realizing that decision through concrete skills, components, and ROS 2 navigation interfaces. For the running example, selecting an edge on the side passage does not mean that the MDP sends velocity commands. It means that the scheduler updates the next topological target, after which the existing `GoToPoi` behavior and navigation stack execute the movement.

## 9.2 Solution Overview

To address Problem 1, the robot must reason about how human congestion will evolve over the course of the tour, and how this will affect navigation durations. The proposed framework addresses this through three steps:

- At first, the current occupancy of the environment is observed, and a distribution over future congestion levels on each edge is computed;
- Following this, an MDP is built that captures the stochastic dynamics of navigation under congestion, and a policy is synthesized to solve the tour problem.
- Finally, the first action of the policy is executed, and the process is repeated at the next decision step until the tour is complete.

In the following sections, the process for computing congestion distributions is described, followed by the approach for building and solving the tour MDP.

## 9.3 Modeling Congestion

This subsection describes how congestion is predicted and modeled to support planning. In particular, the probability of different congestion levels on a topological edge  $e$  at time  $t$  is computed. Combining this with the robot duration distributions in Def. 1 allows the robot to reason over how long it will take to traverse an edge at time  $t$ . The process for computing congestion probabilities is summarized in Figure 9.1.

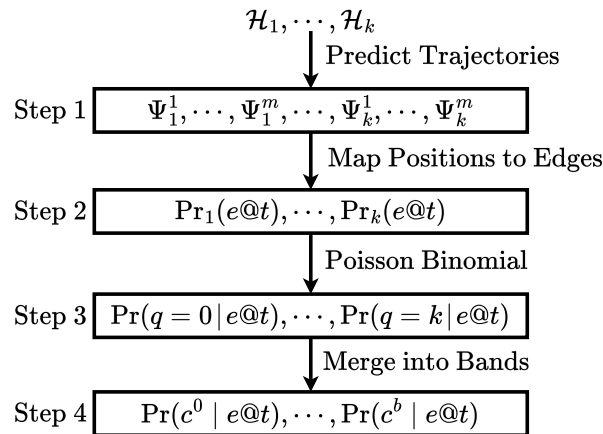


Figure 9.1 Approach for computing human congestion probabilities from a CLiFF map.

The input to Figure 9.1 is a set of observation histories  $\mathcal{H}_1, \dots, \mathcal{H}_k$  for the  $k$  humans currently observed in the environment. At each decision step, the humans currently in the environment are first observed. This could be achieved using onboard sensors, or external sensors located around the environment. Here, external sensors are assumed to provide visibility of all humans currently in the environment.

Given this, the first step is to predict  $m$  trajectories for each human using the CLiFF-LHMP approach outlined in Section 7.3. For a given human, each trajectory is assigned probability  $\frac{1}{m}$ . Human motion is often uncertain and erratic, and so predicting a single trajectory for each human would produce an inaccurate congestion model. By sampling multiple trajectories, the underlying trajectory distribution is approximated which admits a congestion model that explicitly captures motion uncertainty. This is a higher level of abstraction than the trajectories sampled with CLiFF-LHMP, which predict a human's position in 2D space.

Therefore, the second step in computing congestion probabilities is to map predicted human positions onto topological edges. For this purpose, a rectangle  $R_e$  is built around each edge  $e$ , with the edge lying in the middle of the rectangle. A human is on  $e$  at time  $t$  if their position lies inside  $R_e$ . As rectangles for different edges may overlap, a human may be on multiple edges at once. This captures the idea that if two edges are close together, a human walking between them may affect robot navigation on either edge. The rectangular assignment is intentionally simple. It does not attempt to reconstruct the exact area swept by the local planner, but it provides a transparent and tunable approximation from continuous predicted positions to topological edge occupancy. The width of the rectangle controls the conservativeness of this mapping: a narrow rectangle may miss people who still affect navigation, while a wide rectangle may attribute the same person to several nearby edges. This tradeoff is revisited in the evaluation and discussion because it affects both model fidelity and the size of the induced planning problem. In the running example, visitor samples that fall inside the rectangle around the short corridor increase the probability that this corridor will be congested at the future time when the robot would traverse it. Samples that fall near the side passage instead affect the alternative route.

**Example 1** Consider the topological map in Figure 9.2. The red boxes show the rectangles  $R_e$  expanded around each edge. Position  $p_1$  is on edge  $(v_1, v_2)$  as it lies inside  $R_{(v_1, v_2)}$ . Position  $p_2$  is on edge  $(v_1, v_2)$  and  $(v_1, v_3)$ , as it lies in the intersection of  $R_{(v_1, v_2)}$  and  $R_{(v_1, v_3)}$ .

Given a mapping from predicted positions to topological edges, the probability that human  $i$  is on edge  $e$  at time  $t$  can be computed by counting the sampled trajectories that fall inside  $R_e$  and multiplying by  $\frac{1}{m}$ :

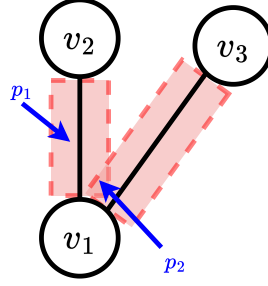


Figure 9.2 Mapping human positions to topological edges. Position  $p_1$  is mapped to  $(v_1, v_2)$ . Position  $p_2$  is mapped to both  $(v_1, v_2)$  and  $(v_1, v_3)$ .

$$\Pr_i(e@t) = \frac{1}{m} \cdot \sum_{x \in \llbracket 1, m \rrbracket} \Psi_i^x(t) \in R_e. \quad (9.1)$$

Step three in computing the congestion probabilities is to compute the distribution over how many humans are on edge  $e$  at time  $t$ . This describes how congested the edge will be. For this purpose, the Poisson binomial distribution is used (Def. 5), since it models the outcome of multiple trials in which each trial is described by a different Bernoulli random variable. In this instance, the Bernoulli random variables model the probability of each human being present on  $e$  at time  $t$ , as computed by equation (9.1).  $\Pr(q \mid e@t)$  is used to denote the probability of  $q$  humans being present on edge  $e$  at time  $t$ . This model is appropriate when the planner needs a distribution over crowd count rather than a set of individual identities. It also allows different people to have different probabilities of occupying the same edge, which would be lost in a standard binomial model. The price of this compactness is the independence assumption between the Bernoulli edge-presence variables. In real crowds, people may move in groups or react to each other; this means the Poisson-binomial distribution should be read as a tractable approximation of edge-level congestion, not as a complete crowd-interaction model.

During execution, there may be little impact on robot navigation duration for similar numbers of humans. For example, for an edge in a wide corridor, the effect of three humans being present will be similar to four. Therefore, in the final step of Figure 9.1, the probabilities for each number of humans  $\Pr(q \mid e@t)$  are merged into *congestion bands*.

**Definition 9** Given an edge  $e$  where  $k$  is the maximum number of humans, a set of congestion bands for  $e$  is given by  $\mathcal{C}_e = \{c^0, \dots, c^b\}$ , where  $c^j = \llbracket lb^j, ub^j \rrbracket$  is an interval over the number of humans. The first band  $c^0 = \llbracket 0, 0 \rrbracket$  captures an empty edge. The final band  $c^b = \llbracket lb^b, k \rrbracket$  is

*bounded by the maximum number of humans. Congestion bands do not intersect, and each number of humans fits in one band, i.e.  $lb^{j+1} = ub^j + 1$ .*

By using congestion bands, the branching factor of the planning model defined in the following section is reduced, improving scalability and computational efficiency. Further, congestion bands help when there is limited data for building duration distributions, as data is aggregated. They also encode the fact that navigation duration is usually not sensitive to every individual count difference. For example, the difference between three and four people in a wide corridor may be negligible, whereas the difference between zero and one person may be important if the robot can otherwise move unobstructed. Thus, congestion bands are both a computational device and a modeling device: they reduce the MDP branching factor while preserving the crowd-density distinctions that are expected to matter for task-level routing. In the running example, the planner does not need to distinguish every possible visitor count in the short corridor. It is often enough to distinguish an empty corridor, a lightly occupied corridor, and a crowded corridor, because these cases induce different expected traversal times and may change whether waiting or taking the side passage is preferable. At a slight abuse of notation, for a topological map  $\mathcal{T}$ ,  $\rho(e, c^j)$  is used to denote the duration distribution for band  $c^j$  on edge  $e$ . The final congestion probabilities are denoted  $\Pr(c^j \mid e@t)$ , i.e. the probability of band  $c^j$  on edge  $e$  at time  $t$ . These are computed by summing the values  $\Pr(q \mid e@t)$  inside each congestion band:

$$\Pr(c^j \mid e@t) = \sum_{q \in c^j} \Pr(q \mid e@t). \quad (9.2)$$

## 9.4 Tour Planning under Human Congestion

With the previous information, an MDP for tour planning is incrementally built and solved using the congestion distributions to reason over the effect of crowds on robot navigation. Each navigation action in the tour MDP has a probabilistic outcome for each congestion band, and the stochastic dynamics of robot navigation are approximated using the expected duration under each congestion band. This approximation improves the tractability of planning. The tour MDP is similar to the single-robot MDPs used for multi-robot planning under congestion in Street et al. [2022], but adapted for a single robot in human-populated environments.

**Definition 10** *Given topological map  $\mathcal{T}$  and time bound  $D \in \mathbb{R}_{\geq 0}$ , the tour MDP is a tuple  $\mathcal{M} = \langle S, \bar{s}, A, T, C, G \rangle$ , where:*

- $S = \mathcal{V} \times \mathbb{R}_{\geq 0} \times 2^{\mathcal{V}}$ . States describe the robot's current node/POI, the current time, and the set of POIs already visited on the tour.
- $\bar{s} = (\bar{v}, \bar{t}, \emptyset)$  for the first decision step, where  $\bar{v}$  and  $\bar{t}$  are the starting location and time for the tour, respectively. In subsequent decision steps,  $\bar{s}$  is set to the current state.
- $A = \mathcal{E} \cup \{\text{wait}\}$ . The robot can navigate along edges of the topological map or wait for duration  $d_{\text{wait}}$ . Waiting allows the robot to hold off from traversing an edge until it is less congested. At each state the robot can navigate along edges that are outgoing from its current node.
- The transition function  $T$  captures the effect of robot navigation and waiting under congestion. If a robot navigates along an edge, there is an outcome for each congestion band with non-zero probability at that time. Upon completing a navigation action, the robot's location is updated to the destination of the edge, the destination is added to the visited set, and the time is increased by the expected edge duration under the corresponding congestion band. Wait actions increase the time by  $d_{\text{wait}}$ , with the location and visited set remaining the same. Adding time to the state makes the state space infinite. Therefore, a time bound  $D$  is added, so that actions can only be executed before  $D$ . Any state with a time greater than  $D$  becomes a dead end. If  $D$  is set greater than the duration of any reasonable tour, solution quality will be unaffected. By using time bound  $D$  and expected edge durations, the MDP state space becomes finite. Formally, for states  $s = (v, t, \omega)$ ,  $s' = (v', t', \omega')$ , and action  $e$ :

$$T(s, e, s') = \begin{cases} \Pr(c^j \mid e@t) & e = (v, v'), \\ & t < D, \\ & t' = t + \mathbb{E}[\rho(e, c^j)] \text{ and} \\ & \omega' = \omega \cup \{v'\} \\ 1 & e = \text{wait}, \\ & t' = t + d_{\text{wait}}, \\ & t < D \text{ and} \\ & \omega' = \omega \\ 0 & \text{otherwise.} \end{cases} \quad (9.3)$$

- The cost function  $C$  captures the expected navigation duration over all congestion bands. Formally, for state  $s = (v, t, \omega)$  and action  $e$ :

$$C(s, e) = \begin{cases} \sum_j \Pr(c^j \mid e@t) \cdot \mathbb{E}[\rho(e, c^j)] & e \in \mathcal{E} \\ d_{wait} & e = wait. \end{cases} \quad (9.4)$$

- $G = \{(v, t, \omega) \in S \mid \omega = \mathcal{V} \text{ and } t < D\}$ , i.e. the robot reaches the goal if it completes the tour within time bound  $D$ .

This definition relies on a small set of modeling assumptions. The topological state  $(v, t, \omega)$  is assumed to contain the information needed for task-level decisions: the current region, the current time, and the POIs already visited. The continuous geometry of the robot and pedestrians is abstracted into edge-level congestion bands, so local collision avoidance and social motion control remain delegated to the navigation layer. Human-presence probabilities on an edge are computed from sampled trajectories and then combined through a Poisson-binomial model, which assumes independence between the Bernoulli variables used for each person's edge presence at a given time. Finally, each stochastic outcome uses the expected traversal duration for the corresponding congestion band. This choice loses information about variance and rare delays, but it keeps the MDP compact enough to be solved repeatedly during execution. The modeling decisions can therefore be summarized as a sequence of controlled approximations. Continuous pedestrian predictions are approximated by edge-presence probabilities; individual edge-presence probabilities are approximated by a distribution over crowd counts; crowd counts are grouped into congestion bands; and traversal-time distributions are represented in the MDP through their expected values. Each approximation removes some detail, but each also removes a source of combinatorial growth. The scientific question tested in the evaluation is whether the retained information, namely the predicted stochastic evolution of edge congestion, is sufficient to improve task-level tour decisions.

Though the time bound  $D$  constrains the state space, tour MDP  $\mathcal{M}$  is still often too large to solve exactly on-line. Therefore, to facilitate on-line planning,  $\mathcal{M}$  is solved using labeled real-time dynamic programming (LRTDP) (Bonet and Geffner [2003]). LRTDP is an anytime, trial-based heuristic search algorithm that relies on an admissible heuristic  $h : S \rightarrow \mathbb{R}_{\geq 0}$  to guide the search process. The choice of LRTDP is motivated by the size of the induced MDP and by the on-line nature of the target application. Exact dynamic-programming approaches would require exploring a very large state space before execution, while in practice the robot only visits a small subset of the reachable states during a single tour. LRTDP is therefore a



natural compromise, as it concentrates computation on the states that are most relevant to the current execution trace.

The heuristic also plays a central role in making the planner practical. For tour planning, the heuristic should be a lower bound on the remaining time to complete the tour. A minimum spanning tree (MST) approach is used, as this is a common heuristic for the TSP (Cormen et al. [2022]). The MST-based heuristic provides an optimistic estimate of the remaining tour cost while preserving the possibility of revisiting nodes through shortest paths in the underlying graph. This is an important detail for this domain, where the best route may require moving through already visited areas in order to avoid temporary congestion elsewhere.

Formally, for a state  $s = (v, t, \omega)$ , a new topological map  $\mathcal{T}_s$  is computed by removing all nodes in  $\omega \setminus \{v\}$ , i.e. the nodes that have already been visited. Each pair of nodes  $(v, v')$  in  $\mathcal{T}_s$  is connected, where the edge duration is the expected duration of the shortest path between  $v$  and  $v'$  in  $\mathcal{T}$ , assuming no congestion. The heuristic value  $h(s)$  is then the sum of all expected edge durations on the MST for  $\mathcal{T}_s$ . Using the shortest path durations in  $\mathcal{T}_s$  allows nodes to be revisited without requiring them in the MST. Once the tour MDP is built and solved, the first action of the resulting policy is then executed before repeating the process at the next decision step.

#### 9.4.1 Why Receding-Horizon Replanning

An important design choice in this work is to solve the tour-planning problem on-line in a receding-horizon fashion instead of computing a single fixed plan at the beginning of the tour. The main reason is that museum environments are highly dynamic: visitors can enter or leave the scene at any time, and congestion may shift from one room to another while the robot is executing the tour. Under these conditions, a plan computed only once quickly becomes outdated.

By replanning after each decision step, the robot can exploit newly observed human trajectories and update its estimate of future congestion. This design trades global commitment for responsiveness, which is particularly important in public environments where the robot must remain adaptive and where suboptimal local decisions may accumulate if the planner ignores the evolution of the crowd. Receding-horizon replanning also reduces the consequences of prediction error. The planner does not need a perfect forecast for the entire tour; it needs a useful forecast over the next portion of the tour, after which new observations can correct the model. This is why short-horizon CLiFF-LHMP predictions are a good fit for the proposed architecture.

# Chapter 10

## Evaluation of Flow-Aware Planning

This chapter evaluates the proposed tour planner using human data derived from a real-world dataset. The aim of this evaluation is twofold: to assess whether flow-aware planning improves tour execution time, and to examine whether the resulting planning times remain compatible with on-line deployment. All experiments are run in a Ubuntu 24.04 Docker image on a machine with an AMD EPYC 7513 CPU @ 2.60 GHz (max 3.65 GHz), 32 cores and 64 threads, 1 TB of RAM, and an NVIDIA A100 GPU. An open-source Python implementation of the approach is available on GitHub<sup>1</sup>.

The evaluation is organized around four questions:

- i. Does modeling future congestion improve tour quality compared with a planner that only reacts to current congestion?
- ii. Does the bounded on-line version retain most of the benefit of the converged planner?
- iii. How sensitive are performance and planning time to the granularity of the congestion abstraction?
- iv. Under which assumptions should the results be generalized to real robotic deployments?

These questions are chosen to separate the algorithmic contribution from the engineering integration: the experiments test whether the stochastic flow-aware abstraction itself provides useful decisions under uncertain human motion.

The evaluation should therefore be interpreted as a controlled algorithmic study rather than as a complete deployment study. It deliberately fixes the robot navigation model, the

---

<sup>1</sup><https://github.com/convince-project/congestion-coverage-plan>

topological maps, the sensing assumptions, and the execution simulator so that the effect of the planning abstraction can be isolated. This makes the comparison with the Hamiltonian-path baseline meaningful, because both planners face the same replayed human flows and differ mainly in whether they model future congestion stochastically. At the same time, this setup does not exercise all sources of uncertainty present on a physical robot, such as localization drift, perception failures, human reactions to the robot, or delays introduced by the full middleware stack. Those issues are considered later in the museum case study and in the discussion of limitations. The evaluation is therefore best read as a validation of the planner’s internal scientific claim, not as a claim that all deployment risks have been eliminated. It answers whether representing predicted congestion as stochastic task-level action outcomes changes decisions in a useful way. It does not, by itself, answer whether the same quantitative gains would hold under every sensing setup, local-navigation controller, or pattern of human-robot interaction.

## 10.1 Experimental Setup

The planner is evaluated by building synthetic simulations from the *ATC* shopping mall dataset (Brščić et al. [2013]). The evaluation is performed by using 50% of the data to build a CLiFF map using the implementation provided by the original authors<sup>2</sup> and the remaining 50% to build the synthetic execution. To study scalability, four topological maps of different sizes were generated. The size of each map is shown in Table 10.1, and the maps themselves are shown in Figs. 10.1 and 10.2.

| Nodes | Edges |
|-------|-------|
| 11    | 46    |
| 16    | 70    |
| 21    | 98    |
| 26    | 122   |

Table 10.1 The sizes of the maps created in the ATC domain.

For each topological map three different sets of congestion bands are considered, one with two bands, one with five, and one with eight. This allows us to analyze the effect of model granularity on both execution performance and planning time. For each set of congestion bands, there is always a single congestion band for an empty edge (see Def. 9), and then the remaining congestion bands are divided equally. The bands are detailed in Table 10.2. Using

<sup>2</sup><https://github.com/tkucner/CLiFF-map-matlab>

several band granularities is an explicit test of a modeling decision made in the framework chapter. If very fine bands were required to obtain good behavior, the approach would be less attractive for on-line use because the MDP branching factor would grow quickly. If coarser bands perform similarly, then the planner can retain the information that matters for task-level decisions while keeping the stochastic model compact.

| # of bands | Bands                                                                                                   |
|------------|---------------------------------------------------------------------------------------------------------|
| 2          | $[[0, 0]]$ , $[[1, max]]$                                                                               |
| 5          | $[[0, 0]]$ , $[[1, 2]]$ , $[[3, 4]]$ , $[[5, 6]]$ , $[[7, max]]$                                        |
| 8          | $[[0, 0]]$ , $[[1, 1]]$ , $[[2, 2]]$ , $[[3, 3]]$ , $[[4, 4]]$ , $[[5, 5]]$ , $[[6, 6]]$ , $[[7, max]]$ |

Table 10.2 The congestion bands used in the experiments.

The framework is evaluated through a synthetic simulation built on the real human trajectories from the dataset. For each experimental run, a time is randomly sampled from the test dataset, then the human movements from this point are replayed and used for the detected occupancies. This data is then mapped to the edges of the topological map using the approach in Figure 9.2, where the rectangles  $R_e$  for each edge  $e$  are 2m wide. The duration for navigating an edge is computed as the nominal traversal time plus a 10 second penalty for each human encountered on that edge. The wait duration is set to 5 seconds based on empirical observations. These parameters should be interpreted as controlled evaluation assumptions rather than universal constants. The human penalty creates a measurable relationship between congestion and task duration, while the wait duration gives the planner an alternative to immediately entering a congested region. Changing these values would change the absolute execution times and may change the relative value of waiting, but it does not change the core comparison between a planner that reasons over future congestion distributions and a baseline that treats current congestion as fixed.

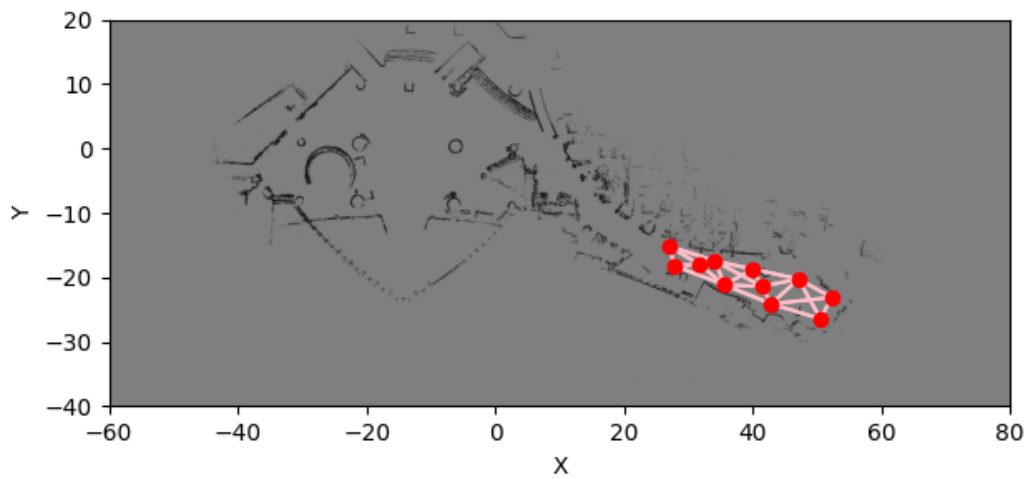
At each time step, the robot senses any humans and uses this to update the CLiFF-LHMP trajectories. Then each trajectory is used to predict the future positions of the humans. For each trajectory, ten future positions are predicted up to 50 seconds ahead. Subsequently the MDP is incrementally constructed through LRTDP with a convergence threshold of 0.1, and a time bound for the planning horizon  $D$  is set to 250 seconds. At this step, there are two variants of the tour planner. The first, referred to as *CPV2* (Coverage Plan Version 2), runs LRTDP until convergence or until 3000 seconds has passed. This is suitable to estimate the maximum performance attainable by the planner, but it is not suitable for on-line use due to the long planning times in some cases. The second, referred to as *CPV2 bounded*,

runs planning for a maximum of 3 seconds per timestep. This is more suitable for on-line planning but may synthesize suboptimal behavior.

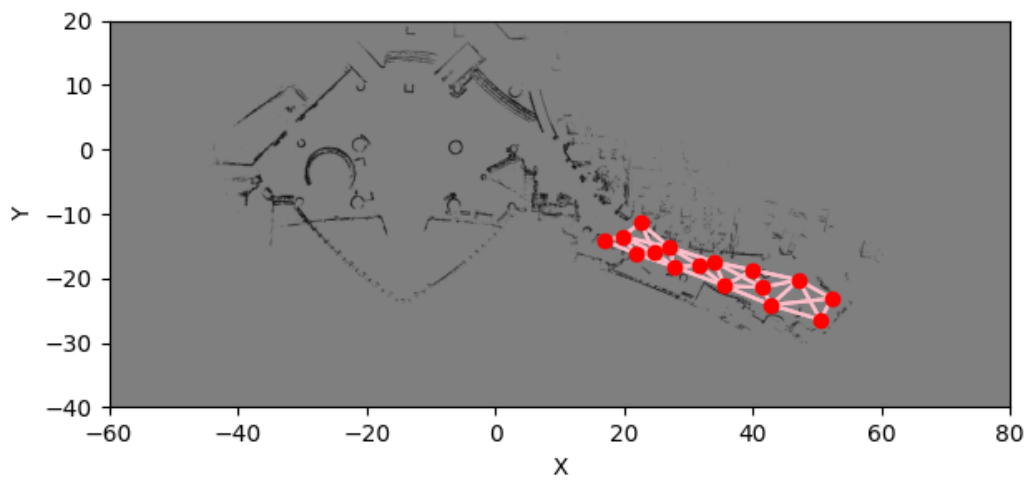
These solutions are then compared to an on-line Hamiltonian-path solver, which serves as a baseline that does not model the stochastic evolution of congestion, but still replans after each action based on the current congestion estimate. For the Hamiltonian path solution, at each step, the current congestion levels are predicted on each edge and are assumed to be fixed for the duration of the planning horizon. Then LKH 3 solver (Helsgaun [2017]) is used to synthesize a valid Hamiltonian path for the remaining POIs. LKH 3 was run with a maximum time limit of 3000 seconds to find the optimal solution, though it often requires less than one second on average.

For each topological map, set of congestion bands, and method, 40 simulations are sampled from the dataset, and from each simulation both planning time and total execution time are recorded. This comparison is useful because it isolates the benefit of reasoning over the future stochastic evolution of congestion, rather than merely reacting to the present occupancy of the environment. In other words, both methods solve the same tour-planning problem at the task level, but only the proposed approach explicitly models uncertainty over future crowd motion. The comparison is therefore not intended to show that MDP solving is faster than a specialized routing solver. Instead, it tests whether the additional stochastic state introduced by the MDP leads to better task-level choices when the environment changes during execution. This baseline is intentionally stronger than a static precomputed tour, because it already receives updated congestion information after each executed action. Consequently, a performance difference in favor of CPV2 cannot be explained simply by the fact that CPV2 replans. It must come from how CPV2 represents the future consequences of the observed flow.

The comparison focuses on two classes of metrics. The first is execution quality, measured through the total duration of the tour. The second is on-line deployability, measured through the time required to compute decisions during execution. These criteria reflect the two main goals of the framework: improving task performance in crowded environments while remaining usable on a real robotic platform. The evaluation does not use social-comfort or collision-avoidance metrics because those are primarily properties of the local navigation layer, not of the task-level route-selection mechanism studied here. This is a limitation of the experimental scope, but it also keeps the evaluation aligned with the algorithmic claim of this part.

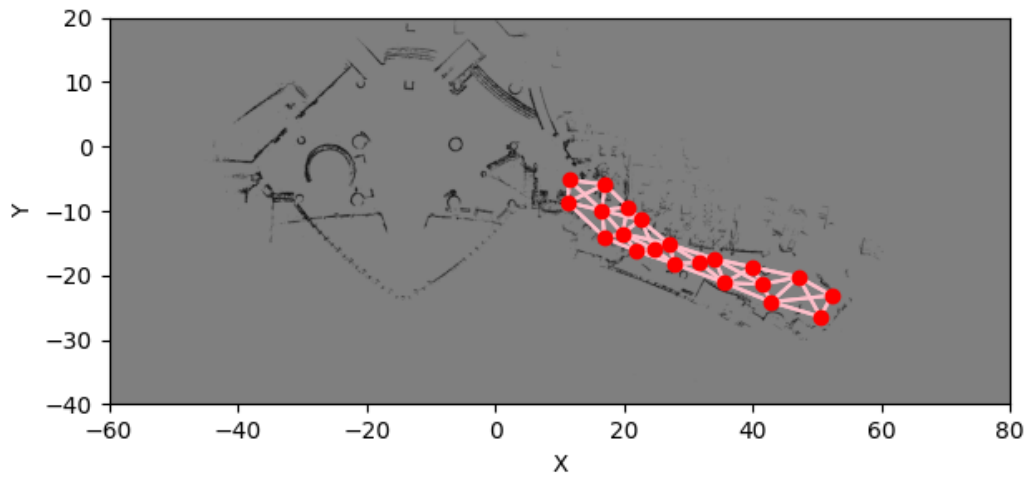


(a) 11 node map.

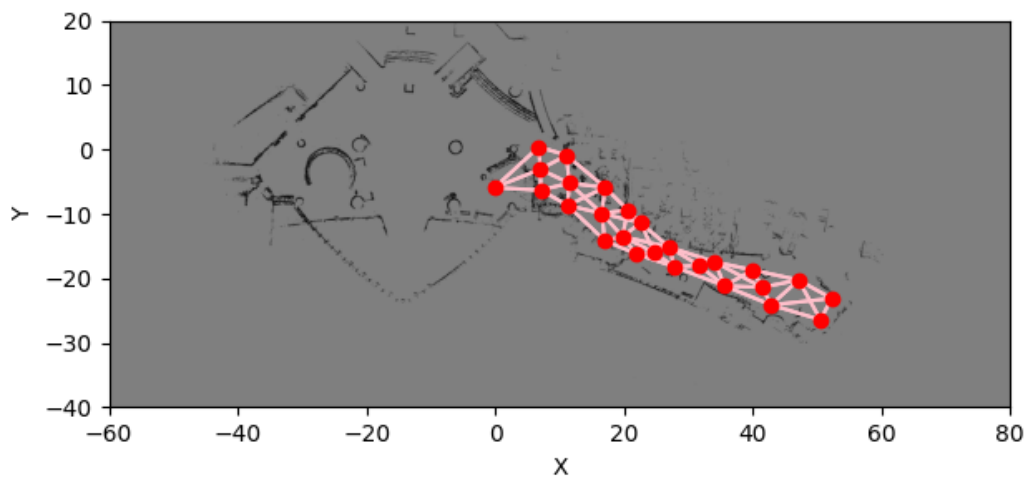


(b) 16 node map.

Figure 10.1 The topological maps for the ATC environment with 11 and 16 nodes.



(a) 21 node map.



(b) 26 node map.

Figure 10.2 The topological maps for the ATC environment with 21 and 26 nodes.

## 10.2 Results

The execution-time results in Figs. 10.3 and 10.4 show a consistent advantage for the proposed flow-aware planner. For every map size and congestion-band configuration, both CPV2 and CPV2 bounded complete the tour faster than the Hamiltonian-path baseline. For example, on the largest 26-node map with five congestion bands, the average execution times are 109.12 seconds for CPV2, 109.71 seconds for CPV2 bounded, and 132.09 seconds for the Hamiltonian path solver. This corresponds to a reduction of about 17% in average execution time for both CPV2 variants in this setting, while keeping the same task, map, and replayed pedestrian-flow conditions. These differences are statistically significant under a one-sided Mann-Whitney U test at  $p = 0.05$ , as reported in Table 10.3.

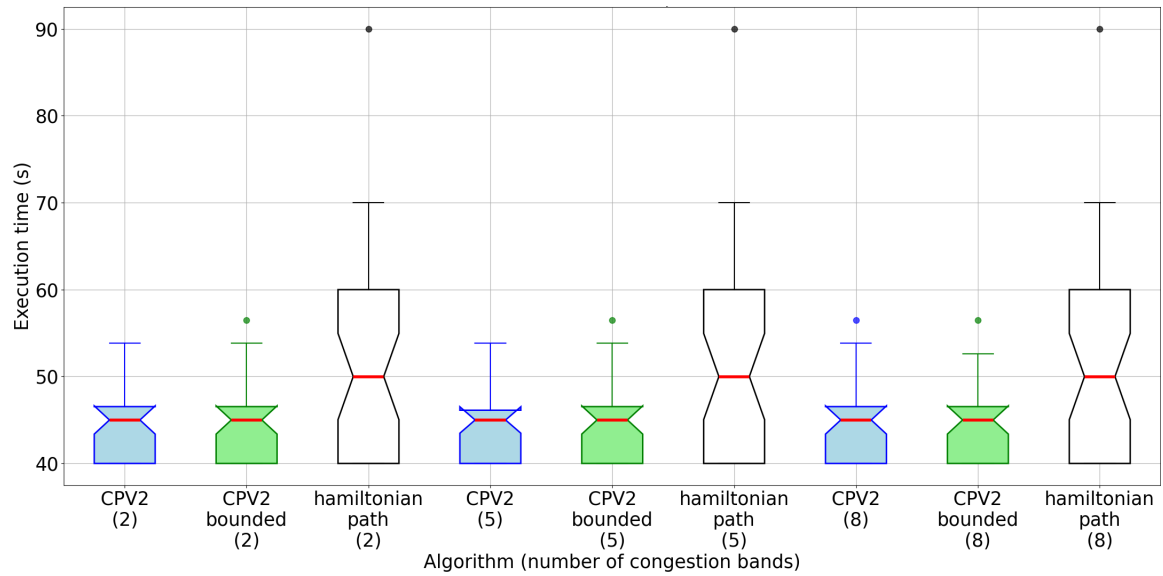
The bounded planner performs nearly as well as the unbounded planner in every setting. Table 10.3 confirms that the execution-time differences between CPV2 and CPV2 bounded are not statistically significant at the  $p = 0.05$  level, suggesting that the limited on-line computation budget still yields high-quality decisions.

The results also indicate that increasing the number of congestion bands does not substantially change the execution-time performance of the CPV2 methods. This implies that a coarse congestion model may be sufficient for effective tour planning in these crowded environments, and that adding finer granularity yields diminishing returns for task performance. The planning-time results in Figure 10.5 reinforce this conclusion. The median planning time remains low across different congestion-band settings, while the worst-case planning time grows predictably with map size as the MDP branching factor and state space increase. Importantly, even for the largest map the median planning time is on the order of two seconds, which is compatible with on-line deployment. Together, these findings support the claim that flow-aware planning can improve tour execution without sacrificing real-time usability on a robotic platform. The result also clarifies where the approach is likely to generalize: the strongest expected benefit is in environments with repeated pedestrian-flow patterns, multiple feasible route choices, and tour objectives that permit the robot to exploit those alternatives. Conversely, the expected benefit is smaller when the graph has few alternative routes, when congestion is uniformly distributed over all routes, or when the task order is so constrained that the robot cannot exploit less congested alternatives.

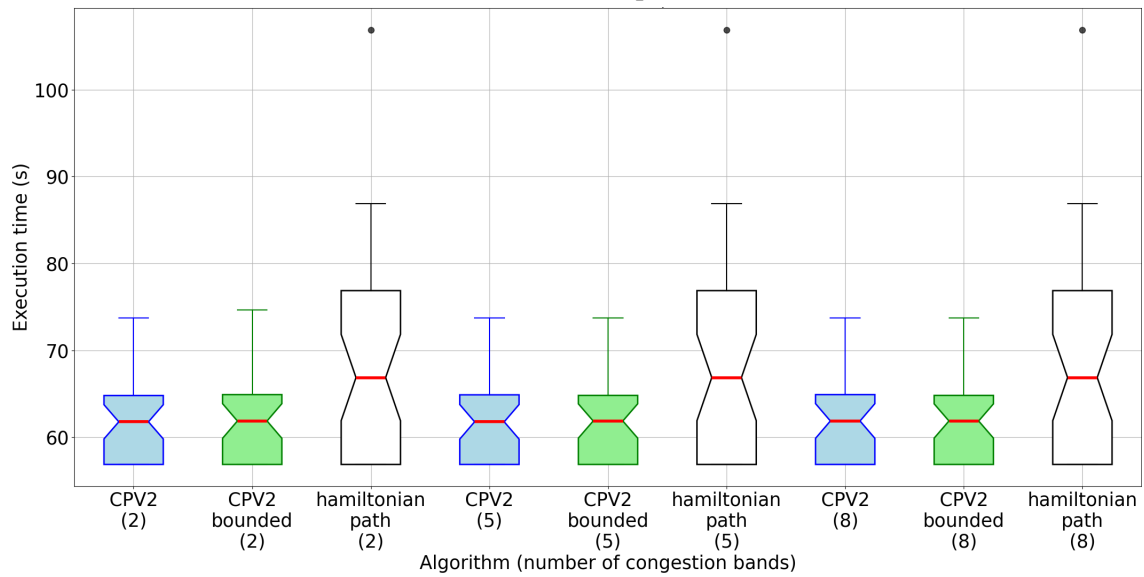


| Map      | Congestion Bands | CPV2 < Hamiltonian                 | CPV2 bounded < Hamiltonian         | CPV2 < CPV2 Bounded |
|----------|------------------|------------------------------------|------------------------------------|---------------------|
| 26 nodes | 2                | <b><math>2.67 * 10^{-9}</math></b> | <b><math>8.77 * 10^{-9}</math></b> | 0.43                |
|          | 5                | <b><math>2.67 * 10^{-9}</math></b> | <b><math>1.43 * 10^{-8}</math></b> | 0.33                |
|          | 8                | <b><math>6.19 * 10^{-9}</math></b> | <b><math>1.40 * 10^{-8}</math></b> | 0.39                |
| 21 nodes | 2                | <b><math>8.47 * 10^{-4}</math></b> | <b><math>2.84 * 10^{-3}</math></b> | 0.44                |
|          | 5                | <b><math>6.01 * 10^{-4}</math></b> | <b><math>8.49 * 10^{-4}</math></b> | 0.42                |
|          | 8                | <b><math>1.09 * 10^{-3}</math></b> | <b><math>1.09 * 10^{-3}</math></b> | 0.58                |
| 16 nodes | 2                | <b><math>2.05 * 10^{-3}</math></b> | <b><math>2.95 * 10^{-3}</math></b> | 0.42                |
|          | 5                | <b><math>2.31 * 10^{-3}</math></b> | <b><math>3.70 * 10^{-3}</math></b> | 0.45                |
|          | 8                | <b><math>2.26 * 10^{-3}</math></b> | <b><math>1.36 * 10^{-3}</math></b> | 0.59                |
| 11 nodes | 2                | <b><math>6.25 * 10^{-3}</math></b> | <b><math>4.19 * 10^{-3}</math></b> | 0.54                |
|          | 5                | <b><math>4.88 * 10^{-3}</math></b> | <b><math>3.69 * 10^{-3}</math></b> | 0.50                |
|          | 8                | <b><math>5.05 * 10^{-3}</math></b> | <b><math>2.90 * 10^{-3}</math></b> | 0.55                |

Table 10.3 The Mann-Whitney U p-values comparing CPV2 and CPV2 bounded to the Hamiltonian path solver, and p-values comparing CPV2 against CPV2 bounded. Values in bold are less than  $p = 0.05$  and are statistically significant.

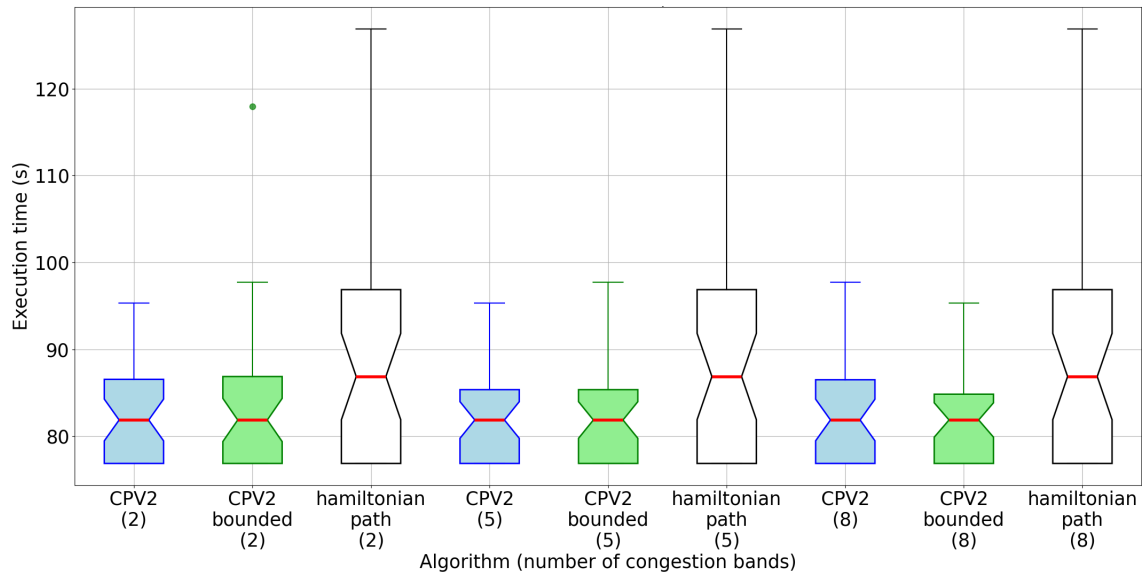


(a) 11 node map.

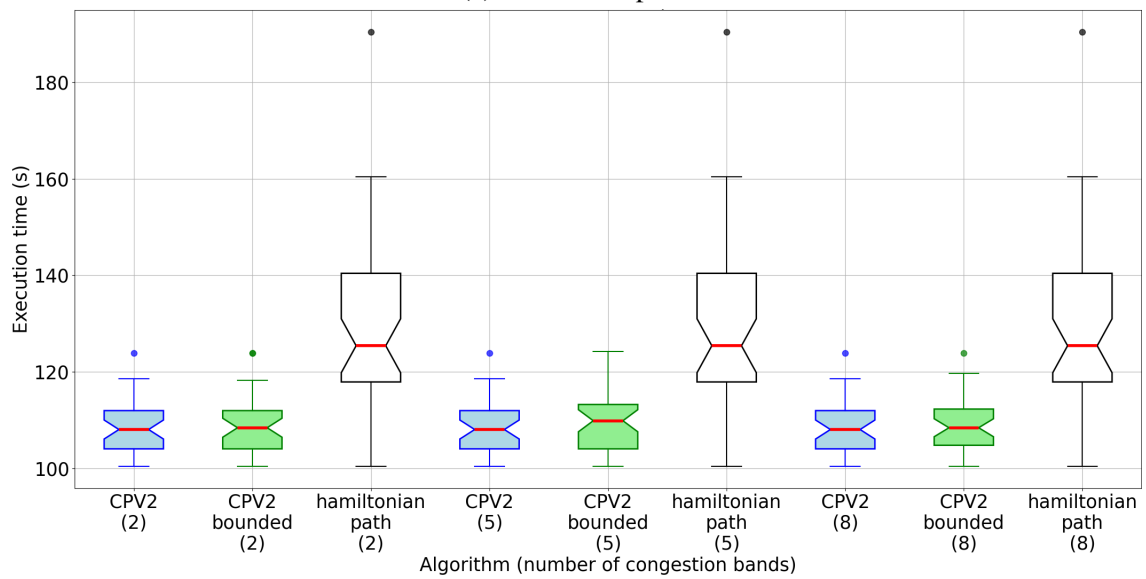


(b) 16 node map.

Figure 10.3 Execution time of the CPV2 solution (blue), the CPV2 bounded solution (green), and the Hamiltonian-path solution (white) for 11 and 16 maps sizes.



(a) 21 node map.



(b) 26 node map.

Figure 10.4 Execution time of the CPV2 solution (blue), the CPV2 bounded solution (green), and the Hamiltonian-path solution (white) for 21 and 26 maps sizes.

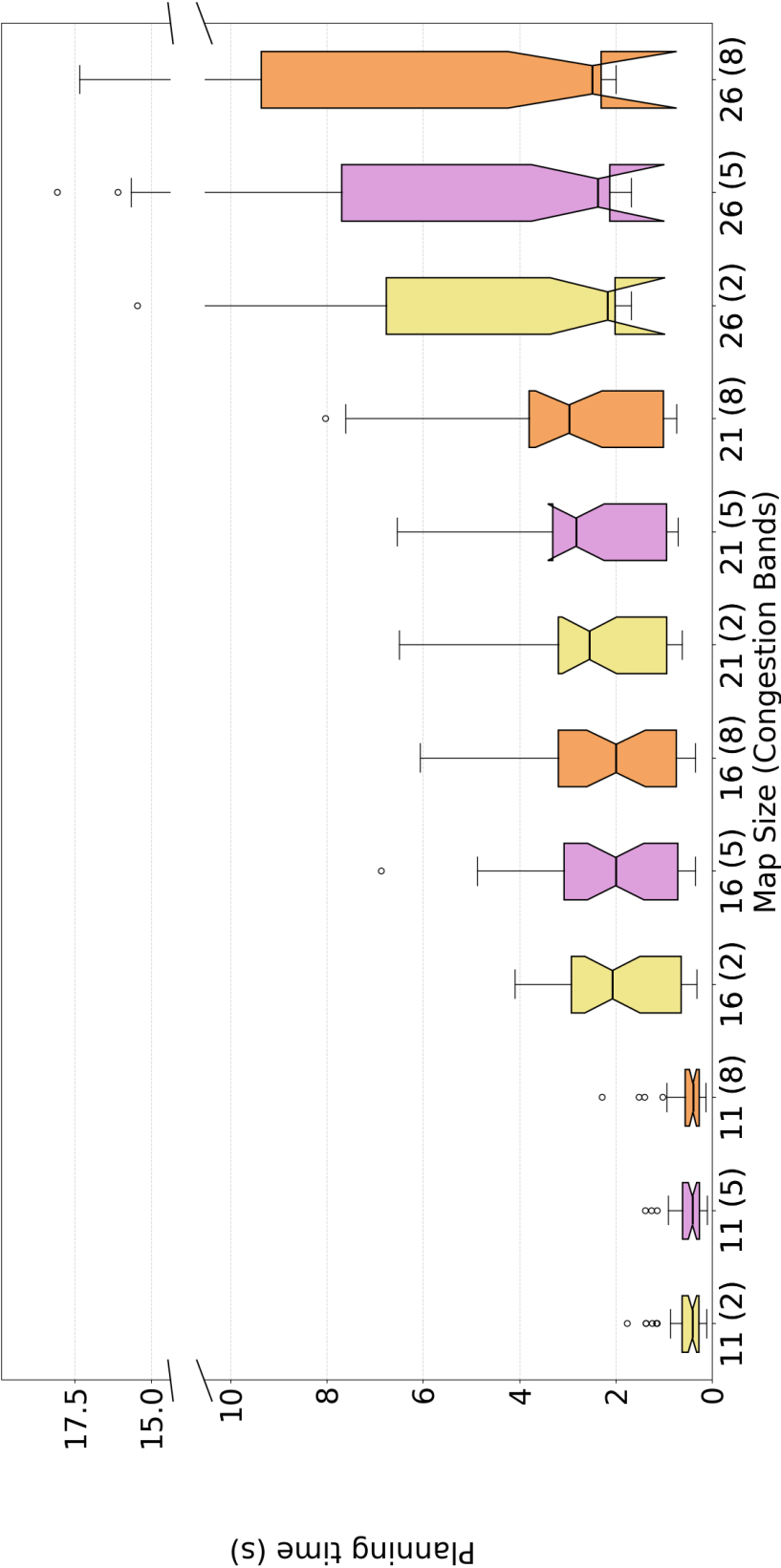


Figure 10.5 Planning time for the first CPV2 planning step under different numbers of congestion bands on different map sizes.

### 10.2.1 Answers to the Evaluation Questions

The four questions introduced at the beginning of this chapter can now be answered as follows.

- i. **Does modeling future congestion improve tour quality?** Yes, within the controlled experimental setting. Across every tested map size and congestion-band configuration, both CPV2 variants produce shorter tours than the Hamiltonian-path baseline, even though the baseline also replans from updated congestion observations. On the 26-node map with five congestion bands, the reduction in average execution time is approximately 17%. Figures 10.3 and 10.4, together with Table 10.3, show that this advantage is consistent and statistically significant.
- ii. **Does bounded planning retain the benefit of converged planning?** Yes. CPV2 bounded achieves execution times close to those of converged CPV2 in every tested configuration, while limiting computation to three seconds per decision step. The CPV2–CPV2 bounded comparisons in Table 10.3 are not statistically significant at  $p = 0.05$ . Full convergence is therefore not required to preserve most of the observed tour-quality benefit in these scenarios.
- iii. **How sensitive is the method to congestion granularity?** Execution quality is only weakly sensitive to the tested granularity. Moving from two to five or eight congestion bands yields little additional reduction in tour duration, indicating that a coarse abstraction captures most of the useful planning signal. Computational cost is more sensitive: Figure 10.5 shows that finer granularity and larger maps increase the upper tail of planning time because they enlarge the MDP branching factor and reachable state space. The median planning time nevertheless remains compatible with on-line use in the tested configurations.
- iv. **Under which assumptions can the results be generalized?** Generalization is conditional rather than universal. The strongest transfer is expected in structured public indoor environments where pedestrian flow is recurrent enough to be learned, the topological map contains meaningful route alternatives, task ordering is sufficiently flexible to exploit them, and human observations and edge-duration models are reasonably reliable. The experiments do not establish end-to-end robustness to perception failures, localization drift, visitor reactions, correlated group motion, or rare high-congestion events. These limitations and modeling assumptions are examined in Sections 11.1

---

and 11.2; and the museum study in Section 15.5 provides complementary evidence from integration with the full tour-guide software stack.

# Chapter 11

## Discussion

The experimental campaign supports the part’s central claim: explicitly modeling the future stochastic evolution of congestion improves long-horizon tour planning.

The results demonstrate that the proposed flow-aware planner is able to reduce execution time compared to a baseline that only reacts to current congestion, while still remaining feasible for on-line deployment in the evaluated scenarios. This comparison is intentionally internal to the experimental setup. A direct numerical comparison with prior human-aware navigation or social-navigation methods would be misleading, because those methods usually optimize local motion, collision avoidance, or social comfort rather than the order of POIs in a long-horizon tour. The relevant result is therefore the gap between two task-level planners exposed to the same pedestrian traces: one that treats current congestion as fixed and one that reasons over its predicted stochastic evolution. Under that controlled comparison, the improvement indicates that the contribution comes from modeling future flow, not only from replanning frequently. The numerical effect is also meaningful: on the largest ATC map with five congestion bands, the representative improvement over the baseline is about 17% in average execution time, and the bounded planner achieves nearly the same performance as the converged planner. At the same time, the evaluation is based on synthetic execution traces generated from real pedestrian trajectories. This means the experiments are a controlled validation of the planning principle, but they do not capture perception failures, localization drift, or robot-visitor interactions that may occur in a real deployment. Therefore, the synthetic results should be interpreted as evidence of the planner’s potential rather than as a complete end-to-end assessment of a deployed robotic system. The following museum chapter complements this analysis by describing how the planner integrates into the full architecture and by highlighting practical issues encountered in a real use case. The generalization claim is consequently conditional: the method is most likely to transfer to

public indoor environments where human flow is recurrent enough to be learned, where the topological graph offers meaningful alternatives, and where task order has enough flexibility for congestion-aware decisions to matter. This is also the most appropriate way to position the contribution with respect to prior planning methods. The thesis does not claim to outperform specialized local social-navigation methods on their own metrics, nor to replace deterministic routing solvers when costs are static. It claims that when travel costs are shaped by predictable but uncertain human flow, task-level planning benefits from carrying that uncertainty into the decision model instead of reducing it to a single current-cost snapshot.

A second important observation concerns planning complexity. Planning time grows with both the size of the topological graph and the granularity of the congestion model. In the extreme case of widespread congestion, the resulting state-space expansion can become substantial, as reflected by the upper tails of the planning-time distributions. In practice, however, these high-branching scenarios are often localized, which limits their impact and keeps on-line planning viable.

The scalability analysis highlights an important tradeoff: planning complexity increases with both the number of topological decision points and the expressiveness of the congestion model. Both factors enlarge the reachable state space of the MDP and therefore raise computational cost. Nevertheless, the measurements show that this growth remains compatible with on-line use in the tested scenarios, especially when bounded planning is used. This is encouraging for environments of similar scale, but it also suggests that larger or denser maps may require additional approximations, hierarchical abstractions, or decomposition strategies.

## 11.1 Uncertainty and Evaluation Scope

Uncertainty enters the planner at three different points. The first is human-motion prediction: CLiFF-LHMP produces sampled future trajectories rather than a single deterministic forecast. The second is congestion abstraction: predicted positions are mapped to topological edges and then aggregated into congestion bands. The third is execution duration: each congestion band is associated with a distribution over traversal time, but the current MDP uses its expected value when constructing transition costs. The experiments therefore evaluate a planner that reasons explicitly about uncertainty in where people may be, while still using a compact expected-duration approximation for how that uncertainty affects navigation time. These sources of uncertainty have different interpretations. Trajectory sampling captures aleatoric uncertainty in where currently observed people may move. The learned CLiFF model and



the edge-duration distributions also introduce epistemic uncertainty, because they depend on finite data and on how well the training environment represents the deployment environment. The current planner handles the first source directly through stochastic congestion outcomes, but it handles the second only indirectly through repeated observation and replanning.

This choice is appropriate for the objective studied in this thesis, namely on-line task-level tour planning. It would be less appropriate for safety-critical navigation guarantees, where variance, worst-case delays, and rare dense-crowd events should be modeled directly. Similarly, the independence assumption used when aggregating individual edge-presence probabilities is reasonable as a compact approximation, but it may underestimate correlations when visitors move as groups. The use of expected traversal duration further reduces the risk sensitivity of the planner: two edges with the same expected duration but different tail risks are treated equivalently by the current cost model. These choices make the planner fast enough for the evaluated setting, but they should not be mistaken for a complete probabilistic safety model. The synthetic ATC evaluation isolates the planning mechanism because all methods are exposed to the same replayed pedestrian traces and the same edge-duration model. The museum evaluation then tests the same idea inside the full tour-guide software stack. Together, these experiments provide evidence that flow-aware task planning improves efficiency and smoothness, but they do not replace a long-term deployment study with real visitors and all perception, localization, and interaction uncertainties active at the same time. A natural extension would therefore be to evaluate risk-aware variants that include variance or high-percentile traversal costs, and to compare them against the expected-cost version used here.

## 11.2 Limitations of the Model

One of the limitations of this approach is that several evaluation parameters were chosen empirically, including the width of the edge-assignment rectangles, the wait duration, the planning horizon, and the number of congestion bands. These choices influence both model fidelity and computational cost, so it is important to discuss them explicitly. The results indicate that the benefit of additional congestion bands saturates quickly. In the environments studied here, a coarse distinction between low and high congestion captures most of the useful planning signal, while finer granularity brings only modest additional gains. This observation supports the use of a compact congestion representation for on-line planning. Similarly, the bounded CPV2 variant delivers execution times comparable to the unconstrained version.

This suggests that the planner can identify good actions early and that full convergence is not always necessary to obtain effective on-line decisions.

Another limitation of the framework is that it relies on several modeling assumptions. First, it assumes the robot has sufficiently reliable observations of nearby humans, either through onboard sensing or an external infrastructure. Second, it assumes that the topological map and edge-duration models are known in advance and remain valid during execution. Third, the planner operates on a topological abstraction rather than the robot's full continuous geometry. Fourth, congestion effects are represented through discrete bands and expected durations, trading some fidelity for tractability. These assumptions are reasonable for the scenario considered in this thesis, where the environment is structured, tours are repeated, and sensing coverage is available. At the same time, they delimit the scope of the current approach and motivate future work on weaker sensing assumptions, richer motion models, and tighter integration between task planning and local navigation.

Finally, human predictions depend on the underlying CLiFF-based model and the available observation history, so prediction errors can propagate to congestion estimates and affect planner decisions. The use of expected traversal times ignores higher-order distributional properties such as variance and tail risk, which may matter in safety-critical or time-critical contexts. The topological abstraction is effective for long-horizon reasoning, but it cannot capture all geometric subtleties of narrow passages, local crowd interactions, or temporary deadlocks. These limitations do not invalidate the method, but they clarify what the framework can solve and what remains delegated to lower-level navigation or future extensions. They also provide a useful foundation for the future work discussion later in the thesis.

## **Part III**

### **Museum Guide Case Study**

# Chapter 12

## From Dependable Execution to Adaptive Planning

This final part evaluates the methods developed in the previous parts on a complete robotic application: a museum tour guide that must interact with visitors, navigate among points of interest (POIs), and adapt its decisions to crowd dynamics. Before presenting the application, this chapter recaps the two technical foundations on which it is built. Part I addressed the dependability of deliberative robotic software by connecting behavioral models, generated code, off-line verification, and runtime monitoring. Part II addressed adaptation by transforming predictions of human motion into stochastic task-level planning decisions. The museum application brings these contributions together and makes explicit the boundary between an abstract plan and its concrete, monitored execution.

### 12.1 Recap of Model-Based Execution and Monitoring

The first technical part of the thesis introduced a model-based workflow for the development of deliberative robot software. The architecture separates the system into three levels. At the deliberation level, Behavior Trees (BTs) coordinate high-level tasks and determine which capability should be executed next. At the skill level, finite-state models describe reusable robot capabilities and their interactions with the rest of the system. At the functional level, components provide perception, navigation, interaction, and other services through ROS 2 interfaces. This separation keeps the high-level control flow readable while placing the detailed communication logic inside explicit skill models.

The central methodological idea is that a behavioral model should remain useful throughout the development cycle rather than serving only as design documentation. The M2C

generator translates the supported ASCXML skill models into executable ROS 2 C++ packages while retaining the finite-state machine as the decision-making core of the generated skill. Consequently, the implemented behavior remains structurally connected to the artifact from which it was produced. The same family of models can also be translated into plain SCXML for off-line analysis and reused to configure runtime monitors. The resulting workflow connects four activities that are often performed using unrelated artifacts:

1. the BT and skill models specify the intended deliberative behavior;
2. M2C generates executable skill code and the corresponding ROS 2 adapters;
3. AS2FM and SCAN support off-line statistical verification of properties on the modeled system;
4. MOON observes the deployed system and checks properties or conformance with the expected skill models at runtime.

Off-line verification and on-line monitoring provide complementary evidence. Off-line verification asks whether the modeled system satisfies a requirement under the semantics and timing assumptions represented in the model. It can explore behavior before deployment, but it cannot guarantee that code generation, middleware integration, or the physical platform will preserve every assumption. Runtime monitoring instead observes a concrete execution after deployment. It cannot explore executions that have not occurred, but it can reveal deviations caused by the reality gap between the analyzed model and the running implementation.

Two forms of runtime monitoring were considered. Property monitors check explicitly stated system requirements over observed communications, whereas model monitors check whether the sequence of events produced by a skill conforms to its behavioral model. The fault-injection experiments showed why both are useful: a property monitor detects a fault when it propagates to a selected requirement, while a model monitor can expose an internal behavioral deviation even when none of the monitored system properties becomes false. The experiments also showed that monitoring overhead is compatible with the deliberation-layer time scale considered in this thesis, although synchronous model monitoring is more expensive than asynchronous property observation.

For the museum application, the main result carried forward from Part I is therefore not an isolated code generator or monitor. It is an execution architecture in which a BT invokes plugins and modeled skills, those skills communicate with functional components through explicit ROS 2 interfaces, and the resulting interactions remain available for verification and

runtime observation. This architecture provides the concrete mechanism through which the abstract decisions of the planner are executed.

## 12.2 Recap of Flow-Aware Tour Planning

The second technical part considered a different problem: how a robot should select task-level navigation actions when traversal time depends on uncertain and changing pedestrian flow. A planner that reacts only to current occupancy may choose a route that appears clear at the decision instant but becomes congested before the robot reaches it. The proposed flow-aware planner instead incorporates predictions of future human motion into a stochastic tour-planning model.

The planning pipeline connects three abstraction levels. First, CLiFF-based human-motion prediction produces multiple possible future trajectories for each observed person. Second, the predicted positions are associated with regions around the edges of a topological map. The individual edge-presence probabilities are aggregated into distributions over congestion bands, yielding a compact representation of how crowded each edge may be at a future time. Third, these distributions are embedded in a stochastic shortest-path Markov Decision Process (MDP) whose state records the robot's topological position, task progress, and time-dependent congestion information.

The MDP actions are deliberately more abstract than the commands executed by the robot. An edge action represents a commitment to move from one topological region to another, while a wait action delays that commitment. The MDP does not generate velocity commands, perform collision avoidance, or compute a socially compliant geometric trajectory. Those responsibilities remain with the navigation stack. Instead, the planner decides which topological movement or POI choice is most promising when future traversal durations are uncertain.

The MDP is solved on-line using LRTDP in a receding-horizon cycle:

1. observe the people currently present in the environment;
2. predict their motion and estimate future congestion on the topological edges;
3. construct and solve the reachable flow-aware tour MDP;
4. execute only the first action of the resulting policy;
5. obtain new observations and repeat the process until the tour is complete.

Executing only the first action prevents the robot from committing to a long open-loop plan based on predictions that become less reliable with time. The bounded variant limits each planning step to three seconds and, in the reported experiments, retained most of the benefit of the converged solution.

The controlled evaluation on ATC maps compared the method with a Hamiltonian-path baseline that also replans, but uses current congestion rather than its predicted stochastic evolution. Across the tested map sizes and congestion representations, the flow-aware variants produced shorter tours; on the largest map with five congestion bands, the average reduction was approximately 17%. The result supports the claim that future flow provides useful task-level information beyond the benefit obtained from replanning alone. At the same time, the analysis identified the conditions under which this benefit can be expected: pedestrian flow must be sufficiently recurrent to predict, the topological graph must provide meaningful route alternatives, and the task order must be flexible enough for the planner to exploit them.

## 12.3 How the Two Parts Meet in the Museum

The previous parts operate at different but compatible levels of abstraction. The flow-aware planner reasons about topological locations, visited POIs, and uncertain traversal durations. The model-based execution framework reasons about BT nodes, plugins, skills, components, and ROS 2 communications. The museum application connects these levels through the POI scheduler. At each planning step, the flow-aware planner selects the next abstract navigation commitment. The scheduler converts this choice into a BT-level update that identifies the next POI. The `GoToPoi` sub-tree then invokes the relevant plugin and skill, and the functional navigation components realize the movement through the concrete navigation stack.

This connection also clarifies the distinct responsibilities of planning and monitoring. The planner adapts which action should be attempted in response to predicted crowd dynamics; it does not establish that every skill involved in executing that action behaves correctly. Verification and monitoring increase confidence in the specified behavior and expose deviations during execution; they do not decide which route is preferable under congestion. The two mechanisms are therefore complementary: planning improves task-level efficiency and execution smoothness, while verification and monitoring support correctness, conformance, and diagnosability.

The chapters that follow examine this integration in detail. Chapter 13 presents the museum application and its BT, skills, and functional components. The verification chapter

describes the properties checked off-line and monitored during execution. The planning chapter explains how the general flow-aware formulation is adapted to the museum's partially constrained order of rooms and POIs and evaluates its effect on tour duration, navigation recoveries, and aborts.

My contribution to this part includes implementing the use-case software, defining the properties used for verification and monitoring, and providing the robotics domain expertise needed to ground MOON in the tour-guide application. It also includes adapting and integrating the planner into the museum execution flow. Validation of the complete system produced domain feedback that informed the monitoring workflow and the shared tools, while demonstrating how the two principal technical contributions of the thesis can operate within one deployed architecture.



## Chapter 13

# Robotic Tour Guide Case Study Description

To perform the final evaluation of the thesis contributions, we consider a real robotic application: a robotic tour guide developed for a museum in Turin. This case study provides a realistic setting in which to assess the framework’s capabilities and limitations. Compared with the previous case studies, it involves a larger Behavior Tree (BT), a richer set of skills and components, and a broader collection of verification and monitoring properties, in addition to the flow-aware planning module. It therefore serves as a recap and integration point for the two previous parts: the model-based workflow from Part I provides the BT, skill, verification, and monitoring structure, while the planner from Part II provides adaptive route selection under visitor-flow uncertainty.

The application corresponds to UC3, the third use case of the CONVINCe project. My contribution to UC3 included implementing the use-case software, defining the verification and monitoring properties, and providing the robotics domain expertise needed to translate the tour-guide requirements into the robot’s behavior, skills, interfaces, and monitoring needs. This grounded MOON in a complete robotic application, while validation feedback from the use case contributed to design decisions for the monitoring workflow and the shared tools. The public repository<sup>1</sup> contains the executable implementation of the case study, including the high-level models, BT and navigation code, generated and handwritten skills, monitoring support, launch and Docker configurations, and the simulation infrastructure used to run the tour-guide system.

---

<sup>1</sup><https://github.com/convince-project/UC3>

The top-level BT of the application is shown in Figure 13.1. It coordinates the main tour phases, including checks on the visitors' state, navigation to the selected point of interest, interaction and narration actions, and the update of the next POI.

The main elements of this case study are as follows:

- A BT that executes a museum tour. The robot plans and selects a POI, navigates to it when visitors are following, and otherwise waits and prompts them to continue following. The BT also includes battery-management behavior, such as raising an alarm, returning to the charging station when necessary, and waiting until charging is complete. In addition, it handles verbal interaction and POI explanation. The top-level BT can be seen in Figure 13.1, while the sub-trees are shown in Figs. 13.2 to 13.9.
- 50 FSMs for the skill layer.
- 14 components for the functional layer.
- A set of properties verified on the model and monitored at runtime, listed in Table 14.1.

The case study is also used as a running example for connecting the two main technical parts of the thesis. At the planning level, selecting the next POI is an abstract decision over a topological graph. At the execution level, the selected POI becomes a concrete BT update: the scheduler sub-tree sets the destination, the `GoToPoi` sub-tree invokes the navigation skill, and the skill/component layer interacts with the ROS 2 navigation stack. The same action is therefore represented at different levels of abstraction, but the system keeps an explicit chain from planner output to executable BT leaves and monitored skill interfaces.

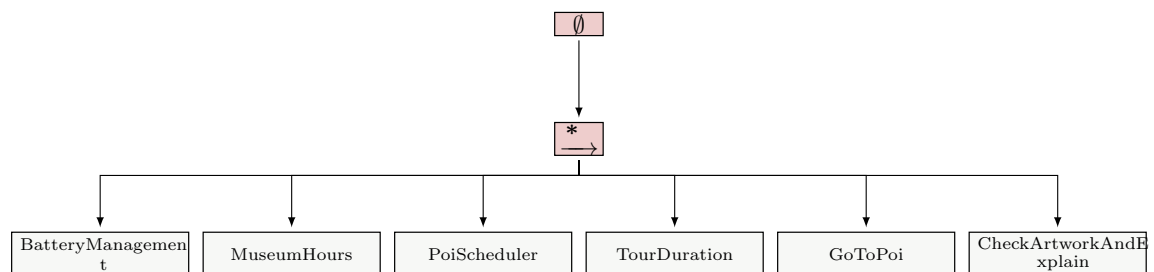


Figure 13.1 Schematic view of the top-level Behavior Tree of a museum tour-guide application. Sub-trees collapse lower-level BT fragments whose leaves are implemented by the skill layer. The BT coordinates the main tour phases, including checks on the visitors' state, navigation to the selected point of interest, interaction and narration actions, and the update of the next POI. The main BT includes the following sub-trees: battery management (Figure 13.2), museum hours and start position check (Figure 13.3), POI scheduler (Figure 13.4), executable scheduler (Figure 13.5), tour duration management (Figure 13.6), navigation to the next POI (Figure 13.7), people following check (Figure 13.8), and artwork explanation and question answering (Figure 13.9).

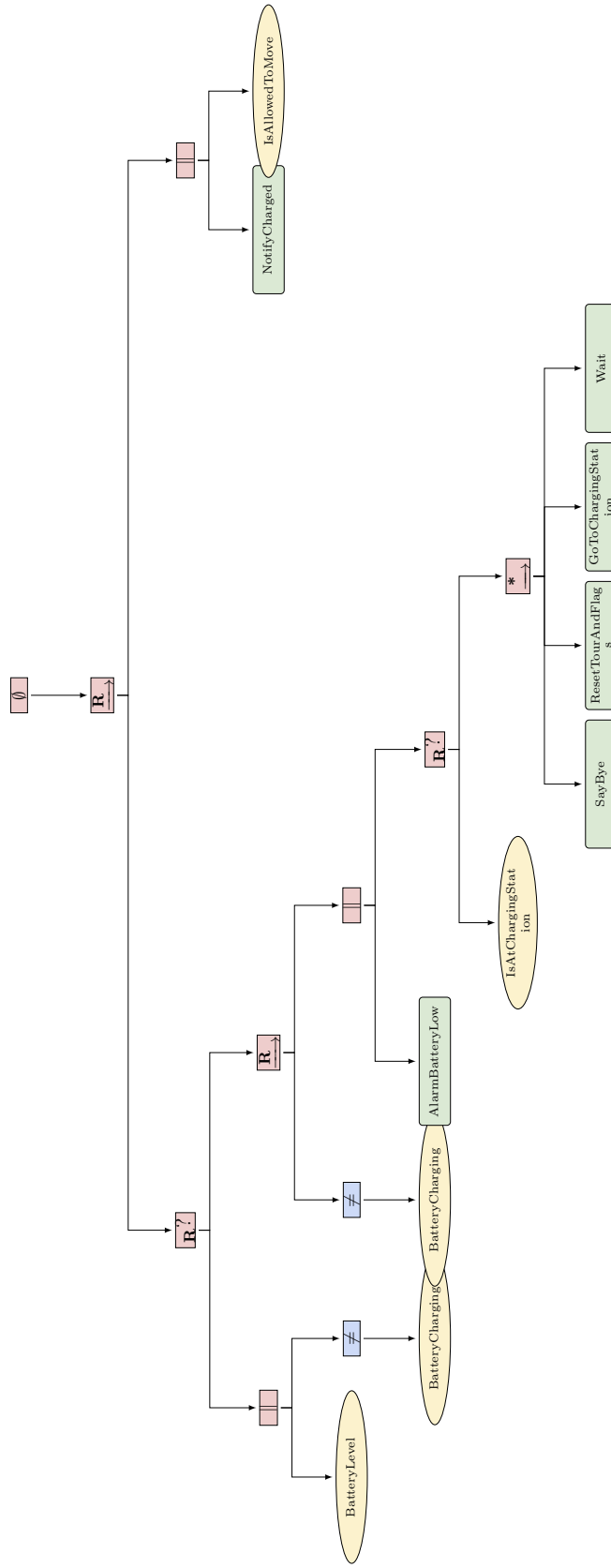


Figure 13.2 Schematic view of the sub-tree responsible for checking the battery level and managing the charging process. If the battery level is low, the robot will return to the charging station and wait until the battery is fully charged. This sub-tree is part of the main BT shown in Figure 13.1.

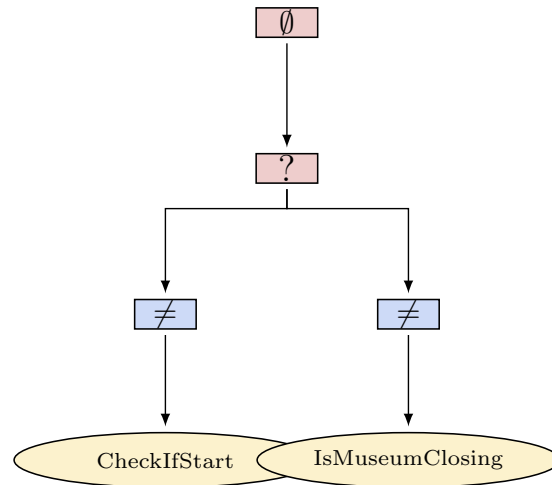


Figure 13.3 Schematic view of the sub-tree responsible for checking if the museum is open and the robot is at start position. This sub-tree is part of the main BT shown in Figure 13.1.

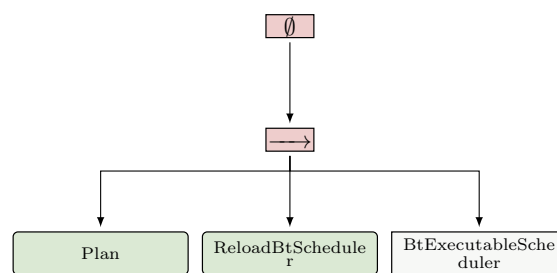


Figure 13.4 Schematic view of the sub-tree responsible for setting the next POI. It plans the next sequence of POIs based on the current occupancy, then writes the planned BT into a file, reloads the BT that sets the POI (Figure 13.5) and ticks it. This sub-tree is part of the main BT shown in Figure 13.1.

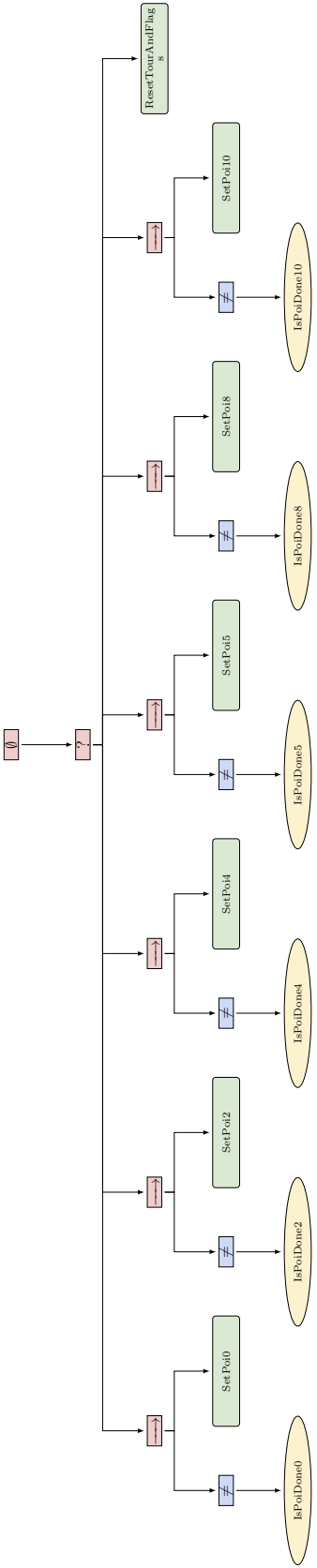


Figure 13.5 Schematic view of the sub-tree responsible for selecting the next POI to explain. This sub-tree is part of the main BT shown in Figure 13.1.

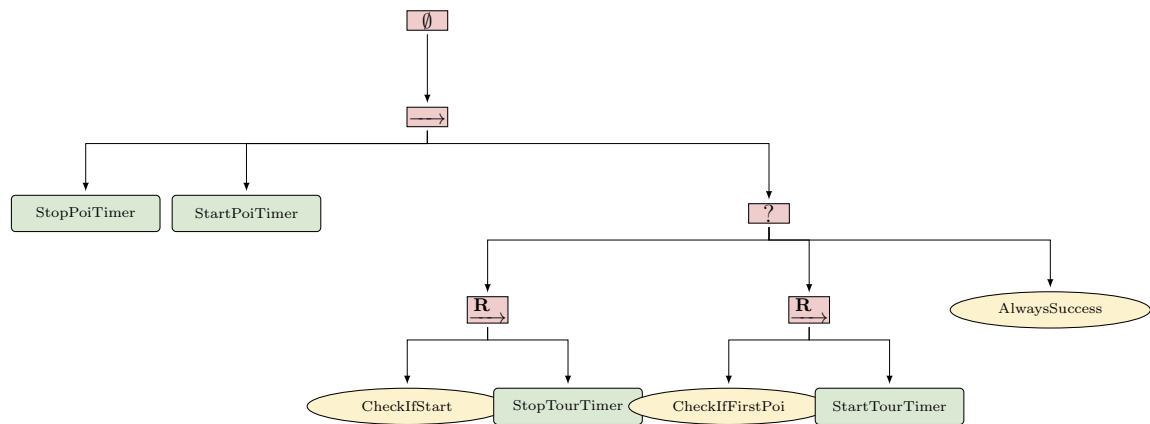


Figure 13.6 Schematic view of the sub-tree responsible for managing the tour duration. This sub-tree is part of the main BT shown in Figure 13.1.

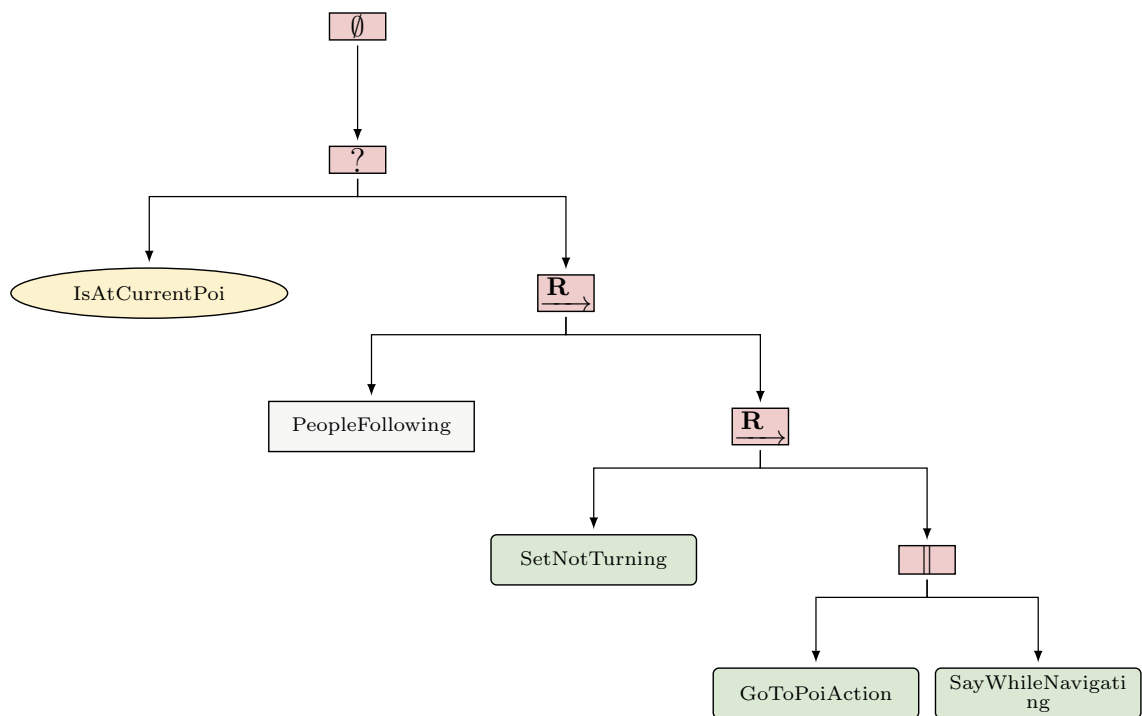


Figure 13.7 Schematic view of the sub-tree responsible for going to the next POI. The BT checks whether the robot is at the correct position, otherwise it moves the robot to the desired location. It also checks if people are following the robot by ticking the sub-bt in Figure 13.8. This sub-tree is part of the main BT shown in Figure 13.1.

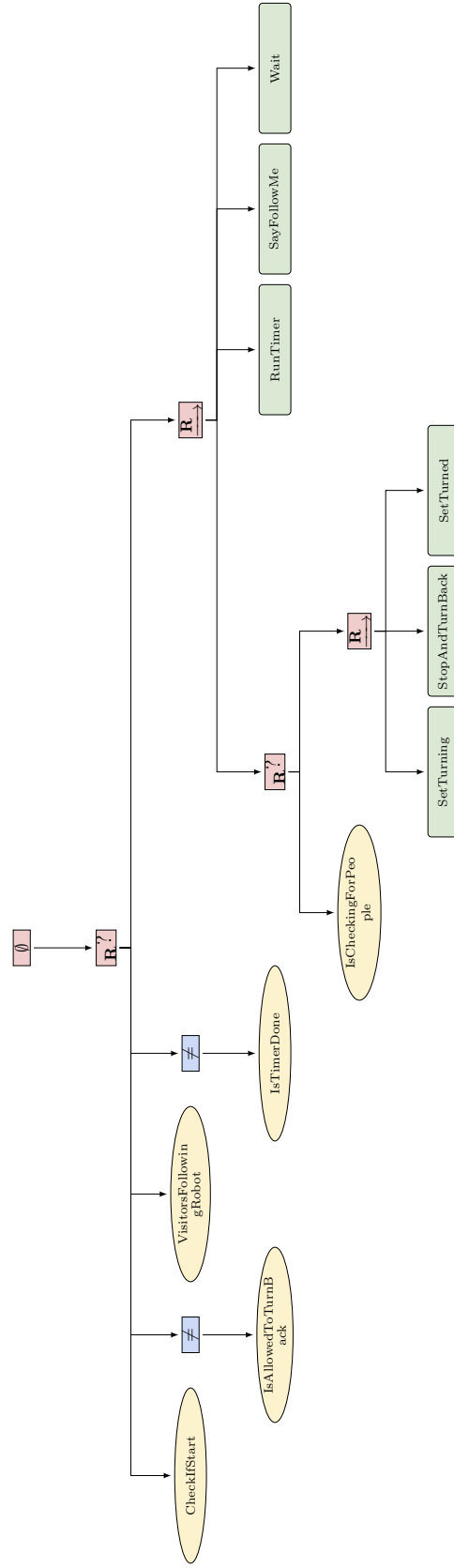


Figure 13.8 Schematic view of the sub-tree responsible for checking if people are following the robot. If not it stops and asks for following. This sub-tree is part of the main BT shown in Figure 13.1.



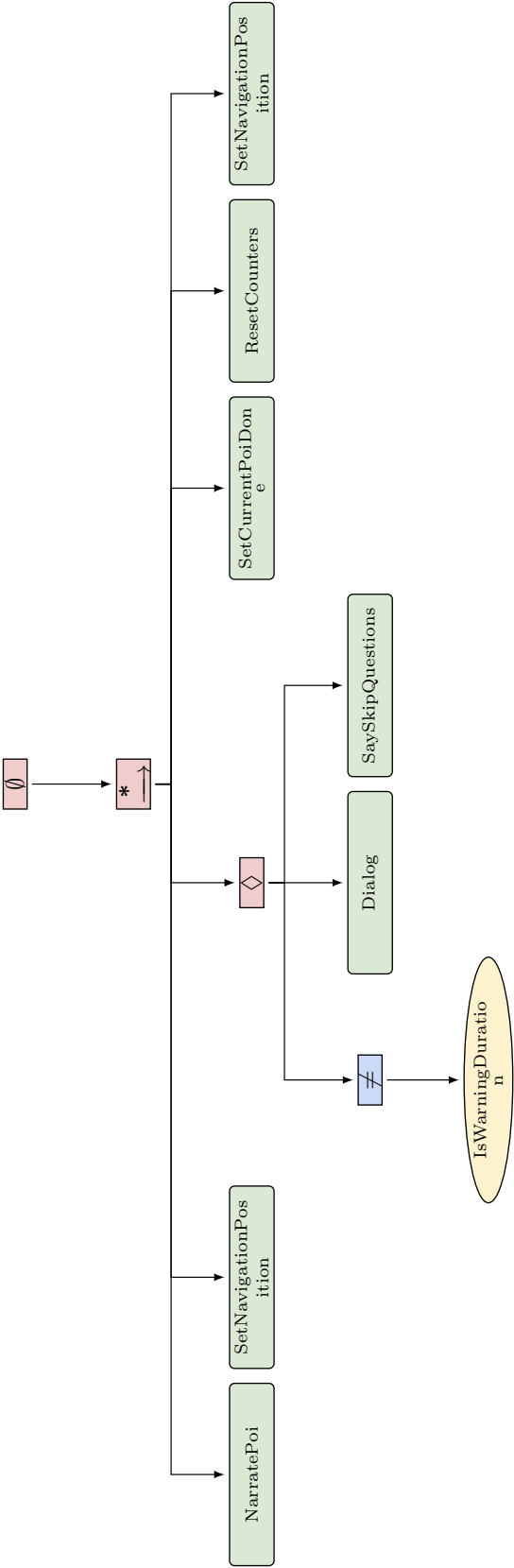


Figure 13.9 Schematic view of the sub-tree responsible for checking the artwork, explaining it and answering questions. If the robot notices that the tour is out of time, it skips the questions and proceeds. This sub-tree is part of the main BT shown in Figure 13.1.

# Chapter 14

## Verification

This chapter applies the verification and monitoring workflow introduced in Part I to the museum tour-guide application. The relevant idea from the earlier part is that system requirements can be expressed as temporal properties over observable ROS 2 events, checked off-line on the model when an abstract environment is available, and monitored on-line during execution. Here, the same principle is used at the scale of the full museum behavior: the properties are not intended to verify every internal detail of the robot, but to cover the main correctness, environment-interaction, and system-health conditions that affect a public tour.

To verify the system, we implemented a set of properties that can be both monitored at runtime and verified off-line on the model. All the properties are listed in Table 14.1, and they cover different aspects of the system, including the correct functioning of the BT, error conditions caused by unexpected situations in the robot's external environment, and error conditions caused by internal hardware faults or software-component failures. The categories of the properties are:

1. Properties that indicate a fault in the code or in the BT itself, which should never fail in a correct execution.
2. Properties that represent error conditions caused by unexpected situations in the robot's external environment, which hinder or delay the execution of BT actions.
3. Properties that represent error conditions caused by internal hardware faults or software-component failures, which support operator awareness by showing whether the system is fully operational and whether components such as sensors or communication links require intervention.

In particular, properties 1, 2, and 7 belong to the first category, properties 4, 10, 11, 13, 14, and 15 belong to the second category, and properties 3, 5, 6, 8, 9, and 12 belong to the third category. These properties show that verification is not an isolated activity but a structural component of the architecture at multiple levels: some properties validate the internal logic of the Behavior Tree and detect errors that should never occur in a correct execution; others capture degradations caused by the external environment, such as delays due to visitors not following the robot or failures to reach a target within a desired time window; and others focus on the health of hardware and software components, enabling early detection of sensor or communication faults.

Overall, the verification framework provides a systematic way to specify and check the desired properties of the system, both off-line and at runtime. It supports design-time analysis, runtime monitoring, and operator awareness, and provides the guarantees required to deploy more adaptive behaviors on the robot with greater confidence. This is especially important in the museum domain, where the robot must remain understandable and dependable while interacting with visitors in a public space.

Table 14.1 Monitoring properties considered in the case study.

| <b>Id</b> | <b>Short Name</b>             | <b>Description</b>                                                                                                                                                  | <b>Property</b>                                                                                                          |
|-----------|-------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------|
| 1         | noAlarmWith<br>HighBattery    | Each time the battery is charged, no alarm should be triggered.                                                                                                     | $\mathbf{H}(high\_battery \rightarrow \mathbf{H}(\neg alarm))$                                                           |
| 2         | alarmWith<br>LowBattery       | If there was an alarm, then there has been at least one previous low-battery event. Moreover, if the battery is low, then the alarm should be set within 5 seconds. | $\mathbf{H}\left((alarm \rightarrow \mathbf{P} low\_battery) \wedge \neg(\neg alarm \mathbf{S}[5:] low\_battery)\right)$ |
| 3         | isBattery<br>Published        | The battery topic should be published at least every 5 seconds.                                                                                                     | $\mathbf{P}[5:](t) \rightarrow \mathbf{P}[:5](battery\_published)$                                                       |
| 4         | isStartReached<br>InTimeAtEnd | If the tour is completed, the robot must arrive at the start point within a certain amount of time.                                                                 | $\mathbf{H}(\neg(\neg startReached \mathbf{S}[600:] tourFinished))$                                                      |

*Continued on next page*

Table 14.1 continued from previous page

| <b>Id</b> | <b>Short Name</b>              | <b>Description</b>                                                                                                                                        | <b>Property</b>                                                                                                                                                                                               |
|-----------|--------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 5         | isPeopleFollowing<br>Published | The <i>peopleFollowing</i> topic must be published at least every 7 seconds.                                                                              | $\mathbf{P}[7:] (t) \rightarrow \mathbf{P}[: 7] (\text{people\_following\_published})$                                                                                                                        |
| 6         | isCamera<br>Published          | The camera topic must be published at least every 5 seconds.                                                                                              | $\mathbf{P}[5:] (t) \rightarrow \mathbf{P}[: 5] (\text{camera\_published})$                                                                                                                                   |
| 7         | notCritical<br>Battery         | The battery level must never become critical.                                                                                                             | $\neg \text{critical\_battery}$                                                                                                                                                                               |
| 8         | notWheels<br>HWFault           | The wheel joints must not be in hardware fault.                                                                                                           | $\neg \text{wheel\_hardware\_fault}$                                                                                                                                                                          |
| 9         | notJoints<br>HWFault           | The robot joints must not be in hardware fault.                                                                                                           | $\neg \text{hardware\_fault}$                                                                                                                                                                                 |
| 10        | isPOIX<br>CompletedInTime      | If a POI is completed, then it must have been selected before. Moreover, if a POI is selected, then it must be completed within 10 minutes (600 seconds). | $\mathbf{H} \left( (\text{poiX\_completed} \rightarrow \mathbf{P} \text{ poiX\_selected}) \wedge \neg(\neg \text{poiX\_completed} \mathbf{S}[600:] \text{poiX\_selected}) \right),$<br>with $X = 1, \dots, 5$ |
| 11        | isWell<br>Localized            | The robot must be well localized, i.e. the covariance provided by AMCL must be below a threshold.                                                         | $\text{well\_localized}$                                                                                                                                                                                      |
| 12        | isMicrophone<br>PluggedIn      | If the microphone is recording, then it must not be unplugged.                                                                                            | $\text{is\_recording} \rightarrow \neg \text{is\_unplugged}$                                                                                                                                                  |
| 13        | isExternalPC<br>Connected      | The laptop used for launching the tour should be connected.                                                                                               | $\mathbf{P}[5:] (t) \rightarrow \mathbf{P}[: 3] (\text{connected\_to\_network})$                                                                                                                              |
| 14        | isConnected<br>ToWeb           | The laptop or R1 base should always be connected to the web.                                                                                              | $\mathbf{P}[5:] (t) \rightarrow \mathbf{P}[: 3] (\text{connected\_to\_web})$                                                                                                                                  |

Continued on next page

Table 14.1 continued from previous page

| Id | Short Name                               | Description                                                                                               | Property                                          |
|----|------------------------------------------|-----------------------------------------------------------------------------------------------------------|---------------------------------------------------|
| 15 | isArrivedAt<br>ChargingStation<br>InTime | When the battery is low, the robot should arrive at the charging station within a certain amount of time. | $\mathbf{H}(\neg(\neg arrivedAtCSS[600:] alarm))$ |

## 14.1 Off-Line Verification

The first step of the verification process was to validate the models and properties off-line, before deploying the system on the real robot. This was done using the design-time verification tools described in Section 4.5.2. Different versions of the BT were tested during development. To support this verification, the environment was modelled as a set of interconnected state machines in RoAML format. We tested two properties related to the behavior of the BT, (properties 1 and 4) two properties related to internal hardware faults or software-component failures (properties 2 and 3) and three properties related to error conditions caused by unexpected situations in the robot's external environment. (properties 5, 6, and 7).

To validate the model, SCAN was executed iteratively on the system and the results were analysed afterwards. After the first run, almost all the properties failed. This showed that some components had not yet been modelled correctly, so the models were refined until the sources of failure became clear. One of the problematic components, for example, was the navigation component responsible for simulating movement between POIs.

Additional detected problems were caused by faulty skills. In particular, their misbehavior during the BT halt command caused them not to forward the halt request to the ROS 2 actions they were executing.

The BT failed in different ways. The first failure was due to a condition node being connected where an action node was actually required. This caused the BT to get stuck on ticking the condition node, since the service associated with the condition node was never called, and therefore the condition was never satisfied. This prevented the BT from progressing and caused it to fail to complete the tour. This issue was detected by AS2FM, which reported a type mismatch caused by incompatible service types. The second involved a wrong parameter of a parallel node: the parallel node (as seen in Figure 14.1) was configured to succeed when 3 children succeeded, but it had only 2 children. This caused the BT to get

stuck in that node, which led to a failure of the tour. This issue was detected by SCAN as a failure of properties 1 and 4. The issue was discovered by looking at the battery trace, which with the wrong BT goes to zero without recharging (Figure 14.2).

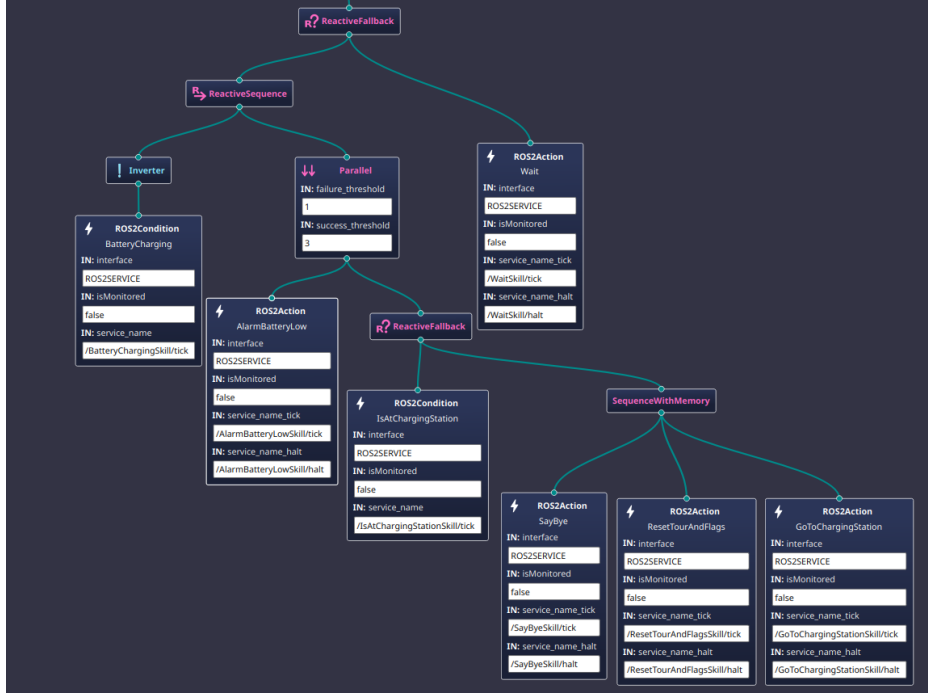


Figure 14.1 The wrong node inside the BT. It can be seen the parallel node with a parameter of success of 3 but instead it has only two children, which caused the BT to get stuck.

To confirm that the final system behaved correctly, and that the properties were not being falsified by modeling errors, we inspected variable traces generated by SCAN over time. These traces showed that the BT selected all POIs and returned to the starting point on each run (Figure 14.4), while the battery level evolved consistently with the expected execution (Figure 14.3).

## 14.2 On-line Verification

For this use case, we focused on monitoring the properties at runtime, while we did not implement model monitoring. This was a deliberate design choice: given the number of active models, running model monitoring as well would have made the system too heavy. Since the models had already been validated off-line and M2C had been used for translation, the runtime focus was placed on property monitoring, which is more directly relevant to the operator and to the reliability of the deployed system.

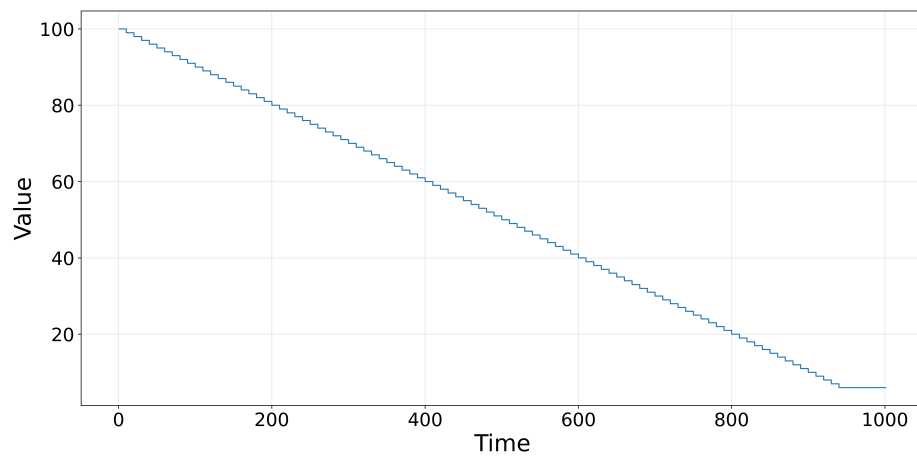


Figure 14.2 Traces of the battery level variable during wrong SCAN execution. It can be seen that the battery level goes to zero.

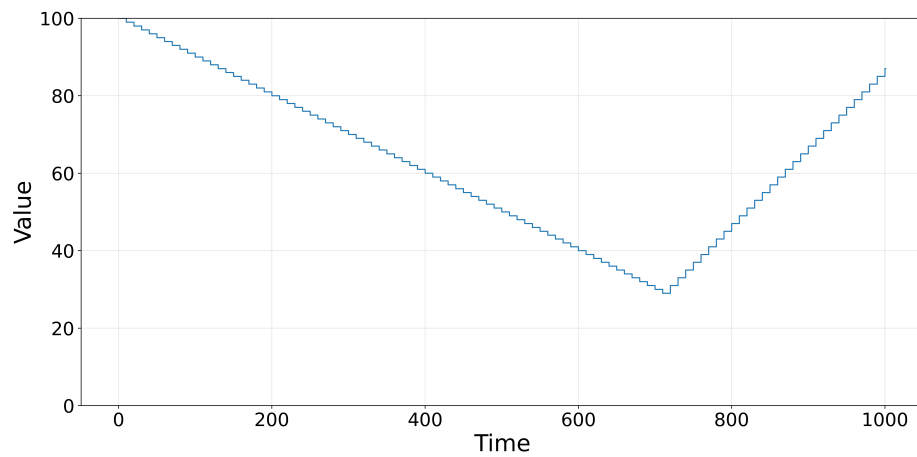


Figure 14.3 Traces of the battery level variable during SCAN execution.

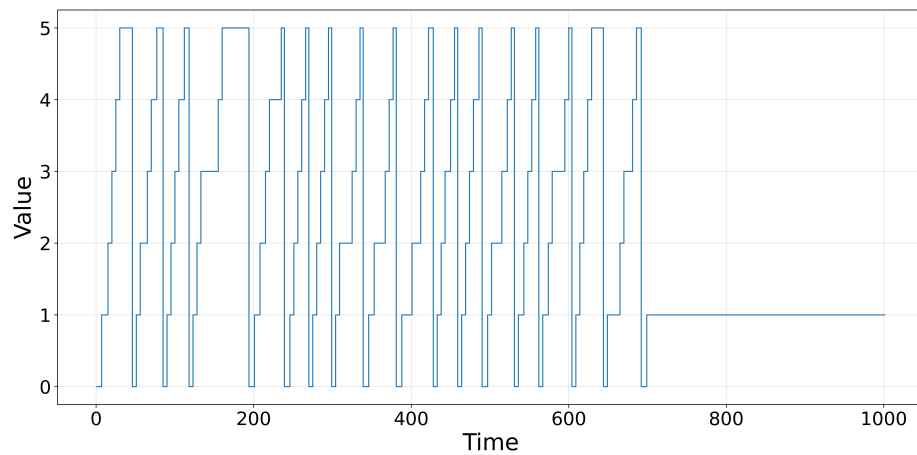


Figure 14.4 Traces of the POI selection and completion variables during SCAN execution.

To monitor the properties at runtime, we relied either on ROS 2 service introspection or on ROS 2 topic subscription, which avoids introducing man-in-the-middle code into the implementation. This enables non-intrusive monitoring of the deployed system without modifying its code, thereby reducing the risk of introducing additional faults.

To validate the monitors, we performed tests in simulation and on the real robot, where we manually triggered some of the monitored conditions, such as unplugging the microphone or simulating a low battery by reducing the battery level in the model. These tests confirmed that the monitors were correctly implemented and that they provided accurate information about the system's state during execution, since the expected properties were violated when the corresponding conditions were triggered, while no properties were violated during normal execution. In addition, we performed a long-term test in which the robot executed multiple tours in a row, while we monitored the properties and checked that they were not violated.

The results of this test can be seen in Table 14.2, which reports 30 executed tours, their outcomes, and the number of violations for each property. These tests confirmed that the monitors were correctly implemented and that they provided accurate information about the system's state during execution.



| #<br>Tour | Tour<br>successful                                      | Manual<br>interventions | Critical<br>failure | Navigation<br>conflicts<br>(replanning/aborted) | Duration                 | speech<br>comprehension<br>(errors / questions) | Property<br>violations<br>(det/tot) | Property<br>ids | Event<br>Tour<br>too long<br>(det/tot) | Event<br>Battery<br>low<br>(det/tot) | Event<br>network<br>down<br>(det/tot) | # Persons |
|-----------|---------------------------------------------------------|-------------------------|---------------------|-------------------------------------------------|--------------------------|-------------------------------------------------|-------------------------------------|-----------------|----------------------------------------|--------------------------------------|---------------------------------------|-----------|
| 1         | a. all POIs done                                        | 1                       | No                  | 1 / 0                                           | 28m                      | 7 / 13                                          | 0/0                                 | -               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 7         |
| 2         | a. all POIs done                                        | 0                       | No                  | 0 / 0                                           | 34m                      | 5 / 11                                          | 0/0                                 | -               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 4         |
| 3         | a. all POIs done                                        | 1                       | No                  | 0 / 0                                           | 28m                      | 3 / 9                                           | 5/5                                 | 6,8,9,13,14     | 0 / 0                                  | 0 / 0                                | 1 / 1                                 | 6         |
| 4         | a. all POIs done                                        | 0                       | No                  | 0 / 0                                           | 29m                      | 6 / 12                                          | 2/2                                 | 13,14           | 0 / 0                                  | 0 / 0                                | 1 / 1                                 | 2         |
| 5         | a. all POIs done                                        | 0                       | No                  | 0 / 0                                           | 28m                      | 3 / 9                                           | 2/2                                 | 6,9             | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 5         |
| 6         | -                                                       | 0                       | Yes                 | 0 / 0                                           |                          | 0 / 4                                           | 0/0                                 | -               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 4         |
| 7         | a. all POIs done                                        | 0                       | No                  | 0 / 0                                           | 31m                      | 4 / 10                                          | 1/1                                 | 4               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 6         |
| 8         | e. skipped questions                                    | 1                       | No                  | 1 / 0                                           | 34m                      | 5 / 10                                          | 0/0                                 | -               | 1 / 1                                  | 0 / 0                                | 0 / 0                                 | 4         |
| 9         | -                                                       | 0                       | Yes                 | 1 / 0                                           |                          | 0 / 3                                           | 0/0                                 | -               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 8         |
| 10        | a. all POIs done                                        | 0                       | No                  | 0 / 0                                           | 28m                      | 3 / 9                                           | 0/0                                 | -               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 14        |
| 11        | a. all POIs done                                        | 1                       | No                  | 1 / 0                                           | 26m                      | 0 / 6                                           | 1/1                                 | 11              | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 8         |
| 12        | a. all POIs done                                        | 0                       | No                  | 0 / 0                                           | 24m                      | 1 / 7                                           | 1/1                                 | -               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 12        |
| 13        | d. battery low                                          | 0                       | No                  | 0 / 0                                           | 24m                      | 2 / 7                                           | 0/0                                 | -               | 0 / 0                                  | 1 / 1                                | 0 / 0                                 | 6         |
| 14        | a. all POIs done                                        | 0                       | No                  | 0 / 0                                           | 30m                      | 4 / 10                                          | 2/2                                 | 6,9             | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 31        |
| 15        | a. all POIs done                                        | 1                       | No                  | 0 / 0                                           | 40m                      | 4 / 10                                          | 1/1                                 | 9               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 14        |
| 16        | a. all POIs done                                        | 0                       | No                  | 0 / 0                                           | 34m                      | 0 / 6                                           | 2/2                                 | 6,11            | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 8         |
| 17        | a. all POIs done                                        | 1                       | No                  | 0 / 0                                           | 31m                      | 1 / 7                                           | 1/1                                 | 11              | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 8         |
| 18        | -                                                       | 0                       | Yes                 | 0 / 0                                           |                          | 0 / 3                                           | 0/0                                 | -               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 2         |
| 19        | a. all POIs done                                        | 0                       | No                  | 0 / 0                                           | 24m                      | 2 / 8                                           | 0/0                                 | -               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 6         |
| 20        | -                                                       | 0                       | Yes                 | 0 / 0                                           |                          | 1 / 3                                           | 1/1                                 | 9               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 4         |
| 21        | a. all POIs done                                        | 0                       | No                  | 0 / 0                                           | 23m                      | 2 / 8                                           | 3/3                                 | 6,9,11          | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 4         |
| 22        | a. all POIs done                                        | 0                       | No                  | 0 / 0                                           | 20m                      | 3 / 9                                           | 1/1                                 | 6               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 6         |
| 23        | b. stopped by user                                      | 0                       | No                  | 0 / 0                                           | 11m                      | 2 / 6                                           | 0/0                                 | -               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 4         |
| 24        | a. all POIs done                                        | 0                       | No                  | 0 / 0                                           | 22m                      | 3 / 9                                           | 0/0                                 | -               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 3         |
| 25        | a. all POIs done                                        | 0                       | No                  | 0 / 0                                           | 26m                      | 1 / 7                                           | 0/0                                 | -               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 6         |
| 26        | a. all POIs done                                        | 0                       | No                  | 0 / 0                                           | 24m                      | 2 / 8                                           | 1/1                                 | 9               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 6         |
| 27        | a. all POIs done                                        | 0                       | No                  | 0 / 0                                           | 24m                      | 2 / 8                                           | 0/0                                 | -               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 4         |
| 28        | a. all POIs done                                        | 0                       | No                  | 0 / 0                                           | 25m                      | 1 / 7                                           | 0/0                                 | -               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 4         |
| 29        | a. all POIs done                                        | 0                       | No                  | 0 / 0                                           | 24m                      | 0 / 6                                           | 0/0                                 | -               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 4         |
| 30        | a. all POIs done                                        | 0                       | No                  | 0 / 0                                           | 22m                      | 0 / 6                                           | 0/0                                 | -               | 0 / 0                                  | 0 / 0                                | 0 / 0                                 | 6         |
|           | 26 tours successful<br>(20 without any<br>intervention) | 6                       | 4                   | 4 / 0                                           | mean: 26.8m<br>tot: 694m | 67 / 232                                        | 24 / 24                             | -               | 1 / 1                                  | 1 / 1                                | 2 / 2                                 | 206       |

Table 14.2 Results from the tours in Palazzo Madama.

# Chapter 15

## Using Flow-Aware Planning in a Real-World Museum Scenario

### 15.1 Changes for Museum Tour Guide Case Study

This chapter applies the flow-aware planning framework introduced in Part II to the museum tour-guide system. The original planner treats the tour as a flexible task-level routing problem over a topological map, where edge costs depend on predicted pedestrian congestion. The museum application adds two practical constraints: the guide must respect the narrative order of the rooms, and each artwork may be reachable from more than one explanation position. The following sections therefore describe how the abstract MDP formulation is adapted before being connected to the BT-based execution architecture introduced in Part I.

To integrate the congestion-aware planner into the museum tour-guide case study, several modifications were introduced to the original system architecture and execution flow. In the museum scenario, the rooms must be presented in a predefined order, so the planner described in Part II was not sufficient on its own: it was designed to compute an efficient route over a set of goals without enforcing ordering constraints.

The planner was adapted to meet the specific requirements of the museum tour while preserving its core functionality: using the CLiFF map to estimate future congestion and select the least congested route. In particular, the planner was extended to support multiple vertices associated with the same POI. This allows it to consider different possible entry points or observation positions for the same exhibit, enabling the robot to explain a POI from the least congested direction. This is especially useful when visitors obstruct the robot while it is attempting to reach a specific exhibit.

The planner also needed to distinguish between ordinary topological vertices, vertices from which the robot can explain a POI, and final vertices from which the tour can be completed. For this reason, the topological map was extended with additional vertex attributes. The robot does not provide explanations at every vertex in the map, but only at a subset corresponding to the POIs included in the tour, as shown in Figure 15.4. Each vertex is annotated with a boolean value indicating whether it corresponds to a POI, an integer representing the POI index, and a boolean flag indicating whether it is a final goal vertex. These annotations are used both to define the goal states and to compute the heuristic adopted by LRTDP.

Finally, the planner is executed whenever the robot completes the explanation of a POI and is ready to move toward the next area. After computing the new route, it produces an updated sequence of remaining POIs and generates a new .xml file for the navigation sub-tree. The main Behavior Tree then reloads this navigation sub-tree and executes it.

This design goes beyond simply reordering a vector of POI indices. By modifying the Behavior Tree structure at runtime, the system can support a richer and more flexible form of adaptation, and it provides a basis for more complex adaptive interactions in future work.

Subsequent sections describe the details of the modified planner, the data collection and processing steps, the integration with the Behavior Tree, and the evaluation results obtained in the simulated museum environment.

### 15.1.1 Main Changes

First, to define a correct MDP for the museum scenario, we need to modify the definition of the vertices in the topological map. In the original formulation, vertices were simply defined as unique identifiers corresponding to specific locations in the environment. However, in the museum scenario, we need to distinguish between different types of vertices based on their role in the tour.

The vertices are then defined as:

**Definition 11** A vertex  $v_M \in \mathcal{V}_M$  is defined as a tuple  $\langle id, isPOI, POIIndex, isFinal \rangle$ , where  $id$  is a unique identifier for the vertex,  $isPOI$  is a boolean indicating whether the vertex corresponds to a point of interest (POI) in the museum,  $POIIndex$  is an integer indicating the index of the POI if the vertex is a POI, or -1 otherwise, and  $isFinal$  is a boolean indicating whether the vertex corresponds to the final vertex of the tour. The set of POI indices is denoted by  $\mathcal{P}_M$  and the number of POIs in the tour is denoted by  $N_{POI} = |\mathcal{P}_M|$ .

With these changes, the resulting MDP is also redefined as follows:

**Definition 12** Given topological map  $\mathcal{T}$  with vertices defined as in Definition 11 and time bound  $D \in \mathbb{R}_{\geq 0}$ , the museum MDP is a tuple  $\mathcal{M}_M = \langle S_M, \bar{s}_M, A_M, T_M, C_M, G_M \rangle$ , where:

- $S_M = \mathcal{V}_M \times \mathbb{R}_{\geq 0} \times 2^{\mathcal{P}_M}$ . States  $s = (v, t, \omega)$  describe the robot's current vertex, the current time, and the set  $\omega$  of POIs already explained on the tour.
- $\bar{s}_M = (\bar{v}, \bar{t}, \emptyset)$  for the first decision step, where  $\bar{v}$  and  $\bar{t}$  are the starting location and time for the tour, respectively. In subsequent decision steps,  $\bar{s}_M$  is set to the current state.
- $A_M = \mathcal{E} \cup \{\text{explain}\} \cup \{\text{end}\}$ . At each state the robot can either navigate along edges that are outgoing from its current vertex, explain the current POI if it is at a POI vertex that has not been explained yet, or end the tour if all POIs have been explained and the robot is at a final vertex.
- The transition function  $T_M$  captures the effect of robot navigation under congestion, explaining a POI, and ending the tour. Formally, for states  $s = (v, t, \omega)$ ,  $s' = (v', t', \omega')$ , and action  $e$ :

$$T_M(s, e, s') = \begin{cases} \Pr(c^j \mid e@t) & e = (v, v'), \\ & t < D, \\ & t' = t + \mathbb{E}[\rho(e, c^j)] \text{ and} \\ & \omega' = \omega \\ 1 & e = \text{explain}, \\ & v \text{ is a POI vertex with index } p(v), \\ & p(v) \notin \omega, \\ & \omega' = \omega \cup \{p(v)\} \\ 1 & e = \text{end and } s \in G_M \\ 0 & \text{otherwise.} \end{cases} \quad (15.1)$$

- The cost function  $C_M$  captures the expected navigation duration over all congestion bands. Formally, for state  $s = (v, t, \omega)$  and action  $e$ :

$$C_M(s, e) = \begin{cases} \sum_j \Pr(c^j \mid e@t) \cdot \mathbb{E}[\rho(e, c^j)] & e \in \mathcal{E} \\ d_{\text{explain}} & e = \text{explain} \\ d_{\text{end}} & e = \text{end} \end{cases} \quad (15.2)$$

- $G_M = \{(v, t, \omega) \in S_M \mid v \in F_M, \mathcal{P}_M \subseteq \omega, t < D\}$ , where  $F_M = \{v \in \mathcal{V}_M \mid \text{isFinal}(v)\}$  is the set of final vertices. Thus, the robot reaches the goal if it is at a final vertex, it has explained all POIs, and it remains within the time bound  $D$ .

Given a state  $s = (v, t, \omega)$ , the heuristic used in the museum experiments combines three terms: the shortest-path distance from the current vertex to any final vertex, the explanation time for the POIs that remain to be explained, and a large penalty for reaching a POI while earlier POIs in the tour order are still unexplained. Let

$$d_F(v) = \min_{g \in F_M} d_{\text{sp}}(v, g)$$

be the congestion-free shortest-path distance from  $v$  to any final vertex, and let

$$r(s) = \max(0, |\mathcal{P}_M| - |\omega|)$$

be the number of POIs that still need to be explained. If  $p(v)$  denotes the POI index of vertex  $v$ , with  $p(v) = \perp$  when  $v$  is not a POI vertex, the ordering penalty is

$$\eta(s) = \begin{cases} \lambda \cdot |\{i \in \mathcal{P}_M \mid i < p(v) \wedge i \notin \omega\}| & p(v) \neq \perp, \\ 0 & p(v) = \perp, \end{cases}$$

where  $\lambda$  is a large constant. The heuristic is then

$$h_M(s) = \begin{cases} 0 & s \in G_M, \\ M & F_M = \emptyset \text{ or } d_F(v) \text{ is undefined,} \\ d_F(v) + r(s) d_{\text{explain}} + \eta(s) & \text{otherwise,} \end{cases} \quad (15.3)$$

where  $M$  is a large finite value used for unreachable states. In the implementation,  $\lambda = 99999$  and  $M = 9999999$ .



Figure 15.1 Left: a LiDAR device that was placed inside the museum to track the visitors. Right: one of the devices placed inside the museum.

The other components of the planner, including the CLiFF-based congestion prediction and the LRTDP algorithm, remain unchanged from the original formulation. The main difference is that the planner now operates on a different state space that includes the set of explained POIs, and it must consider additional constraints related to the tour order and the final goal condition. However, the underlying principles of flow-aware planning and long-horizon reasoning still apply in this modified setting.

## 15.2 Data Collection and Processing

To be able to generate a CLiFF map and to test the planner, we needed to collect data on visitor flow in the museum. This was done by deploying eight custom LiDAR devices on the floor (Figure 15.1) during the museum’s opening hours for a period of a month. All the data was then processed to extract the trajectories of visitors moving through the museum, which were then used to build the CLiFF map and to test the planner in a simulated environment. The trajectories were extracted from the raw LiDAR scans through a multi-step process. At first, the static background was removed, using a distance transform to compensate for small placement inaccuracies. Subsequently, a clustering filter was used to detect moving obstacles corresponding to people and to reconstruct their trajectories over time (Figure 15.2). The data can be found on Zenodo (Rosa et al. [2026]).

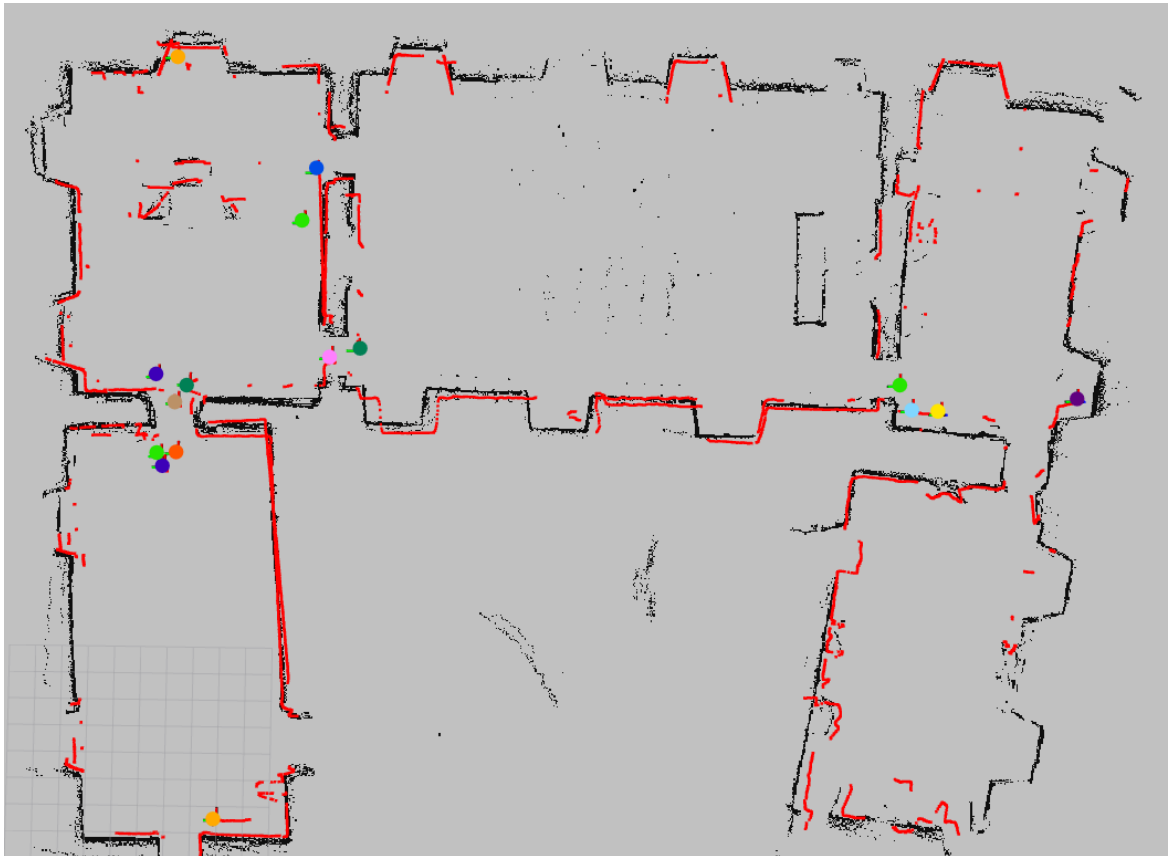


Figure 15.2 The predefined museum map with the scans of the eight LiDAR devices placed in the rooms overlaid, together with the detected people (shown as colored dots). Small mismatches between scans from devices located in the same room are due to manual placement inaccuracies.

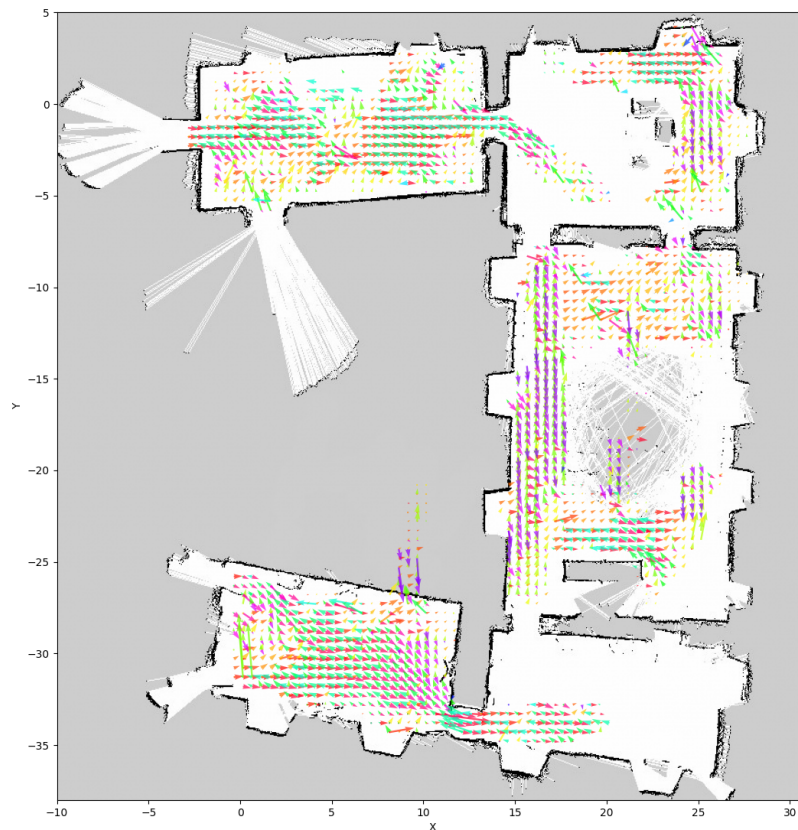


Figure 15.3 The CLiFF map showing the visitors' flow across the rooms of the museum.

## 15.3 Integration with the Behavior Tree

An important aspect of this case study is that the planner is not used as a standalone module, but is embedded into the full tour-guide architecture. The Behavior Tree remains responsible for the high-level execution flow of the tour, including interaction with visitors, battery management, and the sequencing of navigation and explanation actions. The role of the congestion-aware planner is to update the order in which the remaining POIs should be visited, based on the current estimate of visitor flow in the museum.

This integration is particularly relevant from a thesis perspective because it shows that the planner can be inserted into a richer deliberative architecture without replacing the existing control structure. Instead, the planner augments the BT with adaptive decision-making capabilities, while the tree preserves the reactivity and modularity required by the real application.

The interface between the planner and the BT is intentionally narrow. The planner does not call navigation actions directly, and it does not replace the skill layer. Its output is an



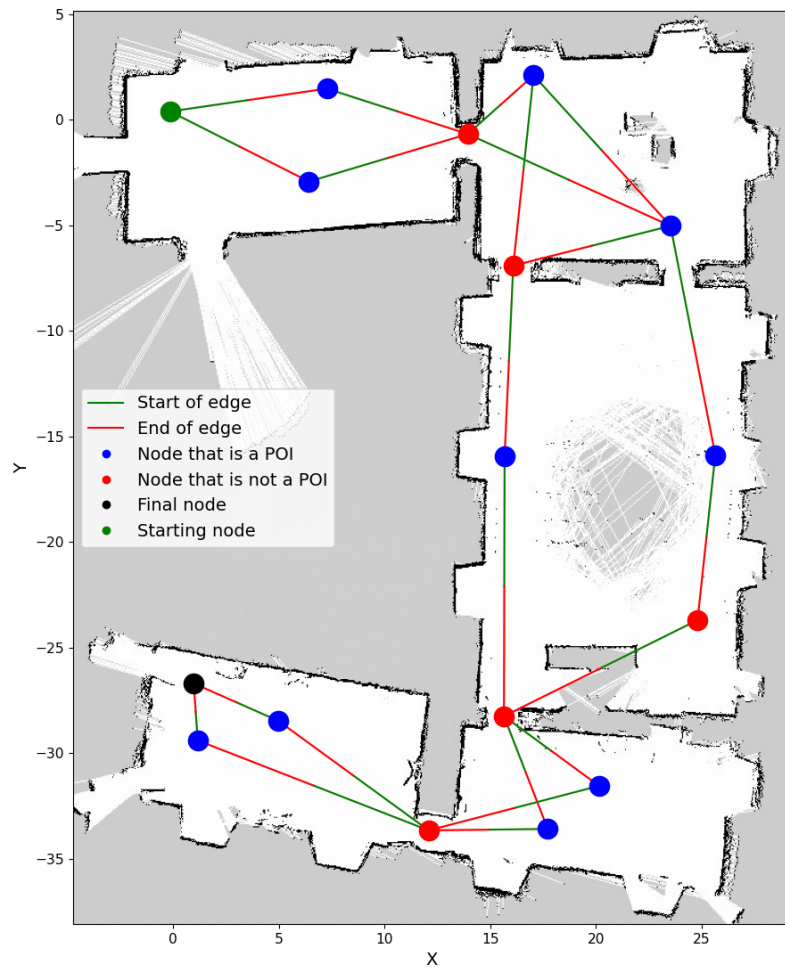


Figure 15.4 The graph of the possible paths connecting the tour POIs and their alternative positions.

ordered set of abstract task decisions: which POI should be selected next, which alternative POI vertex should be used when several explanation positions are available, and when the tour can move toward a final vertex. These decisions are converted into a scheduler sub-tree containing concrete BT leaves, such as the `SetPoi` actions shown in Figure 13.5. Once the scheduler has updated the selected destination, the existing `GoToPoi` sub-tree is responsible for executing the concrete navigation behavior through the plugin, skill, and component layers. This preserves the abstraction boundary used throughout the thesis: the MDP reasons over topological actions and expected congestion, while the BT-based architecture realizes those decisions as executable and monitorable ROS 2 behavior.

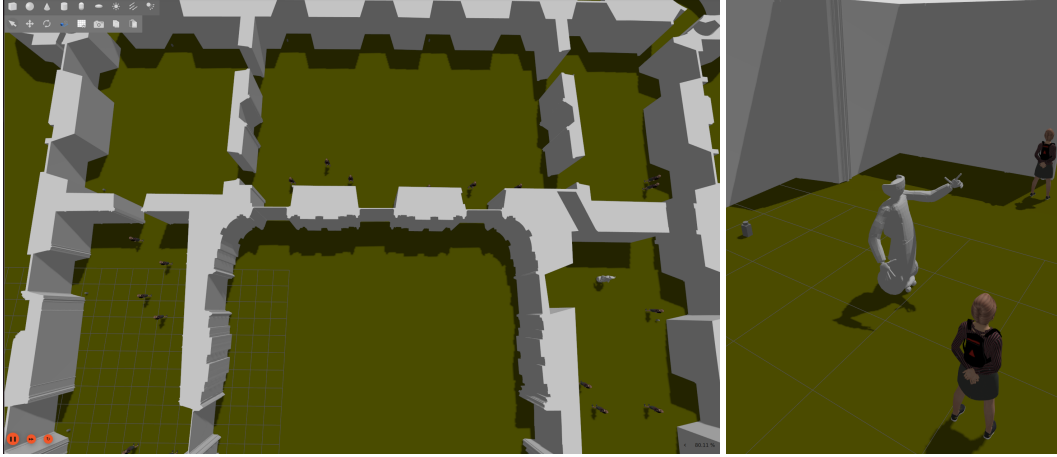


Figure 15.5 Screenshots of the museum environment simulated in Gazebo.

## 15.4 Planner Execution Cycle

During execution, the planner is triggered when the robot completes the explanation of a POI and is ready to move to the next area of the museum. At that point, the current human observations are used to estimate future congestion, and the planner computes a new order for the still-unvisited POIs. The result is then translated into a new scheduler sub-tree, encoded as an XML file, which is loaded and ticked before the next navigation phase begins.

This execution cycle implements a practical form of receding-horizon planning in the real system. Rather than generating a fixed tour once and for all, the robot repeatedly updates the remaining plan at meaningful task-level boundaries. This keeps the computational overhead manageable while still allowing the tour to adapt to changes in visitor distribution over time.

## 15.5 Evaluation

To assess the impact of the congestion-aware planner, we analysed the robot's performance across 60 full museum tours: 30 with the planner enabled (*w/ CPV2*) and 30 with the planner disabled (*w/o CPV2*). Evaluation was carried out in a simulated environment replicating the museum and executing the same code used on the real system, with visitors simulated according to trajectories extracted from the collected dataset (Figure 15.5).

The evaluation metrics gathered were total tour duration, the number of navigation recoveries, and the number of navigation aborts. The statistics in Table 15.1 show an advantage for CPV2 on all three metrics. The difference is particularly noticeable in the number of recoveries and aborts, resulting in smoother trajectories with fewer interruptions

and fewer visually awkward stops to recompute paths around visitors. In relative terms, CPV2 reduces the average tour duration by about 13%, the average number of recoveries by about 82%, and the average number of aborts by about 77%. This indicates that the practical benefit of the planner is larger on execution stability than on raw completion time. For these metrics, not only does the average value decrease significantly, but the distribution of events also becomes more concentrated (Figure 15.6). Overall, the navigation system becomes substantially more stable.

Additional insights can be gained by inspecting the statistics on a POI-by-POI basis. Figure 15.7 shows the time taken to complete each POI under the two conditions. CPV2 does not uniformly improve every segment of the route. The main benefit is concentrated at POI 3, which appears to be the most affected by human congestion due to its configuration. In other POIs, the differences are smaller, and in some cases there is even a slight increase in traversal time when CPV2 is active. This can be explained by the fact that, given the specific graph geometry, the time gained by choosing a less congested path may be offset by the longer distance required to reach the goal through that route, and the fact that the prediction of congestion levels may not be entirely accurate.

One final consideration concerns POIs 4 and 5, where the proposed approach appears to be less effective (Figure 15.7). In this part of the tour, the planner makes the same choice under both experimental conditions. This can be explained by the relatively small size of the room and by the fact that the paths leading to the two POIs partially overlap. Consequently, the lack of improvement should not be interpreted as a failure of the mechanism itself, but rather as the result of ineffective replanning opportunities in that part of the route.

To assess the statistical value of the results, Mann-Whitney U tests were also performed for each metric, comparing the distributions of the two conditions across the 30 tours. Results, shown in Table 15.2, confirm that the differences are statistically significant for all three metrics.

| Metric                 | w/ CPV2 | w/o CPV2 |
|------------------------|---------|----------|
| Average total time (s) | 405.43  | 466.16   |
| Average recoveries     | 13.40   | 74.23    |
| Average aborts         | 0.77    | 3.361    |

Table 15.1 Statistics on 30 simulated tours with and without congestion replanning

| Metric     | p-value          |
|------------|------------------|
| Total time | $1.5 * 10^{-3}$  |
| Recoveries | $2.76 * 10^{-8}$ |
| Aborts     | $3.04 * 10^{-9}$ |

Table 15.2 Mann-Whitney U test results comparing the distributions of total time, recoveries and aborts across the 30 tours with and without CPV2.

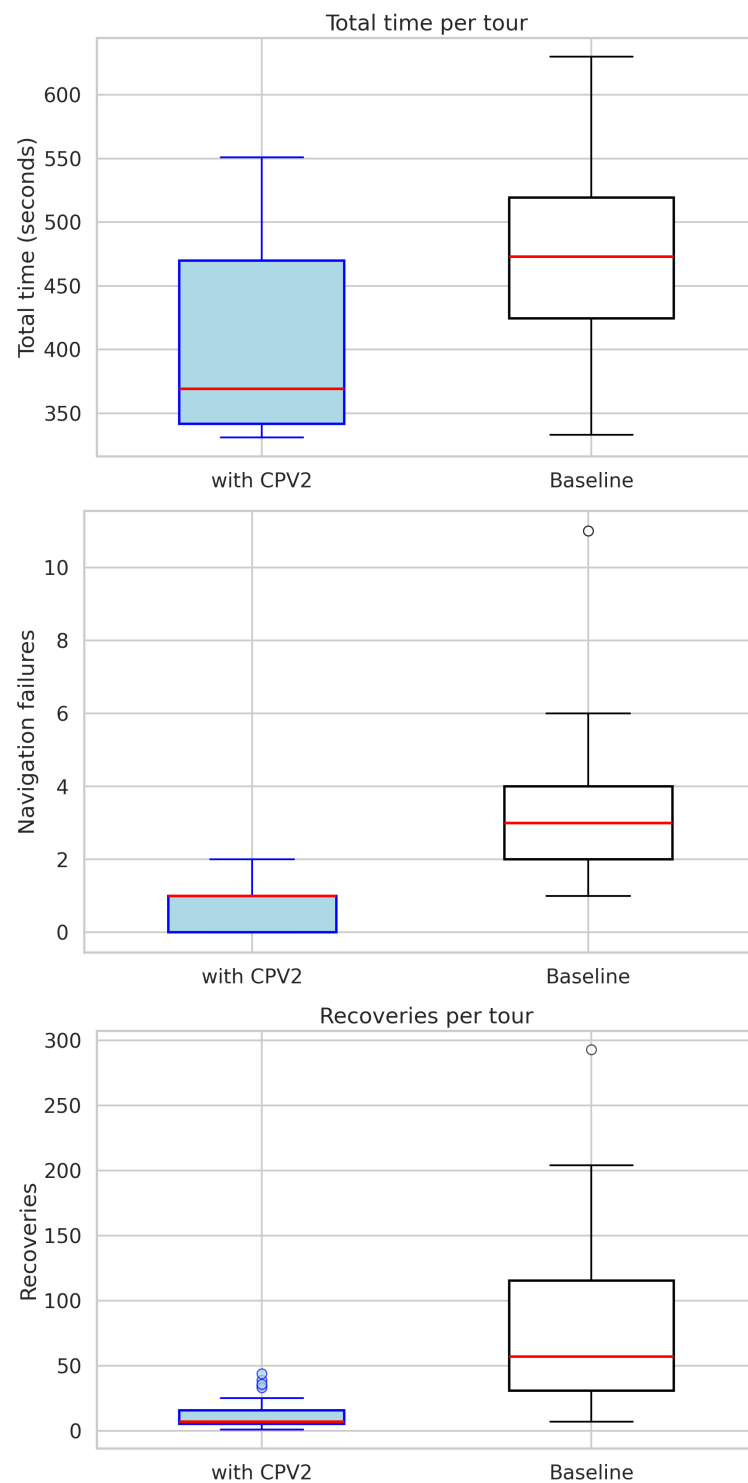


Figure 15.6 Average tour total time, number of navigation aborts and recoveries during the simulated tours.

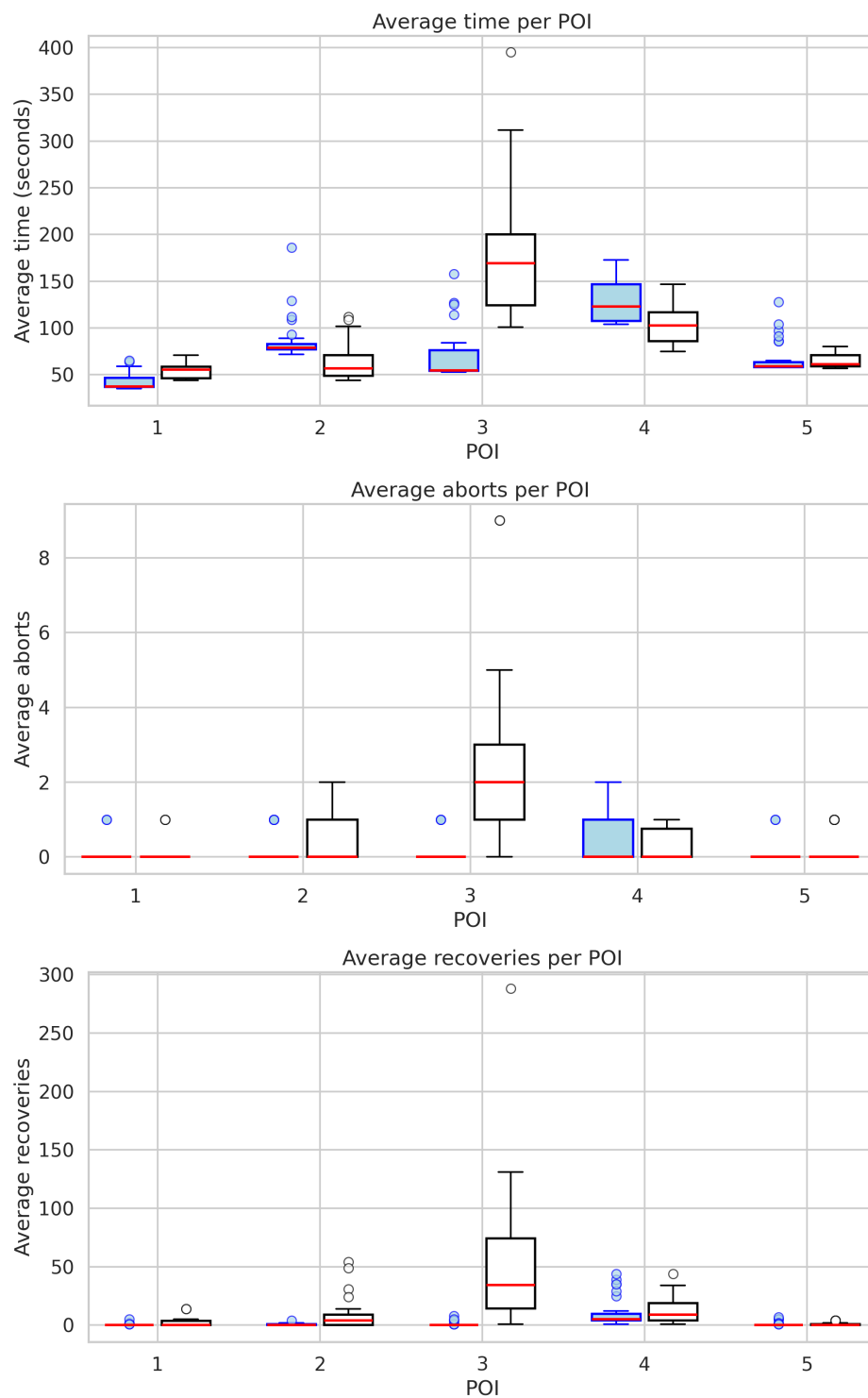


Figure 15.7 Average time, number of navigation aborts and recoveries for each POI.

# Chapter 16

## Discussion

Taken together, the results of this chapter show that the congestion-aware planner provides value not only in synthetic benchmarks, but also when integrated into the logic of a complete robotic application. Its main contribution in the museum scenario is not merely to reduce travel time, but to make the tour smoother by reducing recoveries and aborts, which are particularly visible and undesirable in front of visitors. This makes the planner relevant not only from an optimization perspective, but also from the perspective of user experience and system robustness. Quantitatively, the simulated museum evaluation shows a moderate reduction in average tour duration, about 13%, and much larger reductions in navigation recoveries and aborts, about 82% and 77% respectively. This contrast is scientifically important: it indicates that congestion-aware planning is most valuable when success is measured as stable and predictable execution, not only as shortest completion time.

At the same time, the chapter shows that planning alone is not sufficient. Its benefits emerge most clearly when combined with the monitoring and verification infrastructure described earlier, which helps ensure that the overall behavior of the robot remains correct, diagnosable, and safe during prolonged operation in a public environment.

The scale of this evaluation should also be kept in view. The 30 simulated tours per condition are sufficient to show a clear effect in the controlled museum simulation, especially for recoveries and aborts, but they are not a substitute for a long-term public deployment with real visitors. The results therefore support the claim that the planner improves the integrated system under replayed museum-flow conditions; they do not yet establish how the same benefit would vary across days, exhibitions, visitor groups, or perception conditions in open-ended operation.

## 16.1 Failure Cases and Practical Limits

The museum experiments also make it possible to identify the boundaries of effectiveness of the proposed approach. The planner is most useful when there are meaningful alternatives in the route, and when visitor congestion affects those alternatives differently over time. When the geometry of the environment forces the robot to traverse largely overlapping paths, as in the case of some POIs in the later part of the museum tour, the room for improvement is naturally reduced.

More generally, these results suggest that the benefit of flow-aware planning depends on an interaction between three factors: the structure of the topological graph, the variability of human flow, and the flexibility of the task. If any of these factors is weak, the planner may still be correct and reactive, but its effect on execution time may be limited. This is not a weakness of the implementation alone; rather, it is an inherent property of planning in structured environments with constrained route alternatives.

This case study also exposes a practical boundary of the code-generation, verification, and monitoring workflow introduced in Part I. Code generation preserves the behavior represented by the supported skill models, but it does not cover hand-written functional components or guarantee that assumptions about the middleware, timing, and physical environment will hold after deployment. Similarly, off-line verification establishes only that the modeled system satisfies the selected properties under its abstractions, while runtime property monitoring can detect only violations that occur in the observed execution and are captured by the properties defined in advance. In this application, running model monitors for all active models was considered too computationally expensive, so on-line validation was limited to the less intrusive property monitors.

The resulting evidence therefore increases confidence in the deliberative behavior and makes relevant failures observable, but it should not be interpreted as an end-to-end correctness guarantee for the complete robot.



## **Part IV**

# **Conclusion and Open Research Directions**

# Chapter 17

## Conclusion

This thesis addresses the design, implementation, verification, monitoring, and adaptation of robotic behavior in realistic public environments. Its central argument is that service robots must be both dependable and adaptive: they must execute behavior that is close to their design model and concurrently react to changes in crowded, human-populated spaces.

The first contribution is a model-based workflow for robotic control software centered on Behavior Trees, SCXML-style finite-state models, and executable skill generation. The workflow links design models to generated ROS 2 code, supports off-line verification of temporal and safety properties, and enables runtime monitoring of both requirements and model conformance. The implementation of two use cases of different complexity and the definition of their verification and monitoring properties provided end-to-end validation of this workflow and grounded MOON in concrete robotic scenarios. This reduces the gap between design and implementation, and it is especially valuable in robotics where delivered behavior emerges from the interaction of deliberative logic, skills, middleware, and perception.

The second contribution is a flow-aware planning framework for robot tour planning in crowded environments. The planner predicts human motion using CLiFF maps, maps the resulting congestion estimates to the edges of a topological tour graph, and formulates the decision problem as an MDP solved on-line with LRTDP. This allows the robot to reason beyond momentary occupancy and choose routes that remain efficient as pedestrian flow evolves. Synthetic experiments on the ATC dataset show that the planner reduces tour execution time relative to a baseline that only reacts to current congestion, while the bounded on-line variant remains compatible with real-time operation.

The third contribution is the integration of these techniques in a museum tour-guide case study, including the implementation of the use-case software, the definition of its verification

and monitoring properties. In this scenario, the system combines a rich Behavior Tree architecture, off-line verification and runtime monitoring, and an adaptive planner embedded in the tour execution flow. The museum chapter demonstrates that verification and monitoring are not isolated add-ons, but architectural elements that support correctness, diagnosability, and operator awareness, while the planner improves execution smoothness by reducing recoveries, aborts, and congestion-driven delays. The case study also shows that the planner's benefit depends on the task structure, the variability of visitor flow, and the flexibility of the tour order.

Taken together, the thesis shows that dependable adaptive autonomy emerges from combining model-based engineering with runtime observation and crowd-aware planning. The resulting framework is able to execute predefined behavior, validate it off-line, monitor it on-line, and adapt route choices as the environment changes. A key point of the thesis is that these activities are connected through explicit abstraction boundaries: abstract planner actions over topological maps are translated into BT scheduler updates, and those updates are executed through concrete plugins, skills, components, and ROS 2 interfaces that can be generated or monitored. This combination is essential for service robots operating in public spaces, where robustness is not only about avoiding failures, but also about maintaining smooth, understandable interaction with people.

# Chapter 18

## Open Research Directions

This research opens up a series of research questions to be explored in the future.

A first direction is improving the expressiveness of the model-based framework while preserving analyzability. The current workflow targets a tractable subset of Behavior Tree and SCXML semantics. Future work could extend this supported subset, better capture ROS 2 interaction patterns and additional middleware platforms, and strengthen the semantic link between Behavior Tree models and generated verification artifacts. A tighter integrated engineering environment in which modeling, code generation, verification, monitor generation, and deployment are parts of a single traceable workflow would also increase usability and confidence.

A second direction concerns runtime monitoring beyond detection. In this thesis, monitoring is primarily used to detect violations of requirements and model conformance. For long-term autonomy, violations should also be diagnosed and localized, so that the system or operator can understand whether a failure is caused by a skill, a communication channel, a sensor fault, or an environmental condition. Reducing monitoring overhead will also be important, especially for systems with many monitored interfaces or high-rate topics, so that monitoring can remain a continuous architectural service rather than a temporary validation layer.

A third direction is tighter integration between task-level planning and local navigation. The current flow-aware planner operates on a topological abstraction and therefore separates high-level route decisions from the continuous navigation layer. Future work should explore joint models in which high-level choices are informed by geometric constraints, social navigation costs, and local motion feasibility. This could also enable richer objectives beyond travel time, such as visitor comfort, social acceptability, battery consumption, and explanation quality.

A fourth direction is broader runtime adaptation. The museum case study shows that route replanning can improve tour smoothness, but route adaptation is only one form of deliberative change. Future systems should also respond to visitor requests, temporary room closures, group-size changes, or interruptions during explanations. The open challenge is to make such adaptations expressive enough to be useful while preserving verifiability and monitorability of the resulting behavior.

Finally, a key research direction is longer-term real-world deployment and human-centered evaluation. The simulated museum experiments are valuable because they execute the same code used by the real system, but long-term operation with real visitors would provide stronger evidence about robustness, usability, and tour quality. Such deployment would also allow end-to-end evaluation of the complete framework: design models, generated code, off-line verification, runtime monitoring, adaptive planning, and human-facing execution in the same operational setting. This is necessary to understand how the complete architecture behaves over time under the variability of public environments.

# References

- Rajeev Alur and Thomas A Henzinger. Real-time logics: complexity and expressiveness. *Information and Computation*, 104(1):35–77, 1993.
- Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.
- Santosh Balajee Banisetty, Scott Forer, Logan Yliniemi, Monica Nicolescu, and David Feil-Seifer. Socially aware navigation: A non-linear multi-objective optimization approach. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 11(2):1–26, 2021.
- Jim Barnett, Rahul Akolkar, RJ Auburn, Michael Bodell, Daniel C. Burnett, Jerry Carter, Scott McGlashan, Torbjörn Lager, Mark Helbing, Rafah Hosn, T.V. Raman, Klaus Reifensrath, No’am Rosenthal, and Johan Roxendal. State chart xml (scxml): State machine notation for control abstraction, 2015. URL <https://www.w3.org/TR/scxml/>. last accessed 2024-10-18.
- Marian Johannes Begemann, Hannes Kallwies, Martin Leucker, and Malte Schmitz. Tessla-ros-bridge-runtime verification of robotic systems. In *International Colloquium on Theoretical Aspects of Computing*, pages 388–398. Springer, 2023.
- Stefano Bernagozzi, Sofia Faraci, Enrico Ghiorzi, Karim Pedemonte, Angelo Ferrando, Lorenzo Natale, and Armando Tacchella. Model-based verification and monitoring for safe and responsive robots. In *2025 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pages 1–6, 2025a. doi: 10.1109/SIMPAR62925.2025.10979077.
- Stefano Bernagozzi, Sofia Faraci, Enrico Ghiorzi, Karim Pedemonte, Lorenzo Natale, and Armando Tacchella. Code generation and monitoring for deliberation components in autonomous robots. In *2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 16762–16767. IEEE, 2025b.
- Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2): 171–188, 2005.
- Blai Bonet and Hector Geffner. Labeled rtdp: Improving the convergence of real-time dynamic programming. In *ICAPS*, volume 3, pages 12–21, 2003.
- Dražen Bršćić, Takayuki Kanda, Tetsushi Ikeda, and Takahiro Miyashita. Person tracking in large public spaces using 3-d range sensors. *IEEE Transactions on Human-Machine Systems*, 43(6):522–534, 2013.

- Christoph Brzoska. Temporal logic programming with metric and past operators. In *International Joint Conference on Artificial Intelligence*, pages 21–39. Springer, 1993.
- Alexander Bubeck, Florian Weisshardt, and Alexander Verl. BRIDE—a toolchain for framework-independent development of industrial service robot applications. In *ISR/Robotik 2014; 45th International Symposium on Robotics and 8th German Conference on Robotics*, pages 137–142. VDE, 2014. doi: 10.24406/publica-fhg-384641.
- Carlos E Budde, Pedro R D’Argenio, Arnd Hartmanns, and Sean Sedwards. An efficient statistical model checker for nondeterminism and rare events. *International Journal on Software Tools for Technology Transfer*, 22(6):759–780, 2020.
- Sagar Chaki, Edmund M. Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. State/event-based software model checking. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods*, pages 128–147, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24756-2.
- Konstantinos Charalampous, Ioannis Kostavelis, and Antonios Gasteratos. Recent trends in social aware robot navigation: A survey. *Robotics and Autonomous Systems*, 93:85–104, 2017.
- Yu Fan Chen, Michael Everett, Miao Liu, and Jonathan P How. Socially aware motion planning with deep reinforcement learning. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1343–1350. IEEE, 2017.
- SF Chik, CF Yeong, ELM Su, TY Lim, Yuvashini Subramaniam, and PJH Chin. A review of social-aware navigation frameworks for service robot in dynamic human environments. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, 8(11): 41–50, 2016.
- Edmund M Clarke. Model checking. In *International conference on foundations of software technology and theoretical computer science*, pages 54–56. Springer, 1997.
- Michele Colledanchise and Lorenzo Natale. On the implementation of behavior trees in robotics. *IEEE Robotics and Automation Letters*, 6(3):5929–5936, 2021.
- Michele Colledanchise and Petter Ögren. Behavior Trees in Robotics and AI : An Introduction. *Behavior Trees in Robotics and AI*, 7 2018. doi: 10.1201/9780429489105. URL <https://www.taylorfrancis.com/books/mono/10.1201/9780429489105/behavior-trees-robotics-ai-michele-colledanchise-petter-%C3%B6gren>.
- Michele Colledanchise, Giuseppe Cicala, Daniele E. Domenichelli, Lorenzo Natale, and Armando Tacchella. Formalizing the execution context of behavior trees for runtime verification of deliberative policies. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9841–9848, 2021. doi: 10.1109/IROS51168.2021.9636129.
- Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. Tessla: temporal stream-based specification language. In *Brazilian Symposium on Formal Methods*, pages 144–162. Springer, 2018.

- Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT press, 2022.
- Silvano Dal Zilio, Pierre-Emmanuel Hladik, Félix Ingrand, and Anthony Mallet. A formal toolchain for offline and run-time verification of robotic systems. *Robotics and Autonomous Systems*, 159:104301, 2023. ISSN 0921-8890. doi: <https://doi.org/10.1016/j.robot.2022.104301>.
- Axel David, Kim G. Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Statistical model checking for networks of priced timed automata. In *FORMATS*, 2011.
- Edson de Araújo Silva, Eduardo Valentin, Jose Reginaldo Hughes Carvalho, and Raimundo da Silva Barreto. A survey of model driven engineering in robotics. *Journal of Computer Languages*, 62:101021, 2021.
- Arnab Debnath, Gregory J Stein, and Jana Košecká. A hybrid approach to indoor social navigation: Integrating reactive local planning and proactive global planning. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10432–10438. IEEE, 2025.
- Daniel Dugas, Juan Nieto, Roland Siegwart, and Jen Jen Chung. Ian: Multi-behavior navigation planning for robots in real, crowded environments. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 11368–11375. IEEE, 2020.
- Lukas Dust, Mikael Ekström, Rong Gu, Saad Mubeen, and Cristina Secceanu. A model-based methodology for automated verification of ros 2 systems. In *Proceedings of the 2024 ACM/IEEE 6th international workshop on robotics software engineering*, pages 35–42, 2024.
- Angelo Ferrando, Rafael C. Cardoso, Michael Fisher, Davide Ancona, Luca Franceschini, and Viviana Mascardi. Rosmonitoring: A runtime verification framework for ros. In Abdelkhalick Mohammad, Xin Dong, and Matteo Russo, editors, *Towards Autonomous Robotic Systems*, pages 387–399, Cham, 2020. Springer International Publishing. ISBN 978-3-030-63486-5.
- Seyedshams Feyzabadi and Stefano Carpin. Risk-aware path planning using hirerachical constrained markov decision processes. In *2014 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 297–303. IEEE, 2014.
- Janos Gertler. *Fault detection and diagnosis in engineering systems*. CRC press, 2017.
- Enrico Ghiorzi and Armando Tacchella. Execution semantics of behavior trees in robotic applications, 2024.
- Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Andrzej Wasowski, and Swaib Dragule. Behavior Trees and State Machines in Robotics Applications. *IEEE Transactions on Software Engineering*, 49(9):4243–4267, 9 2023. ISSN 19393520. doi: 10.1109/TSE.2023.3269081.
- Paul Godart, James Gross, Ranjan Mukherjee, and Wyatt Ubellacker. Generating real-time robotics control software from SysML. In *2017 IEEE Aerospace Conference*, pages 1–11. IEEE, 2017. doi: 10.1109/AERO.2017.7943610.



- Keld Helsgaun. An extension of the lin-kernighan-helsgaun tsp solver for constrained traveling salesman and vehicle routing problems. *Roskilde: Roskilde University*, 12: 966–980, 2017.
- Félix Ingrand. A formal implementation of Behavior Trees to act in robotics. *arXiv preprint arXiv:2502.11904*, 2025. URL <https://arxiv.org/abs/2502.11904>.
- Félix Ingrand and Malik Ghallab. Deliberation for autonomous robots: A survey. *Artificial Intelligence*, 247:10–44, 2017. ISSN 0004-3702. doi: <https://doi.org/10.1016/j.artint.2014.11.003>. URL <https://www.sciencedirect.com/science/article/pii/S0004370214001350>. Special Issue on AI and Robotics.
- Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter Ögren, and Christian Smith. A survey of behavior trees in robotics and ai. *Robotics and Autonomous Systems*, 154:104096, 2022.
- Richard M Karp. Reducibility among combinatorial problems. In *50 Years of Integer Programming 1958-2008: from the Early Years to the State-of-the-Art*, pages 219–241. Springer, 2009.
- Linh Kästner, Junhui Lil, Zhengcheng Shen, and Jens Lambrecht. Enhancing navigational safety in crowded environments using semantic-deep-reinforcement-learning-based navigation. In *2022 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, pages 87–93. IEEE, 2022.
- Maryam Kazemi Eskeri, Thomas Wiedemann, Ville Kyrki, Dominik Baumann, and Tomasz Piotr Kucner. A lightweight crowd model for robot social navigation. *arXiv e-prints*, pages arXiv–2508, 2025.
- Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.
- Tomáš Krajník, Jaime P Fentanes, Joao M Santos, and Tom Duckett. Fremen: Frequency map enhancement for long-term mobile robot autonomy in changing environments. *IEEE Transactions on Robotics*, 33(4):964–977, 2017.
- Thibault Kruse, Amit Kumar Pandey, Rachid Alami, and Alexandra Kirsch. Human-aware robot navigation: A survey. *Robotics and Autonomous Systems*, 61(12):1726–1743, 2013.
- Tomasz Piotr Kucner, Martin Magnusson, Erik Schaffernicht, Victor Hernandez Bennetts, and Achim J. Lilienthal. Enabling flow awareness for mobile robots in partially observable environments. *IEEE Robotics and Automation Letters*, 2(2):1093–1100, 2017. doi: 10.1109/LRA.2017.2660060.
- Tomasz Piotr Kucner, Martin Magnusson, Sariah Mghames, Luigi Palmieri, Francesco Verdoja, Chittaranjan Srinivas Swaminathan, Tomáš Krajník, Erik Schaffernicht, Nicola Bellotto, Marc Hanheide, et al. Survey of maps of dynamics for mobile robots. *The International Journal of Robotics Research*, 42(11):977–1006, 2023.
- Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. *CAV*, 2011.

- Axel Legay, Anna Lukina, Louis Marie Traonouez, Junxing Yang, Scott A Smolka, and Radu Grosu. Statistical model checking. In *Computing and software science: state of the art and perspectives*, pages 478–504. Springer, 2019.
- Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 2009.
- Dong-Won Lim and Jackyou Noh. Ros 2 application to the safety monitoring of collaborative robots. In *2024 4th International Conference on Robotics, Automation and Artificial Intelligence (RAAI)*, pages 42–46. IEEE, 2024.
- Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66): eabm6074, 2022. doi: 10.1126/scirobotics.abm6074.
- Arthur B Markman. *Knowledge representation*. Psychology Press, 2013.
- Kohei Matsumoto, Yuki Hyodo, and Ryo Kurazume. Crowd-aware robot navigation with switching between learning-based and rule-based methods using normalizing flows. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4823–4830. IEEE, 2024.
- Christoforos Mavrogiannis, Francesca Baldini, Allan Wang, Dapeng Zhao, Pete Trautman, Aaron Steinfeld, and Jean Oh. Core challenges of social robot navigation: A survey. *ACM Transactions on Human-Robot Interaction*, 12(3):1–39, 2023.
- Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. Yarp: yet another robot platform. *International Journal of Advanced Robotic Systems*, 3(1):8, 2006.
- Sergi Molina, Grzegorz Cielniak, Tomáš Krajník, and Tom Duckett. Modelling and predicting rhythmic flow patterns in dynamic environments. In *Annual conference towards autonomous robotic systems*, pages 135–146. Springer, 2018.
- Ronja Möller, Antonino Furnari, Sebastiano Battiato, Aki Härmä, and Giovanni Maria Farinella. A survey on human-aware robot navigation. *Robotics and Autonomous Systems*, 145:103837, 2021.
- Takumi Onozawa, Hiroshi Fujimoto, and Takuya Azumi. Model-based development for ros 2-based autonomous-driving software. In *2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 2511–2519. IEEE, 2023.
- Christos H Papadimitriou. The euclidean travelling salesman problem is np-complete. *Theoretical computer science*, 4(3):237–244, 1977.
- Ivan Perez. Runtime verification with ogma. In *Invited Talk to University of California*, 2023.
- Ivan Perez, Frank Dedden, and Alwyn Goodloe. Copilot 3. Technical report, NASA, 2020.
- Ivan Perez, Anastasia Mavridou, Tom Pressburger, Alexander Will, and Patrick J Martin. Monitoring ros2: from requirements to autonomous robots. *arXiv preprint arXiv:2209.14030*, 2022.

- Ola Pettersson. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53(2):73–88, 2005.
- Martin L Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 1994.
- Noé Pérez-Higueras, Fernando Caballero, and Luis Merino. Learning human-aware path planning with fully convolutional networks. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5897–5902, 2018. doi: 10.1109/ICRA.2018.8460851.
- Ansgar Radermacher, Matteo Morelli, Mahmoud Hussein, and Reda Nouacer. Designing drone systems with papyrus for robotics. In *Proceedings of the 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings*, pages 29–35. ACM, 2021. doi: 10.1145/3444950.3444956.
- Barry Ridge, Timotej Gašpar, and Aleš Ude. Rapid state machine assembly for modular robot control using meta-scripting, templating and code generation. In *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*, pages 661–668. IEEE, 2017. doi: 10.1109/HUMANOIDS.2017.8246943.
- Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Montiarcautomaton: modeling architecture and behavior of robotic systems. *arXiv preprint arXiv:1409.2310*, 2014.
- Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Code generator composition for model-driven engineering of robotics component & connector systems. *arXiv preprint arXiv:1505.00904*, 2015.
- Stefano Rosa, Marco Randazzo, Stefano Bernagozzi, and Lorenzo Natale. Palazzo madama people detection, 2026.
- Maryam Ghaffari Saadat, Angelo Ferrando, Louise A Dennis, and Michael Fisher. Ros-monitoring 2.0: extending ros runtime verification to services and ordered topics. *arXiv preprint arXiv:2411.14367*, 2024.
- Jacques Saraydaryan, Fabrice Jumel, and Olivier Simonin. Human presence probability map (hpp): A probability propagation based on human flow grid. In *Robot World Cup*, pages 54–65. Springer, 2023.
- Hessam S. Sarjoughian and Savitha Sundaramoorthi. Superdense time trajectories for devs simulation models. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, DEVS ’15, page 249–256, San Diego, CA, USA, 2015. Society for Computer Simulation International. ISBN 9781510801059.
- Christian Schlegel, Alex Lotz, Matthias Lutz, and Dennis Stampfer. Composition, separation of roles and model-driven approaches as enabler of a robotics software ecosystem. In *Software Engineering for Robotics*, pages 53–108. Springer, 2021. doi: 10.1007/978-3-030-66494-7\_3. URL [https://link.springer.com/chapter/10.1007/978-3-030-66494-7\\_3](https://link.springer.com/chapter/10.1007/978-3-030-66494-7_3).

- Yoones A. Sekhavat. Behavior Trees for Computer Games. <https://doi.org/10.1142/S0218213017300010>, 26(2), 4 2017. ISSN 17936349. doi: 10.1142/S0218213017300010. URL <https://www.worldscientific.com/worldscinet/ijait>.
- Ransalu Senanayake, Maneekwan Toyungyernsub, Mingyu Wang, Mykel J Kochenderfer, and Mac Schwager. Directional primitives for uncertainty-aware motion estimation in urban environments. In *2020 IEEE 23rd international conference on intelligent transportation systems (ITSC)*, pages 1–6. IEEE, 2020.
- Marc Sevaux, Kenneth Sörensen, et al. Hamiltonian paths in large clustered routing problems. In *Proceedings of the EU/MEeting 2008 workshop on Metaheuristics for Logistics and Vehicle Routing, EU/ME*, volume 8, pages 411–417, 2008.
- Bruno Siciliano, Oussama Khatib, and Torsten Kröger. *Springer handbook of robotics*, volume 200. Springer, 2008.
- Phani Teja Singamaneni, Anthony Favier, and Rachid Alami. Human-aware navigation planner for diverse human-robot interaction contexts. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5817–5824. IEEE, 2021.
- Phani Teja Singamaneni, Pilar Bachiller-Burgos, Luis J Manso, Anaís Garrell, Alberto Sanfeliu, Anne Spalanzani, and Rachid Alami. A survey on socially aware robot navigation: Taxonomy and future challenges. *The International Journal of Robotics Research*, 43(10): 1533–1572, 2024.
- Dennis Stampfer, Alex Lotz, Matthias Lutz, and Christian Schlegel. The SmartMDSD toolchain: An integrated MDSD workflow and integrated development environment (IDE) for robotics software. *Journal of Software Engineering for Robotics*, 7(1):3–19, 2016. ISSN 2035-3928.
- Charlie Street, Sebastian Pütz, Manuel Mühlig, Nick Hawes, and Bruno Lacerda. Congestion-aware policy synthesis for multirobot systems. *IEEE Transactions on Robotics*, 38(1): 262–280, 2022. doi: 10.1109/TRO.2021.3071618.
- Chittaranjan Srinivas Swaminathan, Tomasz Piotr Kucner, Martin Magnusson, Luigi Palmieri, and Achim J Lilienthal. Down the cliff: Flow-aware trajectory planning under motion pattern uncertainty. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7403–7409. IEEE, 2018.
- Chittaranjan Srinivas Swaminathan, Tomasz Piotr Kucner, Martin Magnusson, Luigi Palmieri, Sergi Molina, Anna Mannucci, Federico Pecora, and Achim J Lilienthal. Benchmarking the utility of maps of dynamics for human-aware motion planning. *Frontiers in Robotics and AI*, 9:916153, 2022.
- Dogan Ulus. Online monitoring of metric temporal logic using sequential networks. *arXiv preprint arXiv:1901.00175*, 2019a.
- Dogan Ulus. Timescales: A benchmark generator for mtl monitoring tools. In *International Conference on Runtime Verification*, pages 402–412. Springer, 2019b.
- Dogan Ulus. Reelay: Online temporal logic monitoring framework. *arXiv preprint arXiv:2604.22384*, 2026.

- Tomáš Vintr, Zhi Yan, Tom Duckett, and Tomáš Krajník. Spatio-temporal representation for long-term anticipation of human presence in service robotics. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 2620–2626. IEEE, 2019.
- Rui Wang, Yong Guan, Houbing Song, Xinxin Li, Xiaojuan Li, Zhiping Shi, and Xiaoyu Song. A formal model-based design method for robotic systems. *IEEE Systems Journal*, 13(1):1096–1107, 2019. doi: 10.1109/JSYST.2018.2867285.
- Ruiqi Wang, Weizheng Wang, and Byung-Cheol Min. Feedback-efficient active preference learning for socially aware robot navigation. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 11336–11343. IEEE, 2022.
- Yufei Zhu, Andrey Rudenko, Tomasz P. Kucner, Luigi Palmieri, Kai O. Arras, Achim J. Lilienthal, and Martin Magnusson. Cliff-lhmp: Using spatial dynamics patterns for long-term human motion prediction. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3795–3802, 2023. doi: 10.1109/IROS55552.2023.10342031.

# Appendix A

## ASCXML Language

### A.1 Models for Skills and Components

The complete system model is built from skills, ROS 2 nodes, *BT* plugins, and abstract models of external components such as the environment. Each element is represented as a state machine, but the concrete implementation of these elements uses middleware mechanisms that are not part of standard *SCXML*: ROS 2 topics, services, actions, timers, and *BT*-specific interactions such as ticks, halts, and ports.

For this reason, the framework uses *ASCXML*, a ROS-flavoured extension of *SCXML*. *ASCXML* keeps the state-machine structure of *SCXML*, while adding explicit constructs for the communication interfaces used by the robotic software. The goal is to make the model close enough to the implementation to support code generation and runtime monitoring, while keeping the language simple enough to translate into formal models for verification.

### A.2 SCXML Tags Excluded by ASCXML

*ASCXML* deliberately uses only a restricted subset of *SCXML*. The excluded constructs are convenient in general-purpose *SCXML* interpreters, but they either complicate the formal semantics or are not needed for the reactive robotic software considered in this thesis. Excluded tags include:

**Hierarchical states.** *ASCXML* does not allow nested state machines. Although hierarchical states can make some specifications more compact, they complicate the translation to channel systems and other formal models of asynchronous computation. Since

the SCXML standard leaves some execution details to the implementation, avoiding hierarchy also avoids ambiguity in the generated formal semantics.

**<parallel> states.** Parallel states are not supported for the same reason. The framework models concurrency through multiple communicating state machines, which is closer to the structure of distributed ROS 2 systems and easier to translate for verification.

**<final> states and <donedata>.** The control software considered here is reactive: it may remain idle, but it should still be able to respond to new events. ASCXML therefore does not use final states. Since final states are not used, <donedata> is not needed either.

**<history> pseudo-states.** History pseudo-states are excluded because they add semantic complexity without being essential for the models used in the framework. Equivalent behavior can be represented explicitly through ordinary states and variables when needed.

**<script> executable elements.** Scripts are useful when a state machine is mixed with hand-written code, for example to call external APIs. In this framework, however, the model itself must be translated and analyzed. Arbitrary executable code inside the model would make this translation difficult and would weaken the connection between the model and its formal interpretation.

**<cancel> executable elements.** The SCXML <cancel> element cancels a delayed event on a best-effort basis. ASCXML assumes that sent events are processed, so this weakly specified behavior is not used.

**<invoke> and <content> elements.** ASCXML does not create external service instances through SCXML invocation. Interactions with external computation are represented explicitly through ROS 2 topics, services, actions, and *BT* interfaces instead.

**System variables.** ASCXML supports only `_event` and the sub-fields needed to model interface calls and returns. For example, `_event.data` carries function arguments in call events and return values in response events.

## A.3 Event and Communication Semantics

ASCXML also changes how events are addressed. In standard SCXML, a <send> element may identify a target. In ASCXML, <send> has no *target* argument: when an event is sent,

every enabled transition that listens to that event may react to it. This choice matches the communication style of ROS 2 systems more closely, especially topic-based communication, where a published message can be received by all subscribers.

## A.4 ROS Extensions

ASCXML introduces dedicated tags for the ROS 2 interfaces used by the modeled software. These tags make middleware communication visible in the model instead of hiding it inside implementation code.

**Timers.** Timers model periodic callbacks. They use two tags:

- `<ros_time_rate>` declares the timer rate.
- `<ros_rate_callback>` marks the transition executed whenever the corresponding autogenerated global timer fires.

**Topics.** Topic tags model the publish-subscribe interaction:

- `<ros_topic_publisher>` declares a publisher.
- `<ros_topic_publish>` publishes a message.
- `<ros_topic_subscriber>` declares a subscriber.
- `<ros_topic_callback>` marks the transition executed when a message is received.

**Services.** Service tags model request-response communication:

- `<ros_service_server>` declares a service server.
- `<ros_service_client>` declares a service client.
- `<ros_service_handle_request>` handles an incoming request on the server side.
- `<ros_service_send_response>` sends the server response.
- `<ros_service_send_request>` sends a client request.
- `<ros_service_handle_response>` handles the corresponding client response.

**Actions.** Action tags model asynchronous goals, feedback, and results:



- `<ros_action_server>` declares an action server.
- `<ros_action_client>` declares an action client.
- `<ros_action_handle_goal>` handles an incoming goal request.
- `<ros_action_accept_goal>` accepts a goal.
- `<ros_action_reject_goal>` rejects a goal.
- `<ros_action_feedback>` reports progress while the goal is executing.
- `<ros_action_succeed>` reports a successful server-side outcome.
- `<ros_action_canceled>` reports a canceled server-side outcome.
- `<ros_action_aborted>` reports an aborted server-side outcome.
- `<ros_action_send_goal>` sends a client goal.
- `<ros_action_handle_goal_response>` handles goal acceptance or rejection on the client side.
- `<ros_action_handle_feedback>` handles client-side feedback.
- `<ros_action_handle_success_result>` handles a successful client-side result.
- `<ros_action_handle_canceled_result>` handles a canceled client-side result.
- `<ros_action_handle_aborted_result>` handles an aborted client-side result.

## A.5 Behavior Tree Extensions

ASCXML also represents the interaction between *BT* plugins and the rest of the system. This is important because, in the framework, *BT* leaves do not hide arbitrary computation: they forward ticks to skills or components and return their status explicitly.

**Ticks.** Tick tags model the normal execution step of a *BT* plugin:

- `<bt_tick>` receives a tick request.
- `<bt_return_status>` returns the plugin status.
- `<bt_tick_child>` ticks a child *BT* node.
- `<bt_child_status>` handles the response of a child node.

**Halts.** Halt tags model the reset of a running plugin:

- `<bt_halt>` receives a halt request.
- `<bt_return_halted>` reports that the halt operation has completed.
- `<bt_halt_child>` halts a child *BT* node.
- `<bt_child_halted>` observes that a child node has completed halting.

**Ports and blackboard data.** Output ports are mapped to *BT* blackboard variables shared by the plugins. Port tags expose this data exchange explicitly:

- `<bt_declare_port_in>` declares an input, read-only port.
- `<bt_declare_port_out>` declares an output, read-write port.
- `<bt_get_input>` reads an input value.
- `<bt_set_output>` writes an output value.