POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY - COMPUTER SCIENCE AND ENGINEERING

MODEL-DRIVEN DEVELOPMENT OF FORMALLY VERIFIED HUMAN-ROBOT INTERACTIONS

Doctoral Dissertation of:
Livia Lestingi

Supervisor:
Prof. Matteo Giovanni Rossi

Co-Supervisor:
Prof. Marcello Maria Bersani

Tutor:
Prof. Luciano Baresi

The Chair of the Doctoral Program:
Prof. Luigi Piroddi

2023 – XXXV Cycle

Abstract

In a near-future society, robots will no longer be confined to industrial environments but are bound to enter the service sector, where they will interact with a wide variety of people with very different needs. Efficiency-related factors drive existing software development techniques, but this will no longer suit everyday interactions. Incorporating factors related to the human behavioral and physiological state into the development process will become essential.

This research addresses these issues by proposing a model-driven approach to design, formally verify, and adjust scenarios where human-robot interaction is a primary element. Target users of the approach are professionals in charge of designing the robotic application but lacking solid technical background, which motivates the high degree of automation of the whole development toolchain.

The entry point to the approach is a user-friendly Domain-Specific Language to specify the scenario and the robotic mission under analysis. A formal model of the scenario based on Stochastic Hybrid Automata is then automatically generated. The approach captures a set of physiological traits—including physical fatigue—and behavioral traits capturing the possibility of the human making haphazard decisions. The formal model is subject to Statistical Model Checking to estimate the most likely mission's outcome.

Subsequently, the approach features a deployment framework to deploy the scenario in a real setting or simulate it in a virtual environment for further investigation. The configured model of the interactive scenario is transformed into an executable version to ensure that properties formally

verified at design time also hold at runtime.

Data collected during deployment are exploited to infer an updated model of human behavior and adjust the robotic mission accordingly. To this end, we introduce an automata learning algorithm called $\mathsf{L}^*_{\mathrm{SHA}}$ specifically targeting Stochastic Hybrid Automata. The learned model of human behavior is plugged into the Stochastic Hybrid Automata network to perform a new round of verification and revise the mission's design, if necessary.

All phases of the model-driven approach—the design-time analysis, deployment, and automata learning—have been empirically validated on realistic case studies inspired by healthcare scenarios. The formal foundation is a key component in guaranteeing the dependability of the resulting software components. At the same time, the high-level abstraction level and the presence of a learning procedure promote the framework's flexibility.

Keywords: Service Robotics, Human-Robot Interaction, Formal Methods for Robotics, Statistical Model Checking, Model-driven Engineering, Domain-specific Languages for Robotics, Stochastic Hybrid Automata, Models of Human Behavior, Software Engineering for Robotics, Automata Learning

Sommario

In un futuro prossimo, i robot non saranno piú confinati all'ambito industriale ma verranno largamente impiegati nel settore dei servizi, dove interagiranno con un'ampia varietá di persone con diverse esigenze. Fattori legati all'efficienza sono alla base di tecniche di ingegneria del software giá esistenti, mació non sará piú sufficiente per le interazioni tra uomo e robot nella vita di tutti i giorni. Diventerá, infatti, essenziale incorporare fattori legati allo stato comportamentale e fisiologico umano nel processo di sviluppo del software.

Questo progetto di ricerca affronta quest'esigenza emergente proponendo un framework *model-driven* per progettare, verificare formalmente e riconfigurare scenari in cui l'interazione uomo-robot é un elemento primario. Gli utenti a cui é destinato l'approccio sono figure professionali responsabili per la progettazione dell'applicazione robotica ma privi di un solido background tecnico, il che motiva l'elevato grado di automazione del processo di sviluppo.

Il punto di ingresso al framework é un domain-specific language accessibile per specificare le caratteristiche dello scenario e la missione robotica soggetti ad analisi. Successivamente, viene generato automaticamente un modello formale dello scenario basato su Stochastic Hybrid Automata. L'approccio include una serie di tratti fisiologici dei soggetti umani, inclusa la fatica fisica, e tratti comportamentali, quali la possibilitá che l'essere umano prenda decisioni inaspettate. Il modello formale é soggetto a Statistical Model Checking per stimare l'esito piú probabile della missione robotica.

In una fase successiva, il framework prevede un approccio per il *deployment* dello scenario in un ambiente reale o la simulazione in un ambiente virtuale per ulteriori indagini. Il modello formale dello scenario interattivo viene trasformato in una versione eseguibile per garantire che le proprietá verificate formalmente in fase di progettazione valgano anche in fase di esecuzione.

I dati raccolti sul campo durante l'esecuzione della missione vengono sfruttati per apprendere un modello aggiornato del comportamento umano e adattare di conseguenza la missione robotica. A tal fine, introduciamo un algoritmo di *automata learning* chiamato $\mathsf{L}^*_{\mathrm{SHA}}$ specifico per Stochastic Hybrid Automata. Il modello appreso del comportamento umano viene inserito nella rete di Stochastic Hybrid Automata per eseguire un nuovo ciclo di verifica e modificare la missione, se necessario.

Tutte le fasi del framework—l'analisi in fase di progettazione, l'implementazione e l'apprendimento degli automi—sono state convalidate empiricamente su casi di studio realistici ispirati a scenari dall'ambito medico-sanitario. La base formale é una componente chiave per garantire l'affidabilitá dei componenti software risultanti. Allo stesso tempo, l'alto livello di astrazione e la presenza di una procedura di apprendimento favoriscono la flessibilitá del framework.

Parole Chiave: Robotica di Servizio, Interazione Uomo-Robot, Metodi Formali per la Robotica, Statistical Model Checking, Metodi Model-Driven, Domain-Specific Language per la Robotica, Stochastic Hybrid Automata, Modelli del Comportamento Umano, Ingegneria del Software per la Robotica, Automata Learning

Contents

1 Introduction **21**
1.1 Context and Motivations 21
1.2 Model-Driven Framework's Overview 23
1.3 Contributions 26
1.4 Dissemination 27
1.5 Structure of the Thesis 28

2 Related Work **31**
2.1 HRI Formal Modeling and Verification 31
2.1.1 Temporal Logic-based Robotic Applications Modeling 32
2.1.2 State-based Robotic Applications Modeling 32
2.1.3 Formalizations of Human Behavior 33
2.1.4 Verification Techniques and Tools 34
2.2 Robotic Domain-Specific Languages 36
2.2.1 DSLs for Scenario Building. 37
2.2.2 DSLs for Human-Robot Interaction 37
2.3 Erroneous Human Behavior Modeling 39
2.4 Robotic Applications Deployment 40
2.5 Automata Learning 43
2.6 Human Behavior Prediction 47

3 Background **51**
3.1 Theoretical Preliminaries 51
3.1.1 Stochastic Hybrid Automata 52

3.1.2 Statistical Model Checking 59
3.2 Pre-Existing Tools 60
3.2.1 Uppaal 61
3.2.2 Robot Operating System 61
3.2.3 CoppeliaSim 62

4 Model-Driven Framework 63
4.1 Design-Time Analysis 64
4.1.1 Scenario Configuration 64
4.1.2 Robotic Mission Verification 66
4.2 Scenario Reconfiguration 66
4.3 Scenario Deployment 67
4.4 Model Adjustment 69

I Design-Time Analysis 73

5 Human-Robot Interactive Scenario Configuration 75
5.1 Conceptual Model 75
5.2 Domain-Specific Language 80
5.2.1 Layout, Areas, and POIs 82
5.2.2 Agents 83
5.2.3 Missions 85
5.2.4 Queries 85

6 Formal Modeling Human-Robot Interactions 87
6.1 Formal Modeling Approach 88
6.2 Robotic System Model 90
6.2.1 Robotic Platform Model 91
6.2.2 Battery Model 93
6.3 Human Behavior Modeling Approach 95
6.3.1 Human Follower 99
6.3.2 Human Leader 105
6.3.3 Human Recipient 106
6.3.4 Human Competitor 108
6.3.5 Human Rescuer 109
6.3.6 Human Applicant 111
6.4 Orchestrator Model 112
6.4.1 r_{idle} Submachine 115
6.4.2 r_{rech} Submachine 115
6.4.3 r_{lead} Submachine 118

6.4.4 h_{lead} Submachine 118
6.4.5 hr_{int} Submachine 119
6.4.6 r_{sync} Submachine 120
6.5 ROS Publisher Queue Model 123

7 Formal Modeling Approach Validation **127**
7.1 Case Study 1: Fatigue Profile Impact Analysis 128
7.2 Case Study 2: Multi-Human Multi-Robot Scenario 130
7.2.1 Scalability Analysis Results 133

8 Erroneous Human Behaviors Model **137**
8.1 Phenotypes of Erroneous Human Behavior 137
8.1.1 Disobey/Obey Add-On 142
8.1.2 Free Will Add-On 142
8.1.3 Timer Expired Add-On 144
8.1.4 Safety Violation Add-On 146
8.1.5 Critical Status Add-On 150
8.2 Human-Robot Interaction Patterns 152
8.2.1 HumanApplicant Pattern 152
8.2.2 HumanFollower Pattern 156
8.2.3 HumanLeader Pattern 157
8.2.4 HumanRecipient Pattern 157
8.2.5 HumanCompetitor Pattern 158
8.2.6 HumanRescuer Pattern 158

9 Experimental Analysis of Human Errors' Impact **161**
9.1 Experimental Setting 162
9.2 Experimental Results 164

II Scenario Deployment **173**

10 Deployment Approach **175**
10.1 Deployment Framework Architecture 175
10.2 Model-to-Code Mapping 177
10.3 Deployable Code Patterns 181
10.3.1 Controllable Switch Pattern 182
10.3.2 Uncontrollable Switch Pattern 183
10.3.3 Sensor Reading Pattern 185
10.3.4 System Monitoring Pattern 186

11 Deployment Framework Validation **189**

11.1 Validation Goals 189
11.2 Experimental Setting 190
11.3 Experimental Validation Process 191
11.4 Experiment 1: Charge-Critical Configuration 194
11.5 Experiment 2: Fatigue-Critical Configuration 196
11.6 Discussion 197

III Model Adjustment **199**

12 L^*_{SHA} for Stochastic Hybrid Automata Learning **201**
12.1 System Under Learning Configuration 201
12.2 L^*_{SHA} Algorithm 205
12.2.1 Signals and Traces 207
12.2.2 Observation Tables 208
12.2.3 Knowledge Refinement 217
12.2.4 Counterexamples 220
12.2.5 Complete L^*_{SHA} Algorithm 222
12.2.6 Correctness, Termination, and Complexity 222

13 L^*_{SHA} Empirical Validation **227**
13.1 Validation Process 227
13.2 CPS1: Room Temperature Control System 229
13.3 CPS2: Energy Consumption of Machining Centers 232

14 Human Behavior Model Adjustment **245**
14.1 Data-Driven Fatigue Estimation 246
14.2 Integrating Learned SHA in the SHA Network 248

15 Human Behavior Learning Validation **257**
15.1 Model-Driven Experiments 257
15.2 Simulation-Driven Experiments 263

IV Overall Framework Validation **267**

16 Model-Driven Framework Validation **269**
16.1 Validation Goals 270
16.2 G1: Formal Model Validation 272
16.3 G2: DSL and Design-Time Analysis Validation 281
16.3.1 Real Case Studies Coverage Analysis 281
16.3.2 DTA of Realistic Healthcare Scenarios 281

16.4 G3: Model Adjustment Impact Analysis 288
16.5 Discussion 293

17 Conclusions and Future Work **297**
17.1 Conclusions 297
17.2 Future Research Outlook 300

A Learned SHA **305**
A.1 Thermostat CPS 305
A.2 Human-Robot Interaction CPS (Model-Driven Experiments) 310

B Full DSL Sources **313**

Bibliography **317**

List of Figures

1.1 High-Level Framework Workflow Representation 23
1.2 DSL Sample Scenario Representation 25

2.1 Scheme of the Interaction between the L* Learner and Teacher. 44

3.1 SHA Network Example 55
3.2 Extended SHA modeling the room 56

4.1 Model-Driven Framework Workflow 65

5.1 Class Diagram of the Conceptual Model 77
5.2 DSL to Model Conversion Workflow 81

6.1 $\langle\mathsf{op}\rangle$_pub$_{\langle\mathsf{id}\rangle}$ Modeling Pattern 91
6.2 SHA Modeling the Robotic Platform 91
6.3 SHA Modeling the Robot's Battery 93
6.4 SHA Modeling the $\langle\mathsf{op}\rangle$_pub Pattern 95
6.5 SHA Modeling the Base HumanFollower Pattern 100
6.6 Disobey/Obey add-on 101
6.7 Free Will SHA add-on 103
6.8 SHA Modeling the HumanFollower Pattern 105
6.9 SHA Modeling the HumanLeader Pattern 106
6.10 SHA Modeling the HumanRecipient Pattern 107
6.11 SHA Modeling the HumanCompetitor Pattern 108
6.12 SHA Modeling the HumanRescuer Pattern 110

6.13 SHA Modeling the HumanApplicant Pattern 111
6.14 SHA Modeling the Orchestrator 113
6.15 ⟨op⟩_chk SHA pattern 114
6.16 Orchestrator SHA submachines 117
6.17 hr_{int} Orchestrator Submachine 119
6.18 r_{sync} Orchestrator Submachine 121
6.19 SHA Modeling the ros_pub$_{\langle id \rangle}$ Node 123
6.20 ros_pub$_{\langle id \rangle}$-⟨op⟩_pub$_{\langle id \rangle}$ Synchronization 124

7.1 Experimental Setup and Results for Case Study 1 128
7.2 Second Case Study Floor Layout and Workflow 131

8.1 Timer Expired add-on 144
8.2 Safety Violation add-on 147
8.3 Critical Status add-on 150
8.4 Extended SHA modeling the HumanApplicant pattern . . . 153

9.1 First Floor Layout for Human Errors' Impact Analysis . . . 162
9.2 Second Floor Layout for Human Errors' Impact Analysis . . 162
9.3 Bar Plots Reporting the Estimated Probability of Success . . 166

10.1 Formal Model-to-Deployment Infrastructure Mapping Diagram 176

11.1 Experimental Setting for Deployment Framework Validation 191
11.2 Experiment 1 Simulation Plot 195
11.3 Experiment 2 Simulation Plot 196

12.1 L^*_{SHA} Application Framework 202
12.2 L^*_{SHA} Workflow 206
12.3 Portion of observation table of the SHA in Fig. 3.1a 209
12.4 Example of sampled signal factorization 213

13.1 L^*_{SHA} Performance on CPS1 231
13.2 Example SHA modeling a machine's energy behavior for CPS2 232
13.3 Examples of controller SHA for CPS2 235
13.4 Example of field data for part type i 239
13.5 Example of L^*_{SHA} for CPS2 240
13.6 Sampled signal of the spindle power for CPS2 242

14.1 Processing stages of a sample EMG signal 246

14.2 SHA modeling human behavior resulting from $\mathsf{L}^*_{\mathrm{SHA}}$ 248
14.3 Non-Critical Operating Condition Transformation Pattern . . 249
14.4 Critical Operating Condition Transformation Pattern 249
14.5 Edge Transformation Pattern 250
14.6 SHA resulting from the application of the three patterns . . . 251

15.1 Benchmark HRI scenario to evaluate $\mathsf{L}^*_{\mathrm{SHA}}$ 258
15.2 Learned SHA for Exp.3 258
15.3 $\mathsf{L}^*_{\mathrm{SHA}}$ Performance for HRI CPS for Model-Driven Experiments 262
15.4 SHA Learned for Exp. 3a and 3b 263
15.5 $\mathsf{L}^*_{\mathrm{SHA}}$ Performance for HRI CPS for Simulation-Driven Experiments 264

16.1 Layout used for the experimental validation. 271
16.2 Hybrid deployment environment 271
16.3 Battery Voltage During a Discharge/Recharge Cycle 277
16.4 Average Time Spent by Case Study Subjects in Learned States 292

17.1 Vision for Future Framework Extensions 301

A.1 Mapping between learned SHA (top) and reference SHA (bottom) for Exp.1. 305
A.2 Mapping between learned SHA (top) and reference SHA (bottom) for Exp.2. 306
A.3 Mapping between learned SHA (top) and reference SHA (bottom) for Exp.3. 306
A.4 Mapping between learned SHA (top) and reference SHA (bottom) for Exp.4. 307
A.5 Mapping between learned SHA (top) and reference SHA (bottom) for Exp.5. 307
A.6 Mapping between learned SHA (top) and reference SHA (bottom) for Exp.6. 308
A.7 Mapping between learned SHA (top) and reference SHA (bottom) for Exp.7. 308
A.8 Mapping between learned SHA (top) and reference SHA (bottom) for Exp.8. 308
A.9 Mapping between learned SHA (top) and reference SHA (bottom) for Exp.9. 309
A.10 Mapping between learned SHA (top) and reference SHA (bottom) for Exp.10. 310

A.11 Mapping between learned SHA (top) and reference SHA (bottom) for Exp.1. 310
A.12 Mapping between learned SHA (top) and reference SHA (bottom) for Exp.2. 311
A.13 Mapping between learned SHA (top) and reference SHA (bottom) for Exp.3. 311
A.14 Mapping between learned SHA (top) and reference SHA (bottom) for Exp.4. 311
A.15 Mapping between learned SHA (top) and reference SHA (bottom) for Exp.5. 312

List of Tables

2.1 HRI formal modeling survey summary 35
2.2 Robotic Domain-Specific Languages survey summary . . . 38
2.3 Erroneous Human Behavior Modeling survey summary . . . 41
2.4 Robotic Application Deployment survey summary 42
2.5 Active Automata Learning survey summary 46
2.6 Human Behavior Prediction techniques survey summary . . 49

6.1 SHA Modeling Principle 90
6.2 Updates for the $\mathcal{A}_{\mathrm{b}}$ SHA modeling the robot's battery. . . . 95
6.3 Updates for the SHA modeling the HumanFollower and HumanLeader patterns. 101
6.4 Orchestrator's Submachines Start and Stop Conditions . . . 116

7.1 First Case Study Performance Data for Formal Model Validation 130
7.2 SMC Results for Second Case Study 133
7.3 Second Case Study Performance Data 134

8.1 Phenotypes of human erroneous behavior 138
8.2 Add-Ons-to-Phenotype Mapping 141

9.1 Scenarios for Human Errors' Impact Analysis 163
9.2 Erroneous Behavior Profiles 164

10.1 Mapping Between SHA Features and Deployment Units . . 178

10.2 Controllable Switch Code Pattern 182
10.3 Uncontrollable Switch Code Pattern 184
10.4 Sensor Reading Code Pattern 185
10.5 System Monitoring Code Pattern 187

11.1 Parameter Set for Deployment Framework Validation 192
11.2 Deployment Framework Validation Metrics 193

13.1 CPS1 L^*_{SHA} Accuracy Metrics 230
13.2 CPS2 Design of Experiments 237
13.3 CPS2 L^*_{SHA} Performance Metrics for Scenario A 238
13.4 CPS2 L^*_{SHA} Performance Metrics for Scenario B 238
13.5 CPS2 L^*_{SHA} Performance Metrics for Scenario C 239
13.6 CPS2 L^*_{SHA} Accuracy Metrics 242

15.1 L^*_{SHA} Accuracy Metrics for HRI CPS Model-Driven Experiments 260
15.2 L^*_{SHA} Accuracy Metrics for HRI CPS Simulation-Driven Experiments 261

16.1 Scenarios used for the formal model validation phase 273
16.2 Formal Analysis Accuracy Metrics 275
16.3 Scenarios for Design-Time Analysis Validation 280
16.4 Results of the DSL2SHA Metric Calculation 284
16.5 Reconfiguration Measures for scenarios DPa, DPb, and DPc 286
16.6 Results of the DSL2SHA Metric Calculation for Reconfigured Scenarios 287
16.7 Model Adjustment Impact Analysis Results 289

Listings

5.1 Example DSL section defining layout areas and POIs. 83
5.2 Example DSL section defining the agents and their features. 84
5.3 Example DSL section defining the mission (i.e., the sequence of services). 86
5.4 Example DSL section defining the set of queries. 86
B.1 DSL section defining layout areas 313
B.2 DSL section defining the robot 314
B.3 DSL section defining the human subjects 314
B.4 DSL section defining the robotic missions 314
B.5 DSL section defining the queries 315

List of Algorithms

1 Learner function to fill empty cells in $\langle S, E, \mathcal{Z}, T\rangle$ with TEACHER replies to mi and ht queries. 211

2 Teacher function that, given string tr, returns all segments of sampled signals in Σ_{obs} that follow tr. 212

3 Teacher function computing the answer to a mi query for a specific trace tr. 212

4 Function within the TEACHER returning the answer to a ht query. 214

5 Learner function to make an observation table closed (multiple iterations might be necessary). 215

6 Learner function to make an observation table consistent (multiple iterations might be necessary). 216

7 Function within the TEACHER to run ref queries. 218

8 Function within the TEACHER that returns a counterexample to $\langle S, E, \mathcal{Z}, T\rangle$ if it finds one, $\perp$ otherwise. 220

9 Main $\mathsf{L}^*_{\text{SHA}}$ algorithm. 221

10 Estimation of the success probability CI for a set of deployment traces 273

CHAPTER 1

Introduction

This chapter introduces the technological and societal drivers for this research and its relevance to the service robotics and software engineering fields. A general overview of the presented framework for the development of interactive service robotic applications is provided, showcasing each contribution and its role within the bigger picture through an illustrative example. Finally, the chapter lists the publications resulted from this research project and summarizes the structure of the thesis.

1.1 Context and Motivations

Breakthrough technological advancements are shaping the future of the service sector. Innovations brought by the phenomenon known as Industry 4.0, such as IoT, pervasive sensorization, Cloud Computing, and Collaborative Robotics, are now spreading to non-industrial settings with significant projected impacts on our everyday lives. Most importantly, highly sophisticated robotic systems under development today are bound to transform the job market once they become commercially available.

The uptake of such solutions poses several problems ranging from technological challenges to ethical and societal implications. A recent study on the future of employment indeed estimates that specific jobs, such as receptionists, information clerks, healthcare support workers, and personal care aides, will be taken over by robots with probabilities ranging from 60% to 90% [71].

In addition, the presence of robots in healthcare has increased in recent years and shows an accelerating trend [160]. The use and penetration of robotics for human care and aid are evidenced by the presence of European calls and projects[1], technology companies[2], and market analysis reports [64, 83].

All these initiatives and companies agree that using robots in care can increase service quality. However, robots are not a substitute for humans but a tool to improve their actions. For instance, a study by the American Nurses Association [13] showed that robots could support and augment nursing care delivery, improve nurse productivity, increase patient time, and encourage positive emotional responses.

Despite this evidence, ongoing research investigates the extent of such a technological and societal shift. This work attempts to answer the question of the feasibility of such a step by addressing the analysis from the software engineering standpoint. In particular, it sheds light on the development of collaborative service robot applications in healthcare.

State-of-the-art technologies dealing with sensing, manipulation, and reasoning capabilities make it feasible for robots to perform complex jobs. Nowadays, a robot may be adequately equipped to sense multiple aspects of its surroundings, efficiently detect obstacles, grasp and manipulate fragile objects, perform surgery, and make decisions in delicate situations. However, these skills usually constitute silos of software, whose integration and reuse are challenging tasks. The EFFIROB project [69], which analyzed the profitability of developing a new service robot application, has estimated that up to 80% of the total cost comes from software development and maintenance.

More generally, software engineering techniques for robotics are not mature yet to handle the complexity and changeability of service settings [77]. Service robots operate in unconstrained environments where humans, who they frequently interact with, constitute a significant source of uncertainty. Decisions made at an early design stage of the application deter-

[1] Examples are the Harmony (https://harmony-eu.org) and the EnrichME (https://cordis.europa.eu/project/id/643691) projects.

[2] Examples are Kompai Robotics (https://kompairobotics.com) and Labrador Systems (https://labradorsystems.com).

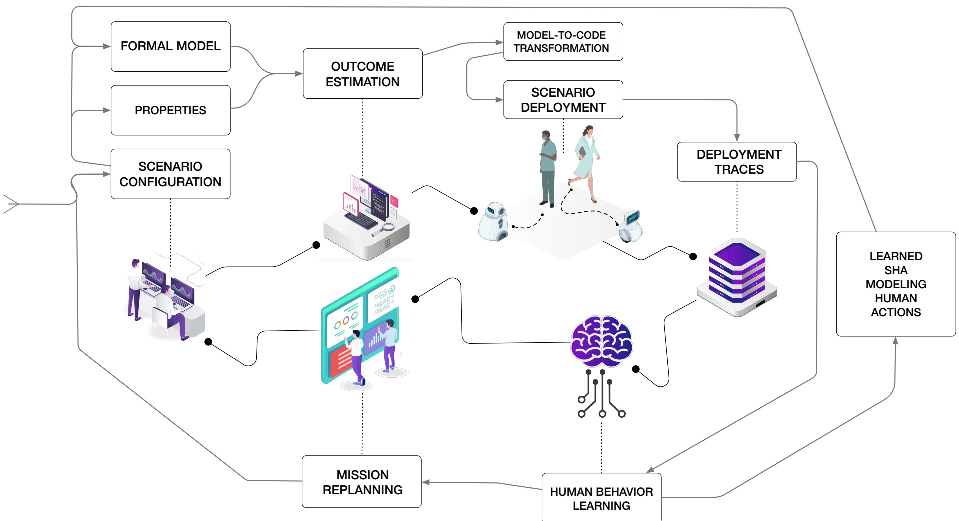


Figure 1.1: *Diagram representing the workflow of the model-driven framework, as seen in [129]. The approach has a cyclical structure, starting with configuring the interactive scenario and evaluating its outcome. The scenario is deployed, and the collected deployment traces are used to learn a model of human actions. The updated model is used to reconfigure the robot mission, if necessary.*

mine up to 90% of the overall life-cycle costs [57], and numerous sources of uncertainty can hinder their validity. Therefore, it is of paramount importance to provide designers with frameworks to develop applications that are simultaneously **reliable** and **flexible** concerning the variability of the environment [30]. Frameworks should also limit the gap between the developer's knowledge and the prerequisites needed to access them, removing the barriers due to the developers' lack of specialized skills.

1.2 Model-Driven Framework's Overview

Designing robotic applications to be deployed in delicate environments where robots will closely interact with humans is challenging, requiring a solid technical background in robotics and software design. This thesis contributes to this line of research by proposing a **model-driven framework** to develop interactive service robot applications.

Target *users* of the framework, called hereafter robotic application designers (or application designers), are professional figures managing the logistics of service facilities where robotic applications will be deployed, such as clinical workflow analysts [171].

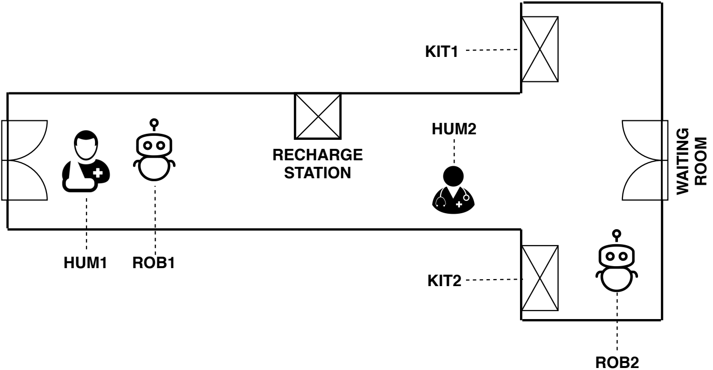
The framework targets robotic applications set in known layouts, fea-

turing a wheeled mobile robot and one (or multiple) humans requesting a service that requires interaction or coordination with the robot. Robot fleets are supported under the constraint that only one is active at a time (i.e., the other robots are available in case a replacement is necessary). The service sequence is also assumed to be fixed; specifically, the framework does not capture situations in which humans make new requests or change the requested service *while* the mission is being executed.

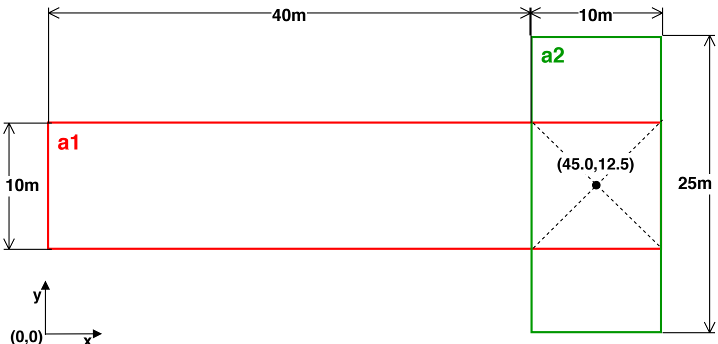
While the geometry of the layout is known, humans are a source of uncertainty as they may make unpredictable choices and stray from the plan while interacting with the robot. Applications eligible for analysis come, though not exclusively, from the healthcare and assisted living settings, where people might be in pain or discomfort. Therefore, the development process encompasses features of human **physiological** (i.e., physical fatigue) and **behavioral** aspects, such as the unpredictability of the human decision-making process. To this end, for the framework to be applicable, agents must be equipped with sensors detecting their position within the layout (e.g., Indoor Positioning Systems), biometric factors (for humans), and level of charge (for robots).

Within the framework's scope, interactions between humans and robots conform to high-level "*patterns*" identifying recurring contingencies in assistive applications. Throughout the thesis, we use the term "*action*" to indicate an "*atomic behavior that any actor executes in a scene*" [104]. The framework approximates both the human body and robots' threedimensional envelopes as *points* (i.e., corresponding to their center of mass). Therefore, we consider an *atomic* action either a displacement of such point (e.g., the human walking) or the anticipation of a synchronization with another agent (e.g., the robot waiting for another robot in the fleet to replace them). Each interaction pattern is a sequence of actions (e.g., move until a specific event occurs, stop, wait for the human to be close).

Although there is no standard definition of robotic "*mission*", with this term, we refer to a sequence of interaction patterns identifying the desired behavior of the robot [76] performed in a specific layout. The mission ends in "*success*" if all services in the sequence are brought to completion. On the other hand, the mission ends in "*failure*" if the active robot gets fully discharged (thus, it can no longer move autonomously) and no other robot is available to replace it or one of the human reaches an excessive level of physical effort that does not allow them to continue with the operations safely. A sequence of missions constitutes a Human-Robot Interaction (HRI) "*scenario*". Hence, in the scope of this work, we understand a robotic application as the realization of a scenario through real or



(a) *Agents (represented in their starting locations) and POIs of the running example.*



(b) *Layout for the running example, highlighting the two areas, their dimensions ([m]), and the intersection point.*

Figure 1.2: *Graphical representation of the example scenario configuration.*

virtual agents.

The framework exploits formal analysis to provide the robotic application designer with reliable insights into the outcome of each mission (each analyzed individually) constituting the scenario. Given the initial configuration of a scenario (e.g., positions of the agents, battery charge), the application designer receives an estimation of how likely the associated missions are to end in success (dually, in failure) and the physical effort each mission imposes on human subjects.

In the following, we present an illustrative scenario analysable through the framework, whose initial setup is shown in Fig. 1.2. The layout is a T-shaped corridor made up of two rectangular areas (see Fig. 1.2b): a horizontal one and a perpendicular vertical one, whose intersection is centered in point $(45.0, 12.5)$. The corridor features four POIs (i.e., elements the agents can interact with, represented in Fig. 1.2a): the robot's recharge sta-

tion, two cupboards containing medical kits (referred to as KIT1 and KIT2), and the door leading to the waiting room. There are four agents in the scenario: two humans (HUM1 and HUM2) and two robots (ROB1 and ROB2). We assume that ROB1 is a Tiago[3] with $C_0 = 40\%$ and ROB2 is a Turtlebot3 WafflePi[4] with $C_0 = 90\%$ (where C_0 is the initial charge). HUM1 is a patient exhibiting a Young/Sick fatigue profile with average walking speed $v = 80\mathrm{cm/s}$ while HUM2 is an Elderly/Healthy doctor with $v = 100\mathrm{cm/s}$. The designer wants to model and assess two alternative missions: the first one features ROB1 leading the patient to the waiting room, then delivering KIT2 to the doctor. The second mission features ROB2 following the doctor to fetch KIT1 and then leading the patient to the waiting room.

The framework's workflow (shown in Fig. 1.1) is structured into three macro-phases:

(1) **design-time analysis**: the application designer configures the scenario through a specification language. Starting from the configuration, a formal scenario model is automatically generated together with a set of path formulae and state formulae (referred to as "*properties*"). Properties are subject to verification. The probability of a property holding quantifies a "*quality measure*" (or a key performance indicator) of the scenario (for example, its probability of success);

(2) **deployment**: when the design-time results are deemed acceptable, the application designer deploys the scenario either in a physical environment or simulated environment; to enable the deployment, the formal model is converted into executable software components. The latter communicate with agents within the deployment environment through a middleware layer;

(3) **model adjustment**: field data collected through deployment are fed to a learning technique to infer a refined model of human behavior and iterate the formal analysis. The application designer examines the refined quality metrics of the scenario and applies reconfiguration measures, if necessary.

1.3 Contributions

Specifically, the contributions presented in this thesis are:

[3]Technical specifications available at: https://pal-robotics.com/robots/tiago/.
[4]Technical specifications available at: https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/.

1. A custom **Domain-Specific Language** (DSL) to specify the scenario under analysis in an accessible, user-friendly manner [131]. The DSL is a lightweight textual notation expressing the features of the formal model that may vary between different scenarios.
2. A **formal modeling** approach exploiting Stochastic Hybrid Automata to model the entities involved in the scenario (i.e., the layout, the robots, and the humans), which incorporates a stochastic characterization of human physiological and behavioral aspects [125–127, 130].
3. A **deployment framework** to run the scenario in a physical, simulated, or hybrid (partially physical and partially simulated) environment [128]. A model-to-code mapping function converts the formally modeled entities into executable software components to ensure that the deployed system behaves correspondingly to the verified model.
4. An active **automata learning** algorithm targeting Stochastic Hybrid Automata. The algorithm exploits signal processing and statistical techniques to infer a formal model from field data. The algorithm is domain-agnostic and has been tested against different case studies capturing different Cyber-Physical Systems, mainly to learn a refined model of human behavior as part of the development framework [129].
5. Extensive **experimental validation** assessing the coverage, accuracy, and effectiveness of all the framework phases on illustrative realistic scenarios inspired to the healthcare setting [129].

1.4 Dissemination

The research presented in this thesis resulted in the following publications:

- [127] Livia Lestingi, Mehrnoosh Askarpour, Marcello M Bersani, and Matteo Rossi. Statistical model checking of human-robot interaction scenarios. In *First Workshop on Agents and Robots for reliable Engineered Autonomy*, pages 9–17, 2020
- [125] Livia Lestingi, Mehrnoosh Askarpour, Marcello M Bersani, and Matteo Rossi. Formal verification of human-robot interaction in healthcare scenarios. In *SEFM*, pages 303–324. Springer, 2020
- [126] Livia Lestingi, Mehrnoosh Askarpour, Marcello M Bersani, and Matteo Rossi. A model-driven approach for the formal analysis of human-robot interaction scenarios. In *IEEE SMC*, pages 1907–1914, 2020

- [128] Livia Lestingi, Mehrnoosh Askarpour, Marcello M. Bersani, and Matteo Rossi. A deployment framework for formally verified human-robot interactions. *IEEE Access*, 9:136616–136635, 2021
- [130] Livia Lestingi, Cristian Sbrolli, Pasquale Scarmozzino, Giorgio Romeo, Marcello M Bersani, and Matteo Rossi. Formal modeling and verification of multi-robot interactive scenarios in service settings. In *Intl. Conf. on Formal Methods in Software Engineering*, pages 80–90, 2022
- [129] Livia Lestingi, Marcello M Bersani, and Matteo Rossi. Model-driven development of service robot applications dealing with uncertain human behavior. *IEEE Intelligent Systems*, 2022
- [131] Livia Lestingi, Davide Zerla, Marcello M Bersani, and Matteo Rossi. Specification, stochastic modeling and analysis of interactive service robotic applications. *Robotics and Autonomous Systems*, page 104387, 2023
- [21] Marcello M. Bersani, Matteo Camilli, Livia Lestingi, Raffaela Mirandola, Matteo Rossi, and Patrizia Scandurra. Towards better trust in human-machine teaming through explainable dependability. In *Intl. Conf. on Software Architecture Companion*, pages 86–90, 2023
- [20] Marcello M Bersani, Matteo Camilli, Livia Lestingi, Raffaela Mirandola, and Matteo Rossi. Explainable human-machine teaming using model checking and interpretable machine learning. In *Accepted for presentation at Intl. Conf. on Formal Methods in Software Engineering*, 2023

1.5 Structure of the Thesis

The thesis structure is described in the following.

The introductory part contains chapters:

- Chapter 2 surveying state-of-the-art related works, highlighting the gaps addressed by this research;
- Chapter 3 illustrating the preliminary theoretical concepts underlying the work and overviews pre-existing software tools that have been employed;
- Chapter 4 introducing in detail the model-driven framework's workflow;

Part I, focusing on the design-time analysis phase, contains:

- Chapter 5 presenting the scenario configuration task and the DSL;
- Chapter 6 presenting the developed formal models, whose experimental validation is presented in Chapter 7;
- Chapter 8 focusing on the developed erroneous human behavior formal model, with dedicated experiments presented in Chapter 9.

Part II, focusing on the deployment phase, contains:

- Chapter 10 introducing the deployment approach, with dedicated validation in Chapter 11;

Part III, focusing on the model adjustment phase, contains:

- Chapter 12 introducing the $\mathsf{L}^*_{\mathrm{SHA}}$ algorithm for automata learning;
- Chapter 13 presenting the experimental validation of $\mathsf{L}^*_{\mathrm{SHA}}$ on case studies independent of the human-robot interaction domain, highlighting the system-agnostic nature of the algorithm;
- Chapter 14 introducing how learned automata modeling human behavior are extended to be compatible with the formal model;
- Chapter 15 presenting the experimental validation of $\mathsf{L}^*_{\mathrm{SHA}}$ focusing on human behavior learning;

Part IV contains:

- Chapter 16 presenting the experimental validation of the overall framework on illustrative scenarios from the healthcare setting.

Finally:

- Chapter 17 concludes and illustrates future research direction building upon the hereby presented foundation.

CHAPTER 2

Related Work

This chapter surveys existing works on software development for robotics, mainly focusing on applications where human-robot interaction is predominant or that exploit formal methods throughout the pipeline.

To further aid the reader through the survey, sections focus on the different framework's phases following the workflow drafted in Chapter 1: scenario configuration and formal modeling, followed by a focus on erroneous human behavior modeling, scenario deployment, automata learning, and human behavior learning. For each phase, we highlight the identified gap and how the contributions of this thesis address it.

2.1 HRI Formal Modeling and Verification

Developing software for the robotic domain is an elaborate process given the complexity, and the unstructured nature of the system itself [77]. Therefore, it usually requires a combination of different software development techniques to achieve a satisfactory result. Several works focus on tasks such as testing and simulation [1], or implementation [4], which are substantial to the development process but out of the scope of this review. In

the following, we focus on the early design phase and report on works exploiting formal methods.

Existing works can be classified based on the *formalism* used to model the environment and the agents' behavior and the *verification* technique applied to check properties.

2.1.1 Temporal Logic-based Robotic Applications Modeling

As for the first criterion, temporal logic notations are often adopted to model the robotic task.

Gainer et al. [74] present the CRutoN tool to analyze a personal robot's behavior in a domestic setting. The work models the robot's behavior as a set of logic constraints, automatically parsed and converted into a NuSMV model [43]. The generated model is put through model checking to verify relevant properties about the system, e.g., that the robot never fails to alert the user about an event that requires their attention.

Webster et al. [215] had previously exploited the *BrahmsToPromela* tool [203] for the same case study. The human users and the robot are modeled using Brahms as *agents*. Brahms models are then automatically translated into Promela and verified through the SPIN model checker. Both works treat human behavior as a black box whose actions are selected non-deterministically from a pre-determined set.

Vicentini et al. [213] introduce an innovative risk assessment procedure for collaborative industrial tasks based on the TRIO temporal logic language [72]. Similarly to previous examples, the authors model the agents and the task through a set of logic formulae to find safety hazards and assess their severity.

As previously mentioned, human-robot interaction introduces uncertainties into the model; thus, the work has been subsequently extended to include manifestations of erroneous human behavior [10].

2.1.2 State-based Robotic Applications Modeling

State-based formalisms are also a popular choice to model the behavior of robotic systems. Most works pair the state-based model of the system with a set of logic properties to perform verification.

Ding et al. [52] exploit Finite State Machines to model collaborative industrial tasks, later extended to cover multi-robot multi-human tasks [53], where unexpected events due to the presence of humans are modeled as exceptions and paired with a recovery strategy.

Porfirio et al. [175] explore how formal verification can ensure that robots adhere to *social* norms while interacting with humans. Norms expressed as LTL formulae constitute the properties to be verified, whereas interaction sequences are modeled as a composition of Labelled Transition Systems (LTSs).

The work by Adam et al. [2] also targets the social robotics field, as the authors propose the CAIO framework. The authors exploit the Belief Desire Intention (BDI) architecture and models of human cognition to develop a perception and deliberation process that drives the robot towards making decisions in a human-like fashion and making human-robot interaction feel more natural.

Quottrup et al. [178] model multi-robot systems as a network of Timed Automata and verify whether collisions potentially occur or some robots are not able to complete their goal, which is all expressed as CTL properties and verified through Uppaal.

Zhou et al. [228] propose a similar approach based on Timed Automata and MITL properties focused on motion planning to synthesize optimal trajectories based on verification results.

Some works have also exploited Hybrid Automata to incorporate physical laws into the verification process. Molnar et al. [159] introduce the concept of Model Composition Agents (MCA), which encapsulate a Hybrid Automaton modeling either an agent or the environment and its interaction with other automata in the system. The resulting network of MCA is abstracted as an LTS, and model checked to diagnose faults in the original system.

2.1.3 Formalizations of Human Behavior

As human-robot interaction becomes a critical element of modern robotic systems, particular attention has been given to how unpredictability due to the presence of humans can be formally modeled. In this aspect, two main research directions emerge from the literature: game-based approaches and probabilistic models.

The possibility to model the interaction between a robotic agent and the environment as a *game* to synthesize a robot controller strategy (if it exists) is investigated in [119]. Kress et al. emphasize the challenge of finding a proper abstraction of the environment model that allows for significant verification results without leading to state space explosion.

Chen et al. [38] apply the approach based on Timed Game Automata (TGA) and LTL to surveillance, monitoring, and delivery tasks in partially

unknown environments. The work by Bersani et al. [22] addresses applications involving robots and humans working in a shared environment, modeled as TGA networks. Humans are modeled as *uncontrollable* agents to capture the uncertainty of their behavior. A robot controller that accounts for unpredictable human moves is then automatically synthesized through the Uppaal-TIGA tool.

On the other hand, probabilistic models of human behavior and decision-making (e.g., the *Boltzmann policy* [16]) are well-established in the literature and have been successfully applied to the robotic domain.

Mason et al. [151] exploit Markov Decision Processes (MDPs) to model an assistive-living scenario and verify probabilistic properties (expressed in PCTL logic) through the PRISM model checker [121].

The work by Junges et al. [108] combines the two approaches since it models the robot as a stochastic *controllable* agent and the human as stochastic and *uncontrollable*, which, when combined, produce a stochastic two-player game. In this case, optimal robot policies are also synthesized through PRISM-Games [37].

Vibekananda et al. [55] exploit Probabilistic State Machines to perform human pose estimation and predict their intention while interacting with a robot.

Galin et al. [75] build upon a previous study on how Cellular Automata with probabilistic transitions can be used to model human motion in partially unknown environments [28]. The authors exploit these theoretical results to develop the model of a shared workspace where humans and robots work simultaneously to compute the area where their trajectories are more likely to overlap.

2.1.4 Verification Techniques and Tools

Since state-based formalisms and temporal logics are the most popular choices when it comes to modeling the robotic system, it follows that model checking is the natural choice in terms of verification technique [143], given the availability of powerful model checkers such as Uppaal [123] and SPIN [95].

Models based on MDPs, such as the one developed by Ye et al. [220], can be verified through Probabilistic Model Checking, which is most often performed through PRISM [121].

Statistical Model Checking (SMC), the verification technique used in our framework, has also gained momentum over the last few years. The most common motivation pertains to the reduced verification times, which

Table 2.1: *HRI formal modeling survey summary. Works are categorized based on the underlying formalism, the employed verification technique, the approach adopted to model human behavior (if any), and the specific application domain within the robotics area (if any). The framework presented in this thesis is listed first for comparison.*

Ref.	Formalism Employed	Verification Technique	Human Modeling Approach	Application Domain
*	Stochastic Hybrid Automata	Statistical Model Checking	Probabilistic	Service Robotics
[74]	Linear Temporal Logic	Model Checking	None	Domestic Assistance
[215]	Linear Temporal Logic	Model Checking	Non-deterministic	Domestic Assistance
[10, 213]	TRIO Logic	Satisfiability Checking	Non-deterministic	Manufacturing
[52, 53]	Finite State Automata	Simulation	None	Manufacturing
[175]	Labelled Transition Systems	Model Checking	Non-deterministic	Social Robotics
[2]	Belief Desire Intention Agents	Simulation	Non-deterministic	Social Robotics
[178]	Timed Automata	Model Checking	None	Multi-robot Systems
[228]	Timed Automata	Model Checking	None	General-purpose
[159]	Hybrid Automata	Model Checking	None	General-purpose
[119]	Kripke Structure	Controller Synthesis	Non-deterministic	General-purpose
[38]	Timed Game Automata	Controller Synthesis	Non-deterministic	General-purpose
[22]	Timed Game Automata	Model Checking	Non-deterministic	Service Robotics
[151]	Markov Decision Processes	Probabilistic Model Checking	Probabilistic	Domestic Assistance
[108]	Markov Decision Processes	Probabilistic Model Checking	Probabilistic	General-purpose
[55]	Probabilistic State Machines	Simulation	Probabilistic	General-purpose
[75]	Cellular Automata	Simulation	Probabilistic	General-purpose

lead to more practical approaches.

Paigwar et al. [168] exploit SMC to estimate the probability of collisions in automated driving systems.

Foughali et al. [67] apply SMC to formally verify robotic software's real-time properties, like schedulability and readiness.

Herd et al. [88] focus on multi-agent systems and swarm robotics in particular. In this case, SMC dampens issues related to the size of the problem, which cannot be handled by traditional model checking techniques.

Discussion

This survey (summarized in Table 2.1) shows that numerous approaches exploit formal methods to analyze robotic applications. Specifically, several attempts have been made at formalizing the aspects of human behavior that are significant while interacting with a robot and should, thus, impact the results of the formal analysis. Most of these works present either deterministic, game-based, or probabilistic approaches, such as the hereby presented

framework.

Deterministic approaches—such as architectures based on BDI agents—potentially result in less complex models and more favorable verification times. However, assuming complete rationality and absence of fuzziness is reasonable for robotic agents (the orchestrator SHA indeed inherits most of its substructures from the BDI architecture) or for human agents performing small repetitive tasks in controlled environments [2, 8]. Human-robot interactions in the service sector feature virtually no constraint on human behavior, thus deterministic models are overly restrictive. Furthermore, service robot applications involve people from various age groups with different characteristics and performing a broad range of tasks. Therefore, while estimating a scenario's outcome, exploring the state space of all possible behaviors should be guided by such features.

Game-based approaches, although effective when exploited for controller synthesis [22], imply an exhaustive exploration of human actions (i.e., the *opponent*'s move) irrespective of their likelihood given the specific scenario configuration.

For these reasons, probabilistic approaches are particularly suited for the purpose of this framework. Specifically, to the best of the authors' knowledge, this is the first attempt at combining probabilistic weights on human actions with a hybrid and stochastic characterization of physiological processes. Due to its complexity, the resulting model is more practically manageable through SMC rather than probabilistic model checking. Indeed, works exploiting exhaustive techniques such as [216] focus on smaller setups targeting a specific task (e.g., the handover of an item). Despite the loss in reliability introduced by SMC that only relies on a finite set of runs of the systems, the proposed framework is applicable to a broad range of scenarios (as shown by the coverage analysis results) while still providing results at design time that accurately reflect runtime observations.

2.2 Robotic Domain-Specific Languages

In 2014, Nordmann et al. [163] surveyed 137 papers presenting robotic DSLs. At the time of writing, Scopus indexes approximately 70 papers published since 2014 with keywords `robot*` and `dsl`. These numbers show that DSL development is a cornerstone of the robotic software engineering process since it automates the generation of code or complex models and makes development frameworks accessible to a broader audience. Referring to the classification in [163], in the following, we report on the subset of works on this topic dealing with the *scenario building* phase, i.e., DSLs

to specify high-level environment features and the robot's task, as these are the closest to our work.

2.2.1 DSLs for Scenario Building.

Noreils and Chatila [164] present a high-level notation to specify reactive robotic mission plans. The language envisages the specification of modules, which are further structured into three architectural layers: the *functional* layer to specify the lower-level robot's capabilities, the *planning* layer to specify task sequences, and the *control* layer that translates plans into requests to the functional modules.

Knoop et al. [117] present an approach to automatically generate robotic tasks starting from representations of tasks in the human operational space, adhering to the Programming by Demonstration paradigm.

Finucane et al. [63] present the LTLMoP framework to automatically synthesize and deploy robot controllers. The framework converts Structured English specifications describing the robotic task into equivalent LTL formulae, which are then synthesized into an automaton (the *discrete* controller). The work has been subsequently extended by Raman et al. [180] with implicit memory strategies to model robotic tasks depending on events that occurred in the past (e.g., "every time you sense *order*, visit the *kitchen*").

Kunze et al. [120] present SRDL, a framework extending the KnowRob knowledge base [209] with notions about *robots*, hardware *components*, *actions*, and *capabilities* (of performing a certain action).

Miyazawa et al. [158] introduce RoboChart, a DSL to model and verify real-time concurrent robotic tasks with budgets and deadlines (i.e., cost and time constraints). RoboChart semantics, based on Timed Automata and Timed Communicating Sequential Processes (CSP) [193], make the notation amenable to formal verification, specifically model checking.

Ciccozzi et al. [42] propose a family of three languages to specify missions for multi-robot systems: the Monitoring Mission Language to specify task sequences, the Robot Language to configure the individual robots, and the Behavior Language to specify the atomic movements of robots.

2.2.2 DSLs for Human-Robot Interaction

Over the last few years, the advent of human-robot interaction and collaborative robotics has shifted the focus of DSL development. Recent works introduce semantic entities to describe human actions and robots' reactions to human-related events.

Table 2.2: *Robotic Domain-Specific Languages survey summary. Works are categorized based on the targeted architectural level [144], whether DSL models can be subject to verification through an automated procedure, whether the language supports interactive applications, and the specific application domain within the robotics area. The framework presented in this thesis is listed first for comparison.*

Ref.	Architectural Level	Automated Verification	Human-Robot Interaction Support	Application Domain
*	Task Plot	✓	✓	Service Robotics
[164]	Function/Hardware	×	×	Mobile Robots
[117]	Service	×	×	General-purpose
[63, 180]	Skill	✓	×	Domestic Assistance
[120]	Hardware/Service	×	×	General-purpose
[158]	Skill	✓	×	General-purpose
[42]	Mission/Service/Hardware	×	×	Multi-Robot Systems
[8]	Service	×	✓	Human-Robot Interaction
[50]	Service	×	✓	Warehouse Logistics
[65]	Service	×	✓	Rehabilitation Robotics

Araiza-Illan et al. [8] exploit the AgentSpeak language [181] to implement BDI agents and automatically generate test cases for interactive robotic applications. The framework is tested on a cooperative table assembly case study. The robot's BDI agent infers the human's state based on three sensors and reacts accordingly as encoded by the AgentSpeak model.

Detzner et al. [50] present LoTLan, a domain-specific language to describe warehouse material flow processes. The work consists of a procedure to map human vocal requests (e.g., "I need an item") to common semantics, identifying *who* has to perform *which action*, and finally, LoTLan primitives, which are then converted into plans for AGVs.

Forbig et al. [65] exploit their language CoTaL [66] to model interactive tasks between a humanoid robot and a stroke patient performing arm mobility recovery exercises. The resulting specification captures all phases needed for the exercise session, how the humanoid robot can detect whether the patient has completed an exercise, and how to react accordingly.

Discussion

This section shows that the literature is rich with DSLs for the robotic domain, but proposals targeting interactive applications are lacking (as shown in Table 2.2). Specifically, existing works target the manufacturing sector [50, 79] or very specific tasks from the healthcare setting [65]. In contrast, the service sector calls for more general-purpose primitives to define how robots and humans interact. Other works propose a high-level specification of mission patterns for multi-robot teams in environments (pos-

sibly) populated by humans [76, 155], but this has not been attempted for applications where humans are actively involved as in the domain of this framework.

2.3 Erroneous Human Behavior Modeling

Modeling human behavior is a long-standing issue in human-automation interaction analysis. With the advent of collaborative robotics, the problem has recently started to attract attention in its declination to human-*robot* interaction.

Unforeseen human actions, especially those originating from errors, hugely impact the design of general human-machine interaction, especially with robots [10, 49]. In the field of human-robot interaction, most works investigate human errors as sources of safety *hazard* (e.g., leading to a collision with the robot), which is the core issue tackled by Human Reliability Analysis [54, 96] and probabilistic risk assessment techniques [93, 217].

Given the complexity of the human mind and the human decision-making process, a perfectly accurate, all-encompassing model of human behavior is not feasible. However, existing works, mostly from research on human cognition, propose mathematical models of human behavior within specific boundaries, for example, limited to decision-making in the workplace. These models fall into three main categories [9]:

1) cognitive models investigate the mental process leading to a certain decision;

2) task-analytic models capture human behavior as a hierarchy of actions;

3) probabilistic models refine the non-determinism of human behavior through probability distributions over actions.

Well-established cognitive models are Soar [122] and Adaptive Character of Thought (ACT-R) [6]. However, cognitive sources behind human behavior are out of the scope of our model-driven framework; therefore, mental models are not further investigated.

Task-analytic models such as ConcurTaskTrees (CTT) [170], although recently expanded with a taxonomy of human errors [26], suffer from the drawback of being intrinsically case study-specific, thus hardly reusable.

Probabilistic models are considered highly beneficial in designing cyber-physical systems where human factors are critical [48]. Some examples of probabilistic models are Boltzmann rationality [16], the LESS model [24],

and Markovian models such as Partially Observable Markov Decision Processes (POMDPs) [192], and Bayesian Networks [210].

The main issue of probabilistic models is the lack of extensive and reusable datasets to train reliable probability distributions [51]. However, although our work does employ a probabilistic model of human behavior (i.e., SHA), it partially works around this issue by performing design-time analysis as a *function* of probabilistic parameters. Nevertheless, collecting actual observations of human behavior while participating in the analyzed scenarios would still be necessary to provide compelling evidence of the formal model's accuracy.

Previous works propose a formalization of erroneous human behavior models for formal verification. Cerone et al. [36] propose a taxonomy of operator errors in human-computer interaction formalized through the CSP process algebra and temporal logic. Shin et al. [197] present a formal model of human material handlers in manufacturing systems, depending on human tasks and errors modeled through part-state graphs. Rukšėnas et al. [190] present a verification framework for interactive systems with cognitive models of human errors under timing constraints based on the Goals, Operators, Methods, and Selection (GOMS) methodology [107]. Askarpour et al. [10] present an automated risk assessment technique for collaborative robotic applications. The eight phenotypes identified by Hollnagel are expressed through logic formulae and verified through the Zot formal verification tool.

Discussion

The mentioned works are summarized in Table 2.3. The service sector has different demands than the industrial settings, especially in healthcare, since it is characterized by a higher degree of human task diversity and more significant sources of uncertainty [145, 204]. Therefore, given the different target domains and underlying formalisms, the results obtained in [10] cannot be directly embedded into our framework. However, the observations in [10] about the efficacy of Hollnagel's phenotypes constitute the foundation for the work presented in this thesis, which adapts phenotypes to behaviors observed in service settings and integrates them with a stochastic characterization.

2.4 Robotic Applications Deployment

One of the contributions of this thesis is a model-to-code mapping technique to translate a Stochastic Hybrid Automata network into a robotic de-

Table 2.3: *Erroneous Human Behavior Modeling survey summary. Works are categorized based on the modeling approach category (i.e., either cognitive, task-analytic, or probabilistic) and the underlying formalism (if any). The framework presented in this thesis is listed first for comparison.*

Ref.	Cognitive	Task-analytic	Probabilistic	Formalism
*	×	×	✓	Stochastic Hybrid Automata
[122]	✓	×	×	None
[6]	✓	×	×	None
[26, 170]	×	✓	×	ConcurTaskTrees
[48]	×	×	✓	None
[16]	×	×	✓	None
[24]	×	×	✓	None
[192]	×	×	✓	Markov Decision Processes
[210]	×	×	✓	Bayesian Networks
[36]	×	✓	×	CSP Process Algebra
[197]	×	✓	×	Deterministic Finite Automata
[190]	✓	×	×	SAL Language
[10]	×	✓	×	TRIO Logic

ployment framework.

Several works in literature have a similar goal, though often targeting a different formalism or a different phase of the software development process. In some cases, testing rather than deployment is the primary purpose, as TA can be exploited to automatically generate offline and online test cases (e.g., for ROS packages [58]).

Previous attempts are also at porting a formal model to a simulation environment. One example is the TestIt framework, which generates a simulator-agnostic simulation environment for multi-agent robotic applications starting from Timed Automata [111].

Other works focus on code analysis and explore the possibility of highlighting potential flaws in a ROS-based infrastructure through model checking [86]. Another notable example is the work by Wang et al. [214], presenting a model-driven framework to convert a TA network into C++ code, which is tested on a robot grasping task.

This survey suggests that developing code generation techniques starting from formal models is valuable to the robotic field. The approach allows for the testing and deployment of robotic applications whose properties (for example, concerning safety or efficiency) have been formally verified.

On the other hand, the assistive robotics domain, to which this work is tailored, calls for a sound mathematical formulation of human physiological and behavioral properties to be involved in the formal verification pro-

Table 2.4: *Robotic Application Deployment survey summary. Works are categorized based on whether they include a code generation mechanism, the starting formalism (if any), the target programming language (if any), the employed 3D simulator (if any), and whether the approach supports interactive applications. The framework presented in this thesis is listed first for comparison.*

Ref.	Code Generation	Starting Formalism	Target Programming Language	3D Simulator	Human-Robot Interaction Support
*	✓	Stochastic Hybrid Automata	Python / LUA	CoppeliaSim	✓
[58]	✓	Timed Automata	Python	Morse	✓
[111]	✓	Timed Automata	C++ / Python	Any (if compatible with ROS)	×
[86]	✓	Timed Automata	C++	None	×
[214]	✓	Timed Automata	C++	Gazebo	×
[147]	✓	Synchronous Emulation Automata	C	Piha	×
[56]	✓	Hybrid Automata	C	Modelica	×
[15]	✓	Hybrid Automata	Simulink/Stateflow	Simulink	×
[227]	×	-	-	-	✓
[116]	✓	None	Custom Scripts	Custom	✓
[177]	×	-	-	-	✓

cess. More complex time-dynamics mean that Timed Automata—including the non-deterministic or probabilistic extensions—or temporal logic-based notations [213] no longer suffice as they require a *hybrid* formalism.

Pre-existing works that introduce translation principles of Hybrid Automata into executable code do not suit robotic applications. In some cases, modular architectures with parallel components are the target [147], whereas interactive robotic applications require a hierarchical deployment structure with an intermediary middleware layer.

As previously noted, some works target testing and code coverage rather than deployment [56] or model-to-model transformations, such as HA to Simulink/Stateflow (SLSF) diagrams [15]. To the best of the authors' knowledge, the hereby presented deployment approach is the first to introduce a mapping principle between a HA network and a software architecture compatible with a ROS-based robotic system.

The research line on verification and simulation techniques for analyzing human-robot interaction is also worthy of discussion. Simulation can be used to estimate the final level of satisfaction of the human *customer* after short-term human-robot interactions [227].

Some works focus on the planning phase of the robotic mission, for example, analyzing alternative workflows for the task driven by *human input* [116].

Quintas et al. [177] investigate how an agent's performance is affected by interaction workflows in its decision-making process. The framework

processes a high-level description of the scenario and the human interacting with the robot to generate a mission plan *graph*.

The inclusion of a human operator model is also of paramount importance in the Virtual Commissioning (VC) of collaborative manufacturing tasks, whose purpose is to test in advance the system's reaction to malfunctions [189]. Webster et al. [216] also argue that different verification and validation (V&V) techniques are not fully exhaustive when they are used alone but should be combined into a *corroborative* approach to considerably increase their effectiveness.

Discussion

The survey on deployment approaches and code generation techniques for robotic applications is summarized in Table 2.4. The survey shows how an approach based on formal methods like the one presented in this thesis can significantly benefit the software development lifecycle regarding dependability. The vast majority of industry professionals who have employed formal verification techniques report a quality boost for the final product [218], and demand for this approach is rapidly growing [82].

This approach can prove especially beneficial to cyber-physical systems, where robots interact with the environment and humans [78]. Therefore, software running on robots in charge of decision-making must result from a trustworthy verification process. The deployment approach presented in this thesis paves the way for a dependable code generation technique retaining the features verified while designing the robotic application.

2.5 Automata Learning

Throughout the chapter, we surveyed several valid modeling techniques for human-robot interaction that are amenable to formal verification. Nevertheless, whenever a complex Cyber-Physical System (CPS) such as HRI scenarios is involved, a permanent CPS model may not be efficient or reasonable. Unexpected behaviors may arise once software components are deployed, or known behaviors may evolve in time (e.g., due to the decay of physical elements). Data-driven learning techniques are necessary to have the model *evolve* based on field observations. Specifically, since the formalism underlying this work is automata-based, we focus on automata learning techniques.

The research line on automata learning has been active for the past four decades as it is an essential tool to tackle the analysis of complex systems

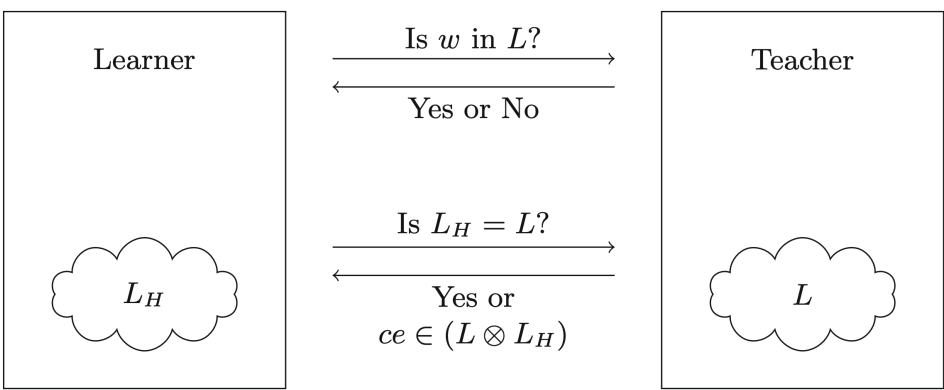


Figure 2.1: *Schematic representation from [97] of the interaction between the learner and the teacher in* L^**, where* $\otimes$ *indicates the symmetric difference operator.*

subject to uncertainty [97]. Automata learning algorithms are mainly classified according to two criteria [146]:

1. *active* or *passive*: active algorithms can request new observations of the system under learning whenever this is found necessary and usually rely on a teacher/learner pattern; passive algorithms solely rely on a collected set of samples;

2. *online* or *offline*: online algorithms can process a collected observation only once, while offline algorithms store observations, thus allowing for repeated processing.

In the following survey, we focus on active online algorithms as they best suit real-time critical CPSs.

The earliest works introduce algorithms to learn Deterministic Finite-state Automata (DFA). Most notably, given a finite alphabet Σ, the active online learning algorithm L^* infers a DFA for a specific language L [7] through the interaction between a *learner* and a *teacher*, also represented in Fig. 2.1. The learner stores the conjecture DFA as an *observation table* and refines it by submitting *queries* to the teacher, which possesses exhaustive knowledge about the language under learning. In L^*, there are two types of queries: *membership* queries and *equivalence* queries. The learner submits membership queries to inquire whether a specific word w is accepted by the language under learning known by the teacher. The learner then submits an equivalence query when a conjecture DFA (i.e., accepting conjecture language L_H) is ready. If the teacher approves the conjecture (i.e., if equivalence $L_H = L$ holds), L^* terminates; otherwise, the teacher

returns a counterexample word $ce \in (L \otimes L_H)$, i.e., a word that the conjecture should accept but it does not or vice-versa.

L^* has been subsequently extended to cover Mealy machines [150], Timed Automata [87], and Petri Nets [59]. The TTT algorithm adopts a similar framework introducing redundancy-free data structures [102].

A wide range of automata learning algorithms (including L^* and TTT) targeting DFA, Mealy machines, and Visibly Pushdown Automata (VPDA) are implemented in the open-source framework LearnLib [179]. LearnLib is a C++ library for automata learning developed to—among other goals—experiment with and compare different learning techniques for finite-state models of real-world systems. It is constituted of three modules:

- the algorithm, for which a range of both active (e.g., L^*, TTT [102], and DHC [157]) and passive algorithms (e.g., RPNI [166]) is implemented;
- the filter, i.e., a strategy to reduce the number of queries;
- the approximative equivalence queries module, exploiting the generation of test suites for the submitted conjectures.

However, deterministic formalisms do not adequately capture the system's behavior when the CPS under analysis exhibits unpredictable behavior (especially with humans involved). To this end, learning algorithms for probabilistic automata have also received a great deal of attention.

Carrasco and Oncina introduced the *ALERGIA* algorithm for Stochastic Finite-state Automata learning [34], later extended by Mao et al. [148].

Some works adopt a frequentist approach, such as the work by Ghezzi et al., where probability weights on transitions are Bayesian estimators extrapolated from traces [80].

Tappler et al. developed an extension of L^* to Markov Decision Processes [207] called $\mathsf{L}^*_{\mathrm{MDP}}$. The authors develop two versions of $\mathsf{L}^*_{\mathrm{MDP}}$, one assuming the existence of an omniscient teacher like in L^* and, most notably, a sampling-based version. The latter lifts the assumption on the availability of an omniscient teacher (which is impractical with real systems), replacing it with a sampling-based interface between the learner and the teacher. The teacher's answer to queries is based on the available samples, and, when necessary, a specific query actively requests new observations (thus $\mathsf{L}^*_{\mathrm{MDP}}$ is an active algorithm) to improve the accuracy of estimations for rarely observed samples.

Several works target HA for systems with complex time dynamics. Medhat et al. present a framework to infer HA from input/output traces based

Table 2.5: *Active Automata Learning survey summary. Works are categorized based on the target learned formalism and whether they imply the existence of an omniscient oracle. The framework presented in this thesis is listed first for comparison.*

Ref.	Algorithm	Target Formalism	Omniscient Oracle
*	$\mathsf{L}^*_{\mathrm{SHA}}$	Stochastic Hybrid Automata	×
[7]	L^*	Deterministic Finite-state Automata	✓
[150]	$\mathsf{L}^*_{\mathrm{Mealy}}$	Mealy Machines	✓
[87]	-	Timed Automata	✓
[59]	-	Petri Nets	✓
[102]	TTT	Deterministic Finite-state Automata	✓
[157]	DHC	Mealy Machines	✓
[34, 148]	ALERGIA	Stochastic Finite-state Automata	✓
[80]	BEAR	Stochastic Finite-state Automata	×
[207]	$\mathsf{L}^*_{\mathrm{MDP}}$	Markov Decision Processes	✓ / × (sampling-based version)
[154]	-	Hybrid Automata	×
[191]	POSEHAD	Hybrid Automata	×
[200]	-	Hybrid Automata	×
[219]	-	Hybrid Automata	×

on sampled signal clustering techniques [154].

Saberi et al. present a similar algorithm exploiting Dynamic Time Warping to cluster comparable signal segments [191].

The work by Soto et al. [200] introduces an online learning algorithm for Linear HA based on membership queries that verify whether a data set is a plausible realization of a hypothesis automaton within a tolerance threshold.

Yang et al. [219] present a framework to infer HA, including linear inequalities guard conditions, exploiting Linear Matrix Inequality to cluster signals with comparable dynamics and Prefix Tree Acceptor to merge modes.

Discussion

Although the selected works (summarized in Table 2.5) show promising results when validated against selected case studies, they are limited to *deterministic* models. However, assuming that complex CPSs such as interactive robotic scenarios behave fully deterministically is not always feasible.

Modeling complex CPSs with a non-deterministic behavior requires both

hybrid components to capture continuous-time dynamics and stochastic features to capture uncertainty, as argued in Section 2.1. However, SHA, which captures some non-deterministic behaviors, are not covered by existing learning algorithms. The presented research addresses this gap by extending the L* framework, whose soundness and robustness to extensions have been repeatedly proven [97].

2.6 Human Behavior Prediction

Predicting human behavior is essential when assessing or deploying systems that require human-machine interaction. Numerous solutions have been presented over the years. For example, Brown et al. [29], and Rudenko et al. [188] surveyed more than 200 prediction algorithms for human driver behavior and human motion, respectively.

Given the complexity of the problem, we bound the analysis to the aspects most relevant to the specific application. Our scenarios involve humans coordinating with mobile robots for tasks that require moving inside a building (e.g., walking and running).

The entry point of our framework subsumes that humans and robots have already established the task to carry out jointly: therefore, the social aspect of the interaction (i.e., dialogues or gestures) is out of scope for this work. Finally, given the high-level perspective of the work, human modeling focuses on the decision-making process rather than lower-level aspects such as motion planning: the latter are still relevant for the selected scenarios, but they are also out of scope.

Given these premises, existing works that fall within the stated boundaries can be classified into two categories based on the underlying techniques: *expert-driven* and *data-driven* [187] (also summarized in Table 2.6).

Expert-driven approaches require an explicit mathematical formulation of the system under scrutiny. Within this category falls the *expected utility maximization* paradigm [109], assuming that agents behave to maximize the utility obtained from a specific action. However, it is found to be hardly applicable in reality. Further examples of expert-driven approaches are *quantal response* [153], assuming that humans maximize the expected utility although with a noisily estimated strategy, and *level-k* [202], assuming that humans perform k bounded iterations of reasoning.

As for approaches based on human cognition, there exists a dichotomy between heuristic-based and knowledge-based models [94]. Heuristic models have a more straightforward structure but lack accuracy, though there

are some successful examples [149, 167].

On the other hand, knowledge-based models can be very complex and are mostly limited to application fields where task-specific prior knowledge is available [27, 226].

Markov Decision Processes (MDPs) are particularly suitable for modeling uncertainty as a probabilistic phenomenon and are closely related to this work's formalism. Adomi et al. [3] exploit Hidden Markov Models to predict the actions of subjects exploring a maze-like environment. Similarly, Carr et al. [33] use MDPs to predict movements inside a grid-like world, whereas McGhan et al. [152] employ MDPs to predict human intent in a Space Station.

Data-driven approaches aim at converting field data into a decision-maker to approximate the outcome of decision-making rather than *reverse-engineering* the process. Reinforcement learning is adopted in several application fields, including interaction with medical robots [137], simulated driving [101], and pedestrian navigation [61].

Various works propose a human behavior prediction technique based on deep learning. Phan et al. focus on the health domain and data collected through mobile devices [173], whereas Lu et al. [142] and Jaouedi et al. [105] both exploit visual data to predict human actions in sports-related activities.

The model of human behavior in our framework is tied, at its core, to physical fatigue. Several works exploit existing fatigue models to assess the ergonomics of tasks that require collaboration between human operators and robotic arms. The purpose is to plan tasks that preserve human health in the long run by minimizing fatigue.

Zhang and Li [225] propose a model for auction-based task allocation in human-robot teams that accounts for human fatigue and predicts human performance at run-time. Peternel et al. [172] present a method for the robot to adapt its behavior in real-time as a function of human fatigue. The approach is validated on collaborative tasks involving an industrial manipulator and a human worker.

The impact of human physical strain on collaborative manipulation tasks has also been explored by Zanchettin et al. [223]. The authors present a motion control strategy so that the robot adapts to the most ergonomic posture for the human operator to minimize fatigue. Li et al. [136] also focus on industrial collaborative tasks, specifically how human efficiency decreases due to fatigue during disassembly tasks. Heydaryan et al. [89] include human fatigue in the set of criteria agreed upon by a team of experts necessary to evaluate the efficiency of their proposed decision-making process

Table 2.6: *Human Behavior Prediction techniques survey summary. Works are categorized based on whether they are expert-driven or data-driven, the employed prediction technique, the employed formalism (if any), and the specific application domain (if any). The framework presented in this thesis is listed first for comparison.*

Ref.	Expert Driven	Data Driven	Prediction Technique	Modeling Formalism	Application Domain
*	✓	×	Statistical Model Checking	Stochastic Hybrid Automata	Service Robotics
[109]	✓	×	Numerical Solution	Analytical Model	Behavioral Economics
[153]	✓	×	Numerical Solution	Normal Form Games	Game Theory
[202]	✓	×	Numerical Solution	Normal Form Games	Game Theory
[149]	✓	×	Numerical Solution	Heuristic Model	General-purpose
[167]	✓	×	Simulation	ACT-R	General-purpose
[226]	✓	×	Simulation	Ontology Model	Service Robotics
[3]	✓	×	Bayesian Estimation	Markov Decision Processes	Exploration Tasks
[33]	✓	×	Probabilistic Model Checking	Markov Decision Processes	Motion Planning
[152]	✓	×	Optimization	Markov Decision Processes	Human-Robot Interaction
[137]	×	✓	Reinforcement Learning	Markov Decision Processes	Medical Robotics
[101]	×	✓	Reinforcement Learning	Markov Decision Processes	Driving Simulation
[61]	×	✓	Reinforcement Learning	Deep Neural Network	Navigation Tasks
[173]	×	✓	Deep Learning	Restricted Boltzmann Machine	Social Networks
[142]	×	✓	Deep Learning	Convolutional Neural Network	Motion Tracking
[105]	×	✓	Deep Learning	Gated Recurrent Neural Network	Motion Tracking

for human-robot teams.

There are also works that incorporate human fatigue into discrete-event system models. Hu and Chen [98] propose a modeling framework based on Petri nets for manufacturing processes, including a continuous-time Markov Decision Process modeling human fatigue.

Discussion

Formal verification intrinsically requires a rigorous mathematical model, which makes a purely data-driven approach impracticable for this work. Therefore, although the learning process relies on field-collected data, the $\mathsf{L}^*_{\mathrm{SHA}}$ output can be classified as *expert-driven* and plugged into the formal model, although it exploits field-collected data to infer critical features of automata modeling the human behavior.

The importance of fatigue modeling and sensing while developing robotic

applications has already been acknowledged. Nevertheless, existing works investigate the role of fatigue during task planning or as a driver for adaptation during motion planning. In contrast, this work focuses on the early design stage. Furthermore, to the best of the authors' knowledge, no existing work in the robotics area examines the fatigue phenomenon's stochastic features [100]. Fatigue rates exhibit significant variations based on the specific subject's characteristics, the muscle group involved, and the task performed. Therefore, this work also learns different fatigue profiles depending on individual variability, which may be critical to healthcare scenarios.

CHAPTER 3

Background

This chapter presents the theoretical concepts and pre-existing tools underlying the work. Firstly, a formal definition of Stochastic Hybrid Automata (i.e., the formalism employed for this work) is provided, followed by a recap of their semantics. The main features of Statistical Model Checking (i.e., the employed verification technique) are then recapped. Finally, the main tool exploited for this framework (specifically, Uppaal SMC for the formal verification phase, ROS for the middleware layer at deployment time, and CoppeliaSim for simulation) are briefly presented.

3.1 Theoretical Preliminaries

In this section, we define Stochastic Hybrid Automata (i.e., the chosen formalism), illustrate the behavior, and overview Statistical Model Checking (i.e., the chosen verification technique).

3.1.1 Stochastic Hybrid Automata

In the following, we provide a formal definition of Stochastic Hybrid Automata (SHA) and illustrate their features through a running example inspired by [46, Section 4].

Example 3.1.1. The example captures a system composed of a room with a heating system, whose model is shown in Fig. 3.1a, and the thermostat controlling its temperature, shown in Fig. 3.1b. The room temperature is the main physical phenomenon of the system, which is modeled by real variable T in the automaton in Fig. 3.1a.

The thermostat is modeled through operating states on (which makes the room warmer) and *off* (thus, letting the room temperature decrease naturally). When the thermostat is *off*, as soon as temperature T decreases below a threshold $\mathsf{T}_{\mathsf{th}_1}$, hence the condition $T \geq \mathsf{T}_{\mathsf{th}_1}$ labeling location *off* does not hold (resp., exceeds a threshold $\mathsf{T}_{\mathsf{th}_2}$, hence the condition $T \leq \mathsf{T}_{\mathsf{th}_2}$ labeling location on does not hold), the thermostat switches the heating on (resp., off). The triggering of the event, indicated with symbol `on!`, makes the thermostat modify its operating state, hence moving to on (resp., switch off the heating, hence moving to *off*).

Room temperature is modeled in three situations, also represented in Fig. 3.1a: the one for which the temperature decreases due to the absence of heating, i.e., $cool$, and two situations for which the temperature increases at different rates, i.e., $high$ and low, when the heating is on.

The temperature grows according to differential equation $\dot{T} = \theta - \frac{T}{\mathsf{R}}$ when the thermostat is on and decreases according to $\dot{T} = -\frac{T}{\mathsf{R}}$ when it is off, where R is a constant and θ is a randomly distributed parameter, i.e., whose value depends on a probability distribution.

At the onset of the system, the thermostat is off, hence the room is cooling down with the temperature conventionally initialized to $\mathsf{T}_{\mathsf{th}_1}$. This value allows the thermostat to spend a non-null amount of time in location $cool$, where constraint $T \geq \mathsf{T}_{\mathsf{th}_1}$ limits the temperature inferiorly. The update on the edge entering location $cool$ initializes T.

When event `on!` (resp., `off!`) is fired by the thermostat, the room reacts immediately, i.e., thermostat and room *synchronize*. Reacting to the event, indicated with symbol `on?` (resp., `off?`), causes the room temperature to rise and the automaton to switch to either $high$ or low (resp., the room temperature to decrease and the automaton to move to $cool$).

The room can be heated at a high or low rate (e.g., if a window is closed or open, respectively): the choice is made probabilistically when the automaton synchronizes with event `on!`. Probability weights are known and

indicated as p_{H} and p_{L}, respectively. Parameter θ in Fig. 3.1a is a realization of a Normal distribution with mean μ_{H} and standard deviation σ_{H} (indicated as $\mathcal{N}(\mu_{\mathrm{H}}, \sigma_{\mathrm{H}}^2)$) when the room is heating quickly because the window is closed (the subscript "H" stands for "high" rate of heating).

Conversely, when the room is heating slowly because the window is open, the probability distribution is $\mathcal{N}(\mu_{\mathrm{L}}, \sigma_{\mathrm{L}}^2)$ (subscript "L" stands for "low" rate of heating). Throughout the thesis, we express that a random parameter θ is a realization of random variable Θ governed by distribution $\mathcal{N}(\mu, \sigma)$ through notation $\theta \sim \mathcal{N}(\mu, \sigma)$.

The definition of SHA is given in the following [5, 47, 62]. Let W be a set of symbols; we indicate with $\Gamma(W)$ the set of conjunctions of constraints of the form $\gamma_1 \sim \gamma_2$, where $\sim$ is a relation in $\{<, =\}$ and γ_i ($i \in \{1, 2\}\}$) is an arithmetical term defined by the sum elements in W and $\mathbb{N}$ (e.g., $w_1 + w_2 + 3$, with $w_1, w_2 \in W$). We indicate with $\Xi(W)$ the set of updates on elements of W. An *update* in $\Xi(W)$ (for example, $w' = w + 2$) is a constraint where free variables are elements of W (e.g., $w \in W$) and of its primed version W' (e.g., $w' \in W'$). We indicate the set of non-negative real numbers with $\mathbb{R}_+$ and, with $\mathbb{R}^W$, the set of assignments to variables of W (i.e., *valuations*).

Definition 1. *A Stochastic Hybrid Automaton is a tuple* $\langle L, W, \mathcal{F}, \mathcal{D}, \mathcal{I}, C, \mathcal{E}, \mu, \mathcal{P}, l_{\mathrm{ini}} \rangle$*, where:*

1. *L is the set of **locations** and $l_{\mathrm{ini}} \in L$ is the initial location;*
2. *W is the set of real-valued **variables** of which clocks $X \subseteq W$, dense-counter variables $V_{\mathrm{dc}} \subseteq W$, and constants $K \subseteq W$ are subsets;*
3. *$\mathcal{F} : L \to \{(\mathbb{R}_+ \cup (\mathbb{R}_+ \times \mathbb{R})) \to \mathbb{R}^W\}$ is the function assigning a set of **flow conditions** to each location, where flow conditions are (continuous) functions with one or two parameters, which assign a valuation to every time instant in $\mathbb{R}_+$ or to every pair constituted by a time instant in $\mathbb{R}_+$ and a constant value in $\mathbb{R}$, depending on the location;*
4. *$\mathcal{D} : L \rightharpoonup \{\mathbb{R} \to [0, 1]\}$ is the* partial *function assigning a **probability distribution** from $\{\mathbb{R} \to [0, 1]\}$ to locations which feature flow conditions with two parameters;*
5. *$\mathcal{I} : L \to \Gamma(W)$ is the function assigning a (possibly empty) set of **invariants** to each location;*
6. *C is the set of **channels**, including the internal action ϵ;*

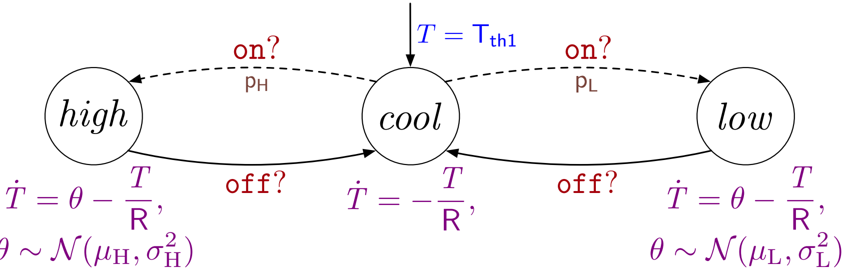
7. *$\mathcal{E} \subset L \times C_{!?} \times \Gamma(W) \times \wp(\Xi(W)) \times L$ is the set of* ***edges****, where $C_{!?} = \{\mathsf{c}! \mid \mathsf{c} \in C\} \cup \{\mathsf{c}? \mid \mathsf{c} \in C\}$ is the set of* events *involving channels in C. Given an edge $(l, c, \gamma, \xi, l') \in \mathcal{E}$, l (resp. l') is the outgoing (resp. ingoing) location, c is the* edge event*, γ is the* edge condition *and ξ is the* edge update*. For each $l \in L$, $\mathcal{E}(l) \subseteq C_{!?} \times \Gamma(W) \times \wp(\Xi(W)) \times L$ is the set of edges outgoing from l (for each $(c, \gamma, \xi, l') \in \mathcal{E}(l)$ then $(l, c, \gamma, \xi, l') \in \mathcal{E}$ and vice-versa);*

8. *$\mu : (L \times \mathbb{R}^W) \rightarrow \{\mathbb{R}_+ \rightarrow [0, 1]\}$ is the function assigning a* ***probability distribution*** *from $\{\mathbb{R}_+ \rightarrow [0, 1]\}$ to each* configuration *of the SHA, where configurations are (l, v_{var}) pairs constituted by a location $l \in L$ and a valuation $v_{\mathrm{var}} \in \mathbb{R}^W$;*

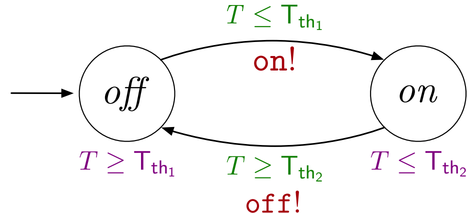
9. *$\mathcal{P} : L \rightharpoonup \{(C_{!?} \times \Gamma(W) \times \wp(\Xi(W)) \times L) \rightharpoonup [0, 1]\}$ is the* partial *function assigning a discrete* ***probability distribution*** *from $\{(C_{!?} \times \Gamma(W) \times \wp(\Xi(W)) \times L) \rightharpoonup [0, 1]\}$ to locations such that, for each $l \in L$, $\mathcal{P}(l)$ is defined if, and only if, $\mathcal{E}(l)$ is non-empty; also, the domain of the distribution is $\mathcal{E}(l)$ (hence, $\sum_{\substack{\alpha=(\mathsf{c}!, \gamma, \xi, l') \in \mathcal{E}(l) \\ \mathsf{c} \in C}} \mathcal{P}(l)(\alpha) = 1$ holds).*

In SHA, real-valued variables (i.e., a generalization of clocks) evolve in time according to generic expressions called *flow conditions* [5]. Flow conditions constraining the evolution over time of variables in W are defined through sets of Ordinary Differential Equations (ODEs). This feature makes SHA a suitable formalism to model systems with complex dynamics, as it is possible to model through flow conditions, for example, laws of physics or biochemical processes. ODEs constraining clocks (for which $\dot{x} = 1$ holds for all $x \in X$), dense-counter variables, and constants (where $\dot{v} = 0$ holds for all $v \in V_{\mathrm{dc}} \cup K$) are special cases of flow conditions.

If variable $\theta \in V_{\mathrm{dc}}$ is an independent term for flow $f \in \mathcal{F}(l)$ on location $l \in L$, i.e., $f = f(t, \theta)$, and θ is interpreted as a randomly distributed parameter, then f is a **stochastic process** [85]. We limit the analysis to flow conditions depending on *at most one* random parameter, as per Definition 1, which is enough to model human-robot interaction within the scope of our work. For example, in the SHA shown in Fig. 3.1a the room temperature is modeled by real-valued variable $T \in W$. When the temperature is decreasing, it is constrained by the flow condition $\dot{T}(t) = -T(t)/\mathrm{R}$, where function $\dot{T}(t) \in \mathcal{F}(cool)$ depends on time only and has solutions in $\mathbb{R}$. When the temperature increases, it evolves according to flow condition $\dot{T}(t, \theta) = \theta - T(t, \theta)/\mathrm{R}$, depending both on time and random parameter θ. The domain of $\dot{T}(t, \theta)$ is, thus, $\mathbb{R}_+ \times \mathbb{R}$ and its solutions belong to $\mathbb{R}$.



(a) *Room SHA model. The SHA receives events from the thermostat to start heating at different rates (locations high and low) or cooling naturally (location cool).*



(b) *Thermostat SHA model. The SHA is in charge of triggering the events that cause the room to start (event* `on`*) or stop heating (*`off`*).*

Figure 3.1: *Example of SHA network. Dashed arrows model probabilistic transitions with weights (in brown) $\mathsf{p_H}$ and $\mathsf{p_L}$ and solid arrows represent transitions with weight 1. Flow conditions, probability distributions, and exponential rates are in purple, channels in red, and guard conditions in green, respectively.*

In the following, we outline the semantics of SHA. In support of the overview, Fig. 3.2 shows an equivalent representation of the automaton of Fig. 3.1a that is fully compliant with the introduced syntax and semantics of SHA. The automaton of Fig. 3.1a must be understood as an equivalent simplified representation of the automaton of Fig. 3.2, where the latter differs from the former in the way the edge exiting location $cool$ is connected to locations $high$ and low.

Complex systems constituted by multiple entities can be modeled as a combination of SHA, which form a **network**. To make n automata $A_1, \ldots,$ A_n (each one defined as in Definition 1) form a network, the following properties must be satisfied [47]. Every automaton A_i must be deterministic, i.e., there are no two (or more) edges, outgoing from a location of A_i, defined by the same event and whose edge conditions can be satisfied by the same valuation. Moreover, they must guarantee the following two semantic properties, called input-enabledness and time divergence. Let $v'_{\text{var}} : W_i \to \mathbb{R}$ be a valuation for variables in W_i, $l_i \in L_i$ be a location of automaton A_i and let pair (l_i, v_{var}) be a configuration of A_i. For every automaton A_i, and for all configurations (l_i, v_{var}) and channel $\mathsf{c} \in C$, there exists an edge $\mathsf{c}?$ that can be taken, i.e., the edge is enabled as the associated edge condition in $\Gamma(W)$ is satisfied by v_{var}. Intuitively, this assumption ensures that every automaton can fire a transition in every possible configuration. Second, every automaton A_i always allows for executions such that if A_i is equipped with an extra clock that is never reset then, in all executions of A_i, this clock cannot be bounded by any arbitrary integer constant (i.e., no Zeno executions are feasible).

Finally, every automaton A_i is defined by considering the same set of channels ($C_i = C_j$ when $i \neq j$) and no pair of transitions, each one be-

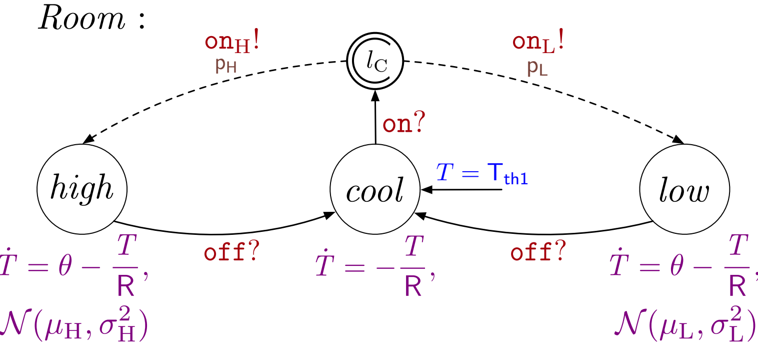


Figure 3.2: *SHA modeling the room from the running example with a detailed representation of how probability weights on receiving edges are handled.*

longing to two different automata in the network, are built by referring to the same event c!. These properties are with no loss of generality. For example, the disjointedness of channels can always be achieved by choosing properly defined symbol sets C_i. In addition, for simplicity, for the network $A_1, \ldots, A_n$ to be composable, the sets of real-valued variables must be pairwise disjoint ($W_i \cap W_j = \emptyset$ when $i \neq j$). However, this constraint can be relaxed through a suitable extension of the semantics.

The semantics of a composable network $A_1, \ldots, A_n$ is defined based on the configurations (of the network), where a configuration is a tuple of the form $(s_1, \ldots, s_n)$ and a state s_i is a configuration of automaton A_i. Configuration changes are realized either through *discrete transitions* or *time transitions*.

A discrete transition occurs when one or more automata take an edge. In the latter case, at least two automata synchronize with each other. Synchronization among different automata inside a network occurs through the channels of set C [123]. Given a channel $\mathtt{c} \in C$ and two edges of two distinct automata, whose events are c! (the *sender*) and c? (the *receiver*), triggering an event through channel c causes both edges to fire simultaneously. Synchronization always requires at most one sender and possibly many receivers (even none).

In Fig. 3.1b, the thermostat can trigger an event through channels on! and off! to start or stop heating the room. The triggered event is then received by the room automaton through labels on? and off?, which makes the corresponding edges fire. Taking an edge (l, c, γ, ξ, l') of automaton A_i with configuration (l, v_{var}) implies that the edge is enabled—i.e., all the conditions in $\Gamma(W)$ associated with the edge are satisfied by the values defined by v_{var}. Upon taking the edge, the location of A_i changes from l to l', and the associated update ξ is executed, resulting in configuration (l', v'_{var}). Since several automata may be involved in the synchronization and many updates can be executed simultaneously, specific rules are needed to regulate their execution. The value of a variable w in v'_{var} is determined based on the interpretation of w, i.e., whether w is a stochastic parameter or not. In the former case, upon entering a location $l' \in L_i$ such that $\mathcal{D}_i(l')$ is defined, a realization of distribution $\mathcal{D}_i(l')$ (e.g., $\mathcal{N}(\mu_{\text{H}}, \sigma^2_{\text{H}})$ in Fig. 3.2) defines the value of w in v'_{var} (e.g., θ in Fig. 3.2); otherwise, when $\mathcal{D}_i(l')$ is not defined, the value of w in v'_{var} and in v_{var} is the same [46]. In the latter case, w is not interpreted as a randomly distributed parameter, and its value in v'_{var} is the value of the assignment associated with w' in ξ, that is obtained by evaluating every non-primed variable of the constraint with values from v_{var}. The configuration (l', v'_{var}) is such that the valuation v'_{var}

satisfies the invariant $\mathcal{I}_i(l')$.

In SHA, besides randomly distributed variables, probability measures can be associated with delays to model the elapsing of time in the network, hence the wait between the occurrence of two discrete transitions. According to [47], the adopted probabilistic semantics is based on the "principle of independence" among automata in the network. Upon the firing of an edge, for every automaton A_i in the network, a delay d_i models the time A_i waits before taking an edge for the event c!, for some $\mathsf{c} \in C$. If no edges for event c! originate from l', then d_i is ∞. Otherwise, let $d_{\min}(l', v'_{\text{var}})$ be the minimum delay that automaton A_i should wait before an edge whose event is c!, and departing from l', is enabled; and let $d_{\max}(l', v'_{\text{var}})$ be the maximum delay that automaton A_i can wait before all edges, for events c!, with $\mathsf{c} \in C$, exiting l' are disabled (note that both values are a function of the invariant $\mathcal{I}_i(l')$, of the edge conditions and the current valuation v'_{var}). If $d_{\min}(l', v'_{\text{var}})$ is not defined, then d_i is ∞; otherwise, if $d_{\min}(l', v'_{\text{var}})$ is defined, d_i is a realization of the probability distribution $\mu_i(l', v'_{\text{var}})$. If $d_{\max}(l', v'_{\text{var}})$ is finite, then $\mu_i(l', v'_{\text{var}})$ is a uniform distribution over the interval $[d_{\min}(l', v'_{\text{var}}), d_{\max}(l', v'_{\text{var}})]$; otherwise, $\mu_i(l', v'_{\text{var}})$ is an exponential distribution over $[d_{\min}(l', v'_{\text{var}}), \infty)$. If d_i is ∞, by input-enabledness, then A_i can take an edge, whose event is c?, for some $\mathsf{c} \in C$. Otherwise, by definition of d_i, after d_i time units from the current discrete transition, automaton A_i can surely take an edge, whose event is possibly c!, for some $\mathsf{c} \in C$. Since the network consists of n automata, the minimum allowed progress d_{m} is selected among the n delays $d_1, \ldots, d_n$. If d_{m} is finite, then d_{m} is the time the network waits before an automaton performs a new discrete transition.

The wait between the execution of two discrete transitions, lasting a generic $\delta > 0$ time units, is a time transition, i.e., a configuration change such that no location of the automata in the network is modified. Still, values of the variables evolve because of the elapsing of time.

The configuration of the i-th automaton at the end of this wait is (l'', v''_{var}), with $l'' = l'$, where (l', v'_{var}) is the configuration whence the timed transition starts. All the variables of the set W_i evolve according to the flow conditions $\mathcal{F}_i(l')$. In the case of clocks $x \in X_i$, for instance, they are incremented by the value δ, hence, $v''_{\text{var}}(x) = v'_{\text{var}}(x) + \delta$ holds. The value of the other variables is determined based on the differential equation specified by $\mathcal{F}_i(l')$. With the adopted semantics, δ is the value d_{m} calculated at the occurrence of the last discrete transition.

After d_{m} units of time, the automaton A_i such that $d_i = d_{\text{m}}$ holds performs a discrete transition for some event c!, with $\mathsf{c} \in C$. If several edges

are enabled in (l', v'_{var}), probability distribution $\mathcal{P}(l')(\mathtt{c}!, \gamma', \xi', l'') \in [0, 1]$ with $l'' \in L_i$, $\gamma' \in \Gamma_i(W_i)$, and $\xi' \in \wp(\Xi_i(W_i))$ determines how likely the system is to evolve in one direction rather than the other. In Fig. 3.2, $\mathsf{p_L}$ and $\mathsf{p_H}$ are the probability of the switching of the heating when the window is open or closed, respectively, which takes place after the synchronization between the two automata has been achieved through channel on. Channels $\mathtt{on}_{\mathrm{H}}$ and $\mathtt{on}_{\mathrm{L}}$ in Fig. 3.2 model a probabilistic choice and are not intended for synchronizing the two automata.

Some of the models presented in this work do not conform with the disjointness of the set of real-valued variables since two or more automata can use the same variable. This, however, is with no loss of generality as it is always possible to introduce suitable transitions and local copies of the shared variables and build a network such that all sets of real-valued variables are pairwise disjoint. An automaton $\mathcal{A}_1$ can always make an automaton $\mathcal{A}_2$ change a variable v in W_2 by means of two synchronizing edges with a dedicated event, possibly representing the operation to be carried out on v.

3.1.2 Statistical Model Checking

Unlike exhaustive model-checking, SMC does not explore the state space. The technique consists of applying statistical techniques to a set of *runs* of the formal model to estimate the *probability* of a desired property holding. Specifically, we compute the value of expression $\mathbb{P}_{\mathrm{M}}(\psi)$ to estimate the probability of ψ holding for M [46]. Property ψ, in our framework, is of the form $\diamond_{\leq\tau}\mathtt{ap}$, where $\diamond$ is the metric "eventually" operator and $\mathtt{ap} \in \mathrm{AP}$. Formula $\diamond_{\leq\tau}\mathtt{ap}$ is true if ap holds within τ times units from the onset.

The value of $\mathbb{P}_{\mathrm{M}}(\psi)$ can be compared against a threshold $\vartheta \in [0, 1]$, i.e., formula $\mathbb{P}_{\mathrm{M}}(\psi) \sim \vartheta$ is evaluated, where $\sim \in \{\leq, \geq\}$. In this case, the SMC experiment yields a Boolean value indicating whether the probability of ψ holding for M is $\sim$ than ϑ (thus, `true`) or not (yielding `false`), which is calculated through *hypothesis testing*.

Otherwise, the SMC experiment returns *confidence interval* $[\mathrm{p_{min}}, \mathrm{p_{max}}]$ of property ψ holding for M, calculated according to the Clopper-Pearson method [44]. In this case, to determine when the generated set of runs is sufficient to conclude the experiment, Uppaal checks the length of interval $\epsilon = (\mathrm{p_{max}} - \mathrm{p_{min}})/2$. Uppaal stops generating new traces when $\epsilon \leq \epsilon_{\mathrm{th}}$ holds, where ϵ_{th} is an experimental parameter indicating the maximum desired estimation error. The smaller ϵ_{th}, the more accurate the estimation must be and, thus, more traces are required.

It is worth noting that SMC, although it partially bypasses the state

space explosion problem, it also provides weaker guarantees than exhaustive probabilistic model checking [121]. As a matter of fact, the latter provides an exact calculation of the probability of a property holding, whereas SMC only guarantees that said probability falls within a certain range [195]. One of the drawbacks of a simulation-based technique such as SMC is that obtaining an accurate result for rare properties may require a significant number of runs or longer traces. Agha and Palmskog survey several techniques addressing this issue, mostly based on the importance sampling technique [201] which is also integrated in an extension of Uppaal SMC [106].

Through Monte Carlo-based simulation, it is also possible to calculate the expected value of distributions defined by means of functions max (maximum) and min (minimum) when they are applied to stochastic processes, such as expressions of variables in W. Given an upper bound τ, formulae $\mathrm{E}_{\mathrm{M},\tau}[\mathsf{max}(v)]$ and $\mathrm{E}_{\mathrm{M},\tau}[\mathsf{min}(v)]$, with $v \in W$, indicate respectively the expected value of the maximum and minimum value of variable v along executions that last *at most* τ time units.

For example, with the model of Fig. 3.1, we can compute the probability of getting to operational state $high$ within 10 seconds since the onset of the system by evaluating the formula $\mathbb{P}_{\mathrm{M}}(\diamond_{\leq\tau} high)$ with $\tau = 10$ and the expected value of the maximum and minimum temperature in the room by computing $\mathrm{E}_{M,\tau}[\mathsf{max}(T)]$ and $\mathrm{E}_{M,\tau}[\mathsf{min}(T)]$.

SHA network M modeling an interactive scenario is put through SMC without specifying a probability bound ϑ. Specifically, we estimate the probability of success corresponding to expression $\mathbb{P}_{\mathrm{M}}(\diamond_{\leq\tau}\ \mathsf{scs})$, where Boolean variable $\mathsf{scs} \in V_{\mathrm{dc}}$ becomes true when all services constituting the mission are complete (thus, the mission has ended in success). Moreover, we estimate the expected maximum value of human agents' fatigue (corresponding to formula $\mathrm{E}_{\mathrm{M},\tau}[\mathsf{max}(F)]$, where $F \in W$ models the human fatigue) and of the minimum battery charge of robots (corresponding to formula $\mathrm{E}_{\mathrm{M},\tau}[\mathsf{max}(Q)]$, where $Q \in W$ models the battery charge).

3.2 Pre-Existing Tools

This section recaps the main features of pre-existing tools integrated into the toolchain for the formal verification and deployment phase.

3.2.1 Uppaal

Uppaal is a tool for the definition, simulation, and formal verification of real-time systems through networks of Timed Automata (or extensions of such formalism) [123]. The Uppsala and Aalborg Universities have jointly developed the tool since 1995.

The main components of Uppaal exploited by this work are the description language and the model-checker. The former is a custom notation to define the network of automata and their behavior with a C-like syntax. Although a graphical interface is provided, this work mainly exploits the XML format compatible with the model checker to generate the automata network programmatically rather than manually.

With purely Timed Automata networks, the model-checker explores the state space exhaustively to check reachability or safety properties. However, with the extension introduced in 2015, the tool supports the modeling and verification of automata with stochastic and hybrid features, to which end the support of SMC has been introduced [46]. Specifically, the model-checker engine has been extended to support the techniques underlying SMC and compute expressions of the form described in Section 3.1.2, which are widely exploited by this work.

It is noteworthy that, unlike other tools for SMC [25,112,196,222], Uppaal allows for the verification of timed systems with fully general hybrid variables, i.e., fundamental requirements for the presented framework.

3.2.2 Robot Operating System

ROS (Robot Operating System) is an open-source framework providing operating system functionalities [176], of which this work mainly exploits message-passing between processes.

At its core, ROS relies on the concepts of processes, called *nodes*, communicating over *topics* to exchange messages, i.e., structured typed data. In ROS, communication between nodes may conform to different architectural patterns, such as client-server and peer-to-peer. In any case, the framework supports the distribution of nodes across different machines.

ROS comes into play within the framework's workflow during deployment, which requires the different agents involved in the interactive scenario to communicate and exchange data. Such exchanges occur over ROS topics, while involved entities (i.e., robotic devices and sensors worn by human agents or present in the environment) constitute the decentralized network of ROS nodes.

3.2.3 CoppeliaSim

CoppeliaSim (previously known as V-Rep) is a 3D robotic simulator [185]. Among other robot simulation tools, some of CoppeliaSim's features make it especially suitable for the presented framework's purposes, specifically to be *integrated* into the deployment infrastructure.

Firstly, the tool is fully compatible with ROS and provides rich remote API functionalities. Consequently, agents in the simulation scene are fully controllable by external scripts (e.g., virtual human agents) or physical devices deployed in the physical setting (e.g., virtual robotic agents mimicking the behavior of physical robotic platforms). Similarly, through such API functions, the simulation scene can be programmatically generated starting from high-level user-defined configuration parameters to spare manual effort on the practitioner's side.

Secondly, unlike other simulators [11], within CoppeliaSim, it is possible to simulate the kinematics and dynamics of human agents (e.g., while sitting, walking, and running at different speeds) through the realistic physics engine. Furthermore, as previously mentioned, the behavior of virtual human agents (i.e., their choice to perform or not perform a particular action) is fully controllable and monitorable remotely.

CHAPTER 4

Model-Driven Framework

This chapter showcases the main features and the operational workflow of the developed model-driven framework for interactive robotic applications in service settings.

The development toolchain, whose workflow is shown in Fig. 4.1, is structured into four phases:

(1) ***design-time analysis****: the designer configures the Human-Robot Interaction (HRI) scenario through the dedicated DSL. The formal model and the set of properties are automatically generated, and SMC is performed;*

(2) ***reconfiguration****: the analyst examines the estimated quality metrics of the scenario and, if necessary, applies reconfiguration measures;*

(3) ***deployment****: when design-time results are deemed acceptable, the analyst either deploys the scenario in a physical environment or simulates it in a hybrid/fully virtual environment;*

(4) ***model adjustment****: traces collected during deployment serve as*

the training dataset for the automata learning algorithm, which infers a refined model of human behavior. The updated model is plugged back into the SHA network to obtain refined formal analysis results.

4.1 Design-Time Analysis

The development framework helps designers develop robotic applications by providing an estimate of the success probability and the level of comfort of involved humans. Target users (i.e., the designers) are domain experts such as healthcare logistics specialists or clinical engineers.

4.1.1 Scenario Configuration

As represented in Fig. 4.1, design-time analysis begins by configuring the scenario through a custom DSL (task "Scenario Configuration" in Fig. 4.1).

The DSL file is automatically processed to generate the SHA network and set of properties according to the user's specifications. The designer specifies the characteristics of the robots, the involved humans, the geometrical representation of the environment layout, and the robotic missions.

Our framework features a predefined (yet extensible) set of high-level patterns identifying recurring interaction contingencies in assistive applications (e.g., a robot following a human). A *service* corresponds to an interaction pattern; therefore, for each service, the robot must perform the actions implied by the associated pattern (e.g., retrieve an object and deliver it back to the human). For each mission, the robot must provide services requested by the human in the order specified by the designer.

Since the framework is not tied to a specific robot manufacturer or model, robotic platforms available in the fleet may not be pre-programmed to perform all required tasks. Therefore, the framework envisages an *ad-hoc* robot controller, hereinafter referred to as "*orchestrator*", in charge of monitoring the state of the system and sending commands to the robotic agent and suggestions to human subjects in conformity with the interaction patterns.

Moreover, it is paramount to take into account the robot's level of charge and charge/discharge cycles (whose parameters vary among different battery models), which may impact the duration of the mission (thus, its probability of success within a certain time range).

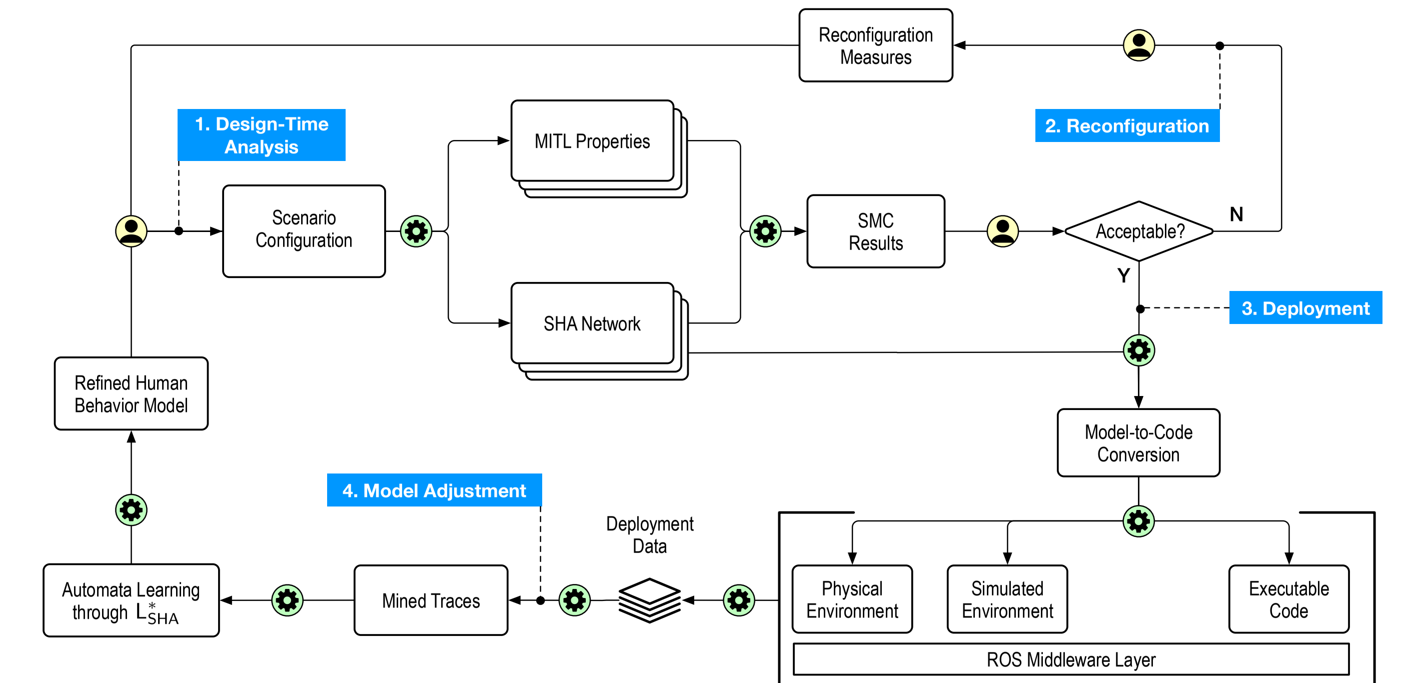


Figure 4.1: *Diagram representing the model-driven framework operational workflow. Yellow circles mark actions performed by the* user *(i.e., the application designer) and green circles correspond to the* automated *steps. The beginning of each phase of the framework is marked in blue and numbered according to the execution order: (1) design-time analysis; (2) scenario reconfiguration; (3) scenario deployment; (4) model adjustment.*

4.1.2 Robotic Mission Verification

Under these premises, each mission is modeled by a SHA network featuring the following automata:

A. $\mathcal{A}_{\mathrm{h}_i}$ with $i \in [0, \mathsf{N}_\mathsf{h} - 1]$ modeling the N_h humans involved in the scenario;

B. $\mathcal{A}_{\mathrm{r}_i}$ with $i \in [0, \mathsf{N}_\mathsf{r} - 1]$ modeling the N_r mobile robots;

C. $\mathcal{A}_{\mathrm{b}_i}$ with $i \in [0, \mathsf{N}_\mathsf{r} - 1]$ modeling the N_r batteries (one for each robot);

D. $\mathcal{A}_{\mathrm{o}_i}$ with $i \in [0, \mathsf{N}_\mathsf{r} - 1]$ modeling the N_r orchestrators (one for each robot).

The tool then automatically verifies through Uppaal the specified properties.

4.2 Scenario Reconfiguration

At the end of the design-time analysis phase, the designer manually examines the verification results and assesses whether they satisfy their quality criteria, for example, if the probability of success is sufficiently high or the expected value of fatigue is not excessive for any human.

If the results are not acceptable, the designer modifies the scenario (e.g., missions, environment layout, fatigue profiles) and repeats the analysis.

Possible reconfiguration measures include:

RM1. Assigning a different **robot** to the task if the facility has more than one available device. It may not be feasible for a human subject participating in a service to change their starting position due to facility policies (e.g., patients necessarily start in the waiting room). On the other hand, two robotic devices in a fleet may differ either because of their starting position or initial battery charge. In both cases, this reconfiguration measure can cut down the overall duration of the mission. In the first case, the robot may require less time to reach the first human to serve. In the second case, the robot may take less time to recharge or skip recharging entirely while carrying out the mission.

RM2. Changing the **order** in which humans are to be served. Note that this is only possible if there are no logical *dependencies* between the services being swapped. This measure can reduce the overall duration of the mission if the robot has to cover a smaller distance between services and the maximum level of fatigue reached by human subjects (thus,

impacting their well-being); for example, if a patient has more time to rest between two services.

RM3. Changing the **target** of a pattern, if feasible and compliant with the facility policies. An example is a patient following the robot directly to the doctor's office without going through the waiting room first. This reconfiguration measure can reduce both the mission's duration and the fatigue level reached by human subjects. Reducing the active time leads to a decrease in the fatigue endured since, during a fatigue cycle, time is the only variable in Eq.6.5.

RM4. Modifying the orchestrator's **thresholds**. For example, reducing the charge threshold at which the robot is instructed to move to the recharge station or the fatigue level at which the orchestrator suggests the human stop walking is possible. The designer must handle the trade-off between the decrease in mission duration and the increased probability of failure (for example, due to battery degradation).

The configuration space resulting from the described measures is, indeed, potentially dense. Therefore, making optimal choices may not always be feasible for a human operator without further support. To this end, the framework should be further extended to minimize the operator's effort in exploring the configuration space. Optimal task allocation [114], scheduling [212], and online adaptation [162, 172] in human-robot interactive systems is a long-standing research field. As a matter of fact, the modeling framework implies an intrinsic non-deterministic nature of the system under modeling that is not cleared through verification since the selected technique, SMC, is also not exhaustive. However, as already mentioned, clearing the non-determinism manually entirely is unreasonably time-consuming. A possible direction, also described in Section 17.2, is to apply optimization techniques *offline* and identify a limited number of alternative plans of actions (each possibly optimizing different quality measures of the robotic missions) for the operator to choose from.

Once the preferred plan of action is selected and design-time results are acceptable, the application can move forward to deployment or simulation.

4.3 Scenario Deployment

The goal of the deployment phase is to run the robotic application in a real environment or simulate it with a realistic physics engine. Also, deployment helps the application designer extract valuable information about the

missions and, possibly, drive a reconfiguration of the scenario, since real executions are available from the scene. In both cases, executable software is built to run the application.

The approach allows for a *hybrid* deployment environment adhering to the digital-twin paradigm [182] with an actual robotic device in the physical environment interacting with human avatars in the virtual environment.

When a simulation is performed, the virtual agents can be controlled by employing specific software components that the simulator executes to manifest the agents' behavior in the virtual scene.

Advanced simulator environments, as in the case of CoppeliaSim, also offer rich control dashboards that render the virtual 3D scene graphically and allow the simulator user to interact with it through input devices while the scene develops.

Within the framework, to ensure that the deployed orchestrator enforces the same policies as in the formal model and that, in the case of simulation, the virtual agents behave correspondingly to their respective SHA, a model-to-code mapping principle converts every SHA into an executable deployment unit. The latter consists of the executable orchestrator script or the scripts governing the agents' behavior—either the humans or the robot—within the virtual scene.

As the presence of humans is not always guaranteed and, when human agents are patients in distress, even discouraged, the application designer performing the simulation directly controls human avatars within the virtual scene to make them act like real humans in real-world scenarios.

The framework allows the designer to issue commands to the avatars through input devices, such as the keyboard, through the scripts that control them. The human actions modeled with the automata are mapped to keys; keystrokes performed by the application designer are interpreted by the scripts and then rendered in the scene.

To replicate physical fatigue sensors in the simulated environment and the stochastic behavior of fatigue rates, scripts extract a random sample from a pool of publicly available electromyography signals and estimate fatigue curves using the technique described in [139, 156].

The deployed orchestrator and the agents communicate over a network of ROS publisher and subscriber nodes [176] (the "Middleware Layer" in Fig. 4.1). Each automaton corresponds to a deployment unit (e.g., $\mathcal{A}_{\mathrm{O}}$ maps to the executable orchestrator and $\mathcal{A}_{\mathrm{R}}$ to the robot). The firing of an event through channels in set C in the formal model (of which Fig. 6.14 shows an overview) corresponds to the publication of a message on a ROS topic. More specifically, the deployment unit corresponding to the "*sender*" SHA

(i.e., the one with the edge labeled with c! with $\mathtt{c} \in C$) is the publisher node, whereas the "*receiver*" SHA (i.e., with the edge labeled with c?) is the subscriber node.

The designer will examine each run or simulation of the application for the reconfiguration phase. Specifically, all data that sensors (either real or simulated) publish through ROS nodes (i.e., the robot's battery and position values and all humans' fatigue and position values) are stored to be examined.

The robot carries out the mission by providing services in sequence. The orchestrator logs relevant events concerning the advancement of each service: when it begins, when it is completed, when it has to be interrupted and why (either the human is too tired or the battery is too low), whether the entire mission fails and the source of the failure. Data logged by the orchestrator are necessary to assess whether the deployed mission has ended with success.

4.4 Model Adjustment

The model adjustment phase begins by processing data collected during the deployment to assess whether unexpected contingencies emerged at runtime. Data analysis from deployment and reconfiguration may be necessary as SHA modeling human behavior have stochastic features that are necessarily an *approximation* of the behavior observable in reality.

On the other hand, the framework targets the *service* level of robotic systems' architectures [144], as it focuses on the workflow of the mission rather than on aspects related to hardware or structural components: automata modeling the robot, its battery, and the orchestrator do not thus have stochastic features (i.e., in $\mathcal{A}_{\mathrm{r_i}}$, $\mathcal{A}_{\mathrm{b_i}}$, and $\mathcal{A}_{\mathrm{o_i}}$, function $\mathcal{D}(l)$ is undefined for all $l \in L$ and all edges have probability weight 1).

Potential hazards due to sensor faults are supposed to be managed at deployment level. Specifically, fault estimation and compensation techniques [135] must be applied at runtime so that the executable orchestrator (i.e., the "executable code" component in Fig. 4.1) reasons on measurements as accurate as possible. Similarly, the probability of hardware malfunctions (i.e., one of the mobile robot's engines breaking during the execution of the mission) is also assumed negligible given the ratio between the estimated lifespan of a hardware component (in the order of magnitude of years) compared to the duration of a mission (in the order of magnitude of minutes). Under these premises, human behavior represents the primary source of uncertainty within the domain of the framework. Rounds of for-

mal analysis before deployment (multiple ones if offline reconfiguration is performed) are carried out with default versions of SHA modeling human behavior for each interaction pattern. However, different contingencies impacting the human's physiological state (and thus the mission's outcome) may be observed while deploying the application on the field.

To this end, the framework envisages a model adjustment phase that exploits field data to learn a refined model of human behavior. Collected data are processed and fed to an active automata learning algorithm called $\mathsf{L}^*_{\mathrm{SHA}}$, representing one of the main contributions of this research.

$\mathsf{L}^*_{\mathrm{SHA}}$ learns a SHA modeling human behavior, identifying the graph (i.e., locations, edges, and channel labels) of the SHA, its flow conditions, and probability distributions from mined traces. Each trace consists of the sequence of *events* that occurred during an individual execution of the mission, where events represent actions performed by the human subject that impact their physical fatigue.

The learned SHA is extended with features that are standard across all SHA (and thus not learned by $\mathsf{L}^*_{\mathrm{SHA}}$) and plugged back into the SHA network replacing the original automaton approximating human behavior. Such features are necessary to make the learned SHA fully compliant with the SHA network, as they include fixed modeling patterns representing the periodic update of sensor readings or the synchronization with the orchestrator to conform to an interaction pattern. As this set of features can be programmatically added to the learned automaton, it is not accounted for by the learning algorithm. By doing so, $\mathsf{L}^*_{\mathrm{SHA}}$'s computational load is fully allocated to features that can only be inferred from data.

Design-time analysis (and eventually reconfiguration) can then be iterated with the refined model of human subjects to obtain updated quality metrics of the scenario that exploit the knowledge accumulated by deploying the mission on the field.

The cyclical nature of the framework allows the analyst to modify the scenario and perform multiple iterations of the analysis until verification results are deemed acceptable. The framework also supports the designer in terms of which and how many scenario parameters require manual specification. The framework provides parameters concerning the robot (i.e., speed and acceleration) and battery behavior (i.e., charge and discharge rates) to decrease the manual effort required on the designer's side.

Designers manually specify parameters concerning the specific robotic mission (i.e., how many humans are involved and the service they request) whose value cannot be known a priori by the framework. As discussed in Section 4.2, as the scenario under analysis grows in complexity, so does the

manual labor required to identify and tune its parameter. A possible way to approach this issue is to split the mission into smaller sequences to be individually analyzed. However, in this case, the framework should also be further extended to identify the dependencies of each task on previous ones. Consequently, the framework would allow for the automated definition of the input parameters of a sub-sequence that derive from the system's state at the end of the previous sub-sequence.

Part I

Design-Time Analysis

CHAPTER 5

Human-Robot Interactive Scenario Configuration

This chapter presents in detail the scenario configuration task (see Fig. 4.1). Firstly, the conceptual model underlying the framework is presented as a Class Diagram containing the configurable entities of a HRI scenario analyzable through the framework. The custom DSL developed to specify interactive scenarios is then introduced in detail, exemplifying the different primitives through the illustrative scenario in Section 1.2.

5.1 Conceptual Model

The framework covers human-robot interaction scenarios with specific characteristics: scenarios take place in a known layout (robotic missions carried out in unknown environments do not fall within the scope of this work), and the service sequence does not change when the application is already running.

Fig. 5.1 shows the conceptual model (represented as a Class Diagram) capturing the main entities of the scenarios and their relations. The dia-

gram, described in detail in the following, constitutes the conceptual foundation of the DSL and the working assumptions that underlie the formal model. Configuring a specific scenario through the DSL to be formally verified is equivalent to defining an instance (i.e., an Object Diagram) of the conceptual model.

A Scenario comprises at least one robotic mission. Each Mission is *set in* a known Layout, which we represent as a *composition* of one or multiple two-dimensional rectangular areas. Each Area is a *composition* of four corner Points, each characterized by a pair of Cartesian coordinates x and y. A Layout also includes a relevant subset of points, called Points Of Interest (POI), that can be the target of an action, such as room entrances, cupboards, and the robot's recharge station. Missions, areas, and POIs are identified through attribute name.

A mission is a sequence of services *requested* by a human and *provided* by a robot (specifically, humans are served according to their id). Each Service conforms to an interaction pattern (attribute ptrn) and has a target POI. Patterns (i.e., the items of enumeration InteractionPattern) group typical interaction contingencies and are listed in the following:

P1. HumanFollower: the human *follows* the robot to a specific destination (attribute target in Service) of which they do not know the precise location. For example, a patient looking for the waiting room or a doctor's office conforms to this pattern. The human follows the robot, but if they decide to stop walking, the robot also stops and waits for the human to get closer again. The robot signals that the service has been completed when both the robot and the human are close to the destination.

P2. HumanLeader: the human has to *lead* the robot to a specific destination of which they know the precise location (attribute target in Service), for example, a nurse requiring the robot to escort them while carrying tools or medications. The human can decide when to start or stop walking, and the robot follows. The human is in charge of signaling when the service has been completed, i.e., when both the human and the robot have reached the destination.

P3. HumanRecipient: the human *waits* for the robot to fetch an item from a specific location (attribute target in Service) and bring it back to the human, for example, a doctor requiring the robot to fetch a tool or a medication from a colleague and bring it back to their office. While the robot fetches and delivers the object, the human is

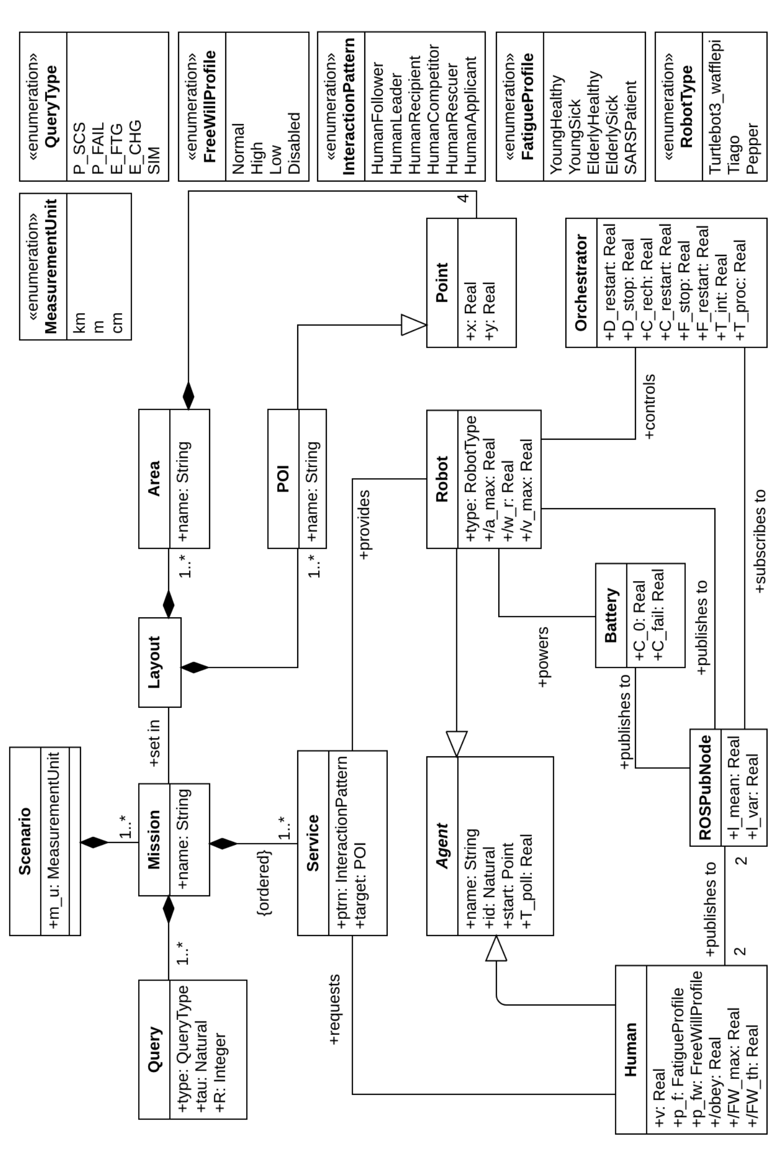


Figure 5.1: *Class Diagram representing the entities constituting a scenario. Throughout the thesis, when referring to a class of the model rather than the abstract concept it represents, its name is capitalized, uses Sans-Serif font (e.g., "Scenario" rather than "scenario"), and italicized for abstract classes (e.g., "Agent"). Attributes with a subscript identifier are reported in the diagram with an underscore for visualization purposes.*

free to move around (and the robot adjusts the delivery destination accordingly). The human is responsible for determining whether the service has been provided when they have successfully collected the item from the robot.

P4. HumanCompetitor: the human and the robot *compete* to fetch a critical resource [68] (for example, a medical kit during an emergency). Both agents move to the location of the resource (captured by attribute target) to reach it as quickly as possible. The competition ends when either of the agents reaches the target location (effectively *winning* the competition). The human may autonomously decide to stop walking at any time.

P5. HumanRescuer: the pattern captures the robot requiring human intervention to complete a task, such as pressing a button to call the elevator or opening a closed door. In this case, the robot will emit audible or visible signals to notify its need for human support. The human autonomously decides to support the robot, move to the robot's current location (captured by attribute target), perform the required action, and conclude the interaction.

P6. HumanApplicant: the pattern captures the human requiring the robot's support in performing a specific task that implies timely or close-contact interaction, such as feeding a patient or administering medication. In this case, as soon as the service starts, the human waits for the robot to approach their current location (attribute target). When the robot is sufficiently close, the action requiring synchronization starts. The human may autonomously interrupt the activity and resume at any time.

Agents enact the mission. Abstract class *Agent* has a name, id, and starting position start within the layout. In the case of human agents, the id attribute determines the order in which humans are served. In the case of robotic agents, the id attribute determines the order in which missions are assigned to robots in the fleet in the case of multi-robot missions [130]. Agents are endowed with sensors that share a new reading every T_{poll} instants. There are two possible *specializations* of an *Agent*: humans and robots.

For each robot, attribute type from enumeration RobotType defines its commercial model (e.g., "TurtleBot3" or "Tiago"). We assume that a Robot moves with a trapezoidal velocity profile, whose maximum acceleration a_{max}, linear velocity v_{max}, and angular velocity ω_r are *derived* from

attribute type. Each Robot is *powered by* a lithium Battery with initial charge C_0. Class Battery's attribute C_{fail} corresponds to the lowest voltage under which the device must not move to prevent the battery pack from being damaged.

SHA modeling human behavior include a model of physical fatigue. Each Human has a p_f attribute determining their fatigue profile (see the FatigueProfile enumeration in Fig. 5.1), which determines their proneness to fatigue and recovery based on physiological factors. We distinguish subjects by age (Young/Elderly) and state of health (Healthy/Sick) or whether they are affected by a severe respiratory syndrome that hinders their mobility (SARSPatient), obtaining five possible fatigue profiles. Attribute v specifies the average walking speed.

Since human behavior is unpredictable in a real setting, our model includes a probabilistic approximation of haphazard human behavior (e.g., the possibility of ignoring a robot's instruction or starting and stopping freely during the interaction). Therefore, a Human also features attribute p_{fw} from which attributes obey, FW_{max}, and FW_{th} determining the probability with which such behavior manifests itself are derived. The p_{fw} attribute has four possible values (corresponding to the elements of the FreeWillProfile enumeration): Normal, High, Low, or Disabled. The latter causes the human free will to be entirely ignored at design time, used for a preliminary test of the scenario setup.

Agents and batteries are equipped with sensors that, during deployment, share data with the orchestrator over topics handled by the middleware layer based on ROS (see Fig. 4.1) [176]. The SHA network features a model of ROS publisher nodes (i.e., instances of class ROSPubNode in Fig. 5.1), mimicking the delay with which messages are processed and published. Delays are normally distributed with mean l_{mean} and variance l_{var} [208].

The Orchestrator monitors the state of the system by *subscribing* to sensor readings' topics of the human's position, fatigue, robot's position, and battery charge. The Orchestrator periodically checks the state of the system against its policies every T_{int} time units and processes data for T_{proc} time units. While processing, it checks sensor data against specific thresholds: D_{stop} and $D_{restart}$ determine the human-robot distance that causes the robot to stop and wait or restart, respectively; C_{rech} and $C_{restart}$ correspond to the battery charge levels that cause the robot to start or stop recharging, respectively; F_{stop} and $F_{restart}$ correspond to the human fatigue levels inducing the human to stop and rest or resume the action, respectively.

Finally, for each scenario, the analyst specifies the quality metrics to be computed through "*queries*". The framework currently supports three

query types (i.e., probability calculation, expected value calculation, and generation of traces) supported by Uppaal. For each Query, it is necessary to specify its type, time-bound τ and—optionally—the number of runs generated to verify the property, i.e., attribute R. We remark that R should only be specified to limit performance issues during preliminary testing. Otherwise, it is advisable to let the verification tool compute the number of runs necessary to perform verification with the required confidence level. The different query types (modeled by enumeration QueryType) allow the designer to estimate:

Q1. the probability of the mission ending with success (item P_SCS) within the time-bound. Success occurs when all services have been completed;

Q2. the probability of the mission ending in failure (item P_FAIL) within the time-bound. Failure occurs either when the robot is fully discharged and cannot move autonomously, or the human is fully fatigued (note that the mission not ending in success due to an insufficient time bound does not constitute a failure; thus the results of a P_FAIL and P_SCS query do not necessarily sum to 1);

Q3. the expected maximum value of fatigue (item E_FTG) for all humans within the time bound;

Q4. the expected minimum battery charge (item E_CHG) value within the time bound;

Q5. one or multiple (i.e., specified as parameter R) runs to have a more detailed overview of how the system behaves during the execution of the mission (item SIM).

5.2 Domain-Specific Language

As represented in Fig. 5.2, configuring a scenario through the DSL is semantically equivalent to defining an Object Diagram of the conceptual model in Fig. 5.1. Therefore, the developed DSL features primitives allowing for the creation of scenarios that reflect the conceptual model. Each primitive is presented in the upcoming subsections through an example scenario.

Fig. 5.2 shows how DSL files are converted into SMC experiment instances. Each DSL file defines a single scenario, which includes the layout geometry, the points of interest, the agents, and the mission, and represents

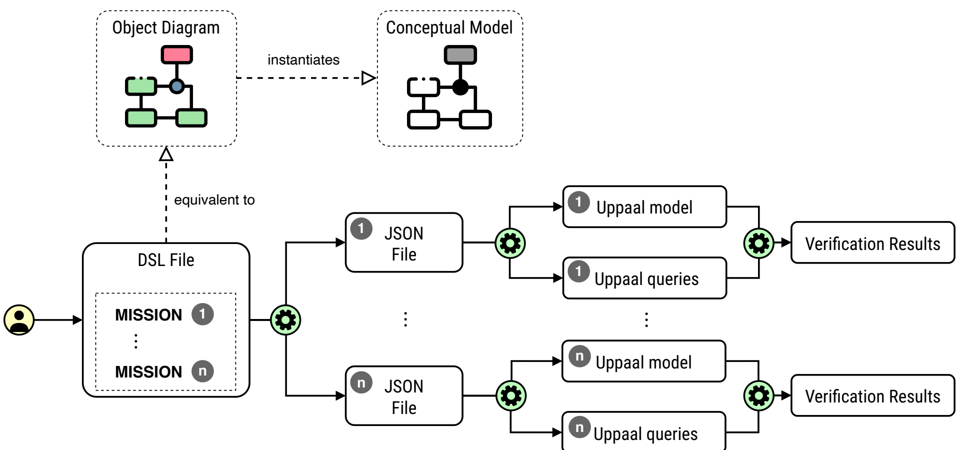


Figure 5.2: *Diagram representing the process that translates a DSL file into Uppaal models. Solid arrows represent operational tasks while dashed arrows represent conceptual equivalences. Solid arrows are marked to distinguish the actions performed by the user from those performed automatically. Boxes are numbered to identify the relations between defined missions and the output files of each phase.*

a well-formed DSL model if specific properties are met (e.g., rooms have non-null area, agents are located within the boundaries of the environment, etc.). Well-formedness properties are automatically verified by the translator every time a DSL file undergoes the conversion process.

Each mission in the DSL model is subject to formal verification separately, requiring, thus, a separate formal model. The conversion process features an intermediate phase: a JSON file containing the mission's characteristics is generated for each mission. Each JSON file is then converted into two files, one with the Uppaal model and one with the queries to perform the SMC experiment. The intermediate JSON notation, which is a lightweight and well-established standard, decouples the DSL from the specific verification tool and makes the framework flexible to the introduction of different verification tools or different DSLs. JSON files are also exploited to automatically set up the deployment environment.

The automated model generation tool (i.e., driving the conversion from JSON to Uppaal notation in Fig. 5.2) exploits Uppaal templates modeling the behavior of the agents in the workspace whose correctness is (probabilistically) verified before they are plugged into the model generation component. The DSL (and equivalent JSON file) determines which and how many instances of these templates must be generated and their starting configuration (i.e., the value of input parameters, such as agents' starting

positions). Above-mentioned well-formedness rules ensure that the JSON file cannot be generated (thus, the model generation process cannot start) unless such parameters are properly specified, thus ensuring that the resulting SHA Network is also well-formed and ready for verification.

The DSL does not have a specific statement for objects of class Scenario because each file inherently instantiates a single scenario, possibly including several missions. Every mission in a scenario consists of four independent sections, each one identified by the keyword **define**, concerning: layout definition, list of agents in the scene, list of services, and list of queries to be computed. Orchestrator and ROS nodes are instantiated automatically when the specification is translated into the model to be used for verification.

In the upcoming sections, the DSL primitives are illustrated through the illustrative scenario from Section 1.2.

5.2.1 Layout, Areas, and POIs

While modeling the HRI scenario, the user must specify the layout where the agents will operate. The DSL allows users to model different layouts (such as different building floors or different sections of the same floor) through the statement:

define layout

which includes a non-empty list of areas and POIs.

The DSL captures all layouts made up of adjacent rectangular areas (i.e., it does not capture curved or diagonal walls), each defined as:

area id **in** $(x_1, y_1)\ \ (x_2, y_2)$

where coordinate pairs (x_1, y_1) and (x_2, y_2) define one of the area's diagonal segments from which the other two corner points are automatically inferred upon generating the SHA network to save manual effort on the user's side. Areas' corners are validated to ensure that they correctly identify a diagonal, i.e., that $x_1 \neq x_2$ and $y_1 \neq y_2$ hold. Layout-related declarations are validated to check whether there are disconnected areas (i.e., all areas must be *reachable* from any point in the layout) and that no pair of areas overlap entirely.

POIs with their coordinates are declared through the following statement:

poi id **in** (x, y)

Listing 5.1 Example DSL section defining layout areas and POIs.

```
1 param measurement_unit m
2 define layout:
3   area A1 in (0.0, 17.5) (40.0, 7.5)
4   area A2 in (40.0, 25.0) (50.0, 0.0)
5   poi RC in (25.0, 17.5)
6   poi WR in (49.5, 12.5)
7   poi KIT1 in (40.5, 21.25)
8   poi KIT2 in (40.5, 3.75)
9
```

Although verification tools only handle adimensional variables, the DSL requires the specification of the length measurement units to ensure that the layout size is consistent with the robot's speed. The measurement unit is specified through the following statement, where $m_u \in \{\text{km}, \text{m}, \text{cm}\}$:

$$\textbf{param}\ \text{measurement_unit}\ m_u$$

The specification of the layout in Fig. 1.2 is given in Listing 5.1: the layout features two areas (a1 and a2) and three POIs corresponding to the recharge station (RC), the waiting room entrance (WR), KIT1, and KIT2. All coordinates are expressed in meters (m), consistently with Fig. 1.2b.

5.2.2 Agents

Each mission must feature a mobile robot and at least one human requiring assistance, specified in two independent sections that are identified through keywords **define robots** and **define humans**, respectively.

The DSL allows designers to declare each available robot through the statement:

$$\textbf{robot}\ \text{name}\ \textbf{in}\ (x, y)\ \textbf{id}\ \text{id}\ \textbf{type}\ \text{type}\ \textbf{charge}\ C_0$$

where parameters name and id univocally identify the robot, C_0 defines its initial level of charge, and coordinates (x, y) define its starting position. As per Section 5.1, while generating the formal model, the robot's type determines the translational and rotational speeds, and the acceleration (attributes v_{max}, ω_r, and a_{max}) in Fig. 5.1) based on the model's technical specifications. This feature of the DSL saves non-technical users the effort of retrieving these data when they might be more familiar with the type of robot available in the facility.

Listing 5.2 Example DSL section defining the agents and their features.

```
1 define robots:
2   robot ROB1 in (10.0, 12.5) id 1 type tiago charge 40
3   robot ROB2 in (45.0, 3.5) id 2 type turtlebot3_wafflepi charge
      90
4
5 define humans:
6   human HUM1 in (5.0, 12.5) id 1 speed 80 is young_sick freewill
      low
7   human HUM2 in (35.0, 9.0) id 2 speed 100 is elderly_healthy
      freewill normal
8
```

Each human is declared through the following statement:

$$\text{human name in } (x, y) \text{ id id speed v is } \mathsf{p_f} \text{ freewill } \mathsf{p_{fw}} \tag{5.1}$$

where the name univocally identifies the human and the id determines the serving order (thus, it is also required to be unique). Coordinates (x, y) determine each human's starting location. Parameters v, $\mathsf{p_f}$, and $\mathsf{p_{fw}}$ define the walking speed, fatigue and free will profiles as described in Section 5.1. The user chooses the values of $\mathsf{p_f}$ and $\mathsf{p_{fw}}$ out of a pre-determined list, corresponding to the enumerations in Fig. 5.1.

Both the robot and human declaration blocks are validated to ensure that no pair of agents share the same id (within the same *Agent* generalization) nor the same name (also across different *Agent* generalizations). The DSL is developed under the simplifying hypothesis that agents occupy a single point in space (corresponding to their center of gravity). This modeling choice is dictated by the need to keep the DSL (and, thus, the formal model) as simple as feasible and spare the designer from defining the three-dimensional envelope of the agents' bodies. The framework assumes that refined collision avoidance routines are already implemented at a lower level within the robotic platform and the DSL validator only checks that no pair of agents have the same center of gravity (i.e., coordinates (x, y)).

The agents from the running example are defined as per Listing 5.2. There are two robots available (ROB1 and ROB2) of different types (thus, they will have different speeds), and two humans (HUM1 and HUM2), of which one has a Young/Sick fatigue profile and low free will, whereas the second one is Elderly/Healthy and has normal free will profile.

5.2.3 Missions

Designers can declare multiple **missions** and associate them with a layout and a set of agents. Each mission is assigned to a single robot and verification experiments resulting from each mission declaration (see Fig. 5.2) are performed separately. A mission is declared as in the following, where parameter m is the name of the mission and r is the name of the robot it is assigned to (association *provides* in Fig. 5.1).

define mission m **for** r

As described in Section 5.1, each mission consists of a sequence of services and each service adheres to one of the interaction patterns described in Section 5.1. As per Statement (5.1) and the conceptual model presented in Section 5.1, humans are declared independently of the interaction pattern, which is specified when declaring the service as in the following:

do ptrn **for** h **with target** poi

where ptrn can be either robot_leader, robot_follower, robot_transporter, robot_competitor, robot_applicant, or robot_rescuer (corresponding to the HumanFollower, HumanLeader, HumanRecipient, HumanCompetitor, HumanRescuer, and HumanApplicant patterns, respectively), h is the name of the human requesting the service (association *requests* in Fig. 5.1), and poi instantiates attribute target in Fig. 5.1. Each service declaration is validated to ensure that h and poi refer to existing human agents and POIs.

The two missions associated with the running example are declared as in Listing 5.3. In mission m1, ROB2 has to lead HUM1 to POI WR and then deliver KIT2 to HUM2. In m2, ROB2 has to follow HUM2 to KIT1, then lead HUM1 to WR.

5.2.4 Queries

Finally, the designer specifies which experiments to perform for each mission. The DSL captures the set of queries shown in Fig. 5.1 and described in Section 5.1. A query is declared through the following statement:

compute query **with duration** τ **runs** R

where query can be either probability_of_success, probability_of_failure, expected_charge, expected_fatigue, or simulation. Parameter τ corresponds

Listing 5.3 Example DSL section defining the mission (i.e., the sequence of services).

```
define mission m1 for ROB1:
  do robot_leader for HUM1 with target WR
  do robot_transporter for HUM2 with target KIT2

define mission m2 for ROB2:
  do robot_follower for HUM2 with target KIT1
  do robot_leader for HUM1 with target WR

```

Listing 5.4 Example DSL section defining the set of queries.

```
define queries of mission m1:
  compute probability_of_success with duration 120 runs 300
  compute expected_fatigue with duration 120 runs 50

define queries of mission m2:
  compute probability_of_success with duration 100 runs auto
  compute probability_of_failure with duration 100 runs auto

```

to the time-bound. At the same time, R is the bound on the number of traces generated for the SMC experiment, whose value must be set to auto if the user wants the verification tool to compute the required number of runs.

A possible set of queries for missions 1 and 2 from the running example is given in Listing 5.4: the SMC experiments will estimate the probability of success and maximum fatigue value for all humans for m1, and probabilities of failure and success (with no bound on runs) for m2.

CHAPTER 6

Formal Modeling Human-Robot Interactions

In this chapter, we present the automata constituting the SHA network, i.e., the humans, robots and their batteries, and the orchestrators.

Human behavior is modeled through patterns Human Follower, Human Leader, Human Recipient, Human Applicant, Human Rescuer, *and* Human Competitor; *it features a stochastic characterization of fatigue and free will and a factorization of the common modeling pattern capturing the periodic sensor reading update.*

The model of the robot and the robot's battery has been fitted to the real platform used for the experimental validation.

Finally, we describe the structure of the orchestrator and each of its components in detail.[a]

[a] The content presented in this chapter also partially appears in [130] and [128]. The author of this thesis declares to have also authored the reproduced text, figures, and data and to have the right to reproduce such content in a dissertation according to the license under which both articles are published.

6.1 Formal Modeling Approach

This section illustrates the modeling approach we have adopted to *map* aspects of the real system to SHA features also summed up in Table 6.1.

The high-level goal of the SHA network is to capture the agents' behavior based on their current *operating state* (e.g., the human resting or walking). For every agent in the scenario and automaton $\mathcal{A}$ modeling its behavior, defined as in Definition 1, every operating state of the agent corresponds to a *location* in L.

SHA capture the evolution of relevant quantitative attributes of the real system, such as human fatigue and battery level of charge. Each physical attribute, characterizing a human or a component, corresponds to a real-valued variable in set $W \setminus \{X \cup V_{\mathrm{dc}} \cup K\}$ of their modeling automata and flow conditions $\mathcal{F}(l)$, associated with a location l, reproduce the set of ODEs constraining the evolution of real-valued variables in that specific operating state.

The switch between two operating states consists of an *edge* between the two corresponding locations. Recurrent features of the specific systems that our modeling approach targets identify two *types* of switches, which we refer to as *controllable* or *uncontrollable*. We remark that we use these two terms in a manner that is specific to our framework, and they are not part of the standard terminology of the formalism (for example, they are unrelated to the notion of controllable and uncontrollable edges in Timed Game Automata [19]); instead, they are merely aliases for specific edges recurring in our SHA (i.e., subsets of $\mathcal{E}$ from Definition 1).

A controllable switch occurs if, and only if, a specific event fires and synchronization among two or more automata occurs: for example, the robot starts accelerating when the orchestrator issues the command to start moving. For this reason, all the edges modeling a controllable switch are defined with a channel in $C \setminus \{\epsilon\}$.

Conversely, uncontrollable switches occur "naturally" in the original system due to the evolution of the physical variables at play: for example, the human unavoidably stops moving when their fatigue level reaches the maximum endurable threshold. Therefore, they are defined through event $\epsilon!$, i.e., by means of the internal action. A location l with an outgoing edge modeling an uncontrollable switch is endowed with a set of invariants of the form $w \leq \mathsf{k}_2$, where $w \in W$ is the real-valued variable subject to the constraint and $\mathsf{k}_2 \in K$ is its maximum allowable value (e.g., $F \leq 1$, with $w = F$ and $\mathsf{k}_2 = 1$, constrains the value of the human fatigue to be less or equal than 1). The outgoing edge then has condition $w \geq \mathsf{k}_1$, such that

$\mathsf{k}_1 \in K$ and $\mathsf{k}_2 \geq \mathsf{k}_1$ hold. If $\mathsf{k}_2 > \mathsf{k}_1$ holds, the edge fires with probability distributed uniformly over interval $[\mathsf{k}_1, \mathsf{k}_2]$, as explained in Section 3.1.1. If $\mathsf{k}_1 = \mathsf{k}_2$ holds, the edge fires with probability 1 when $w = \mathsf{k}_1 = \mathsf{k}_2$ holds (e.g., the edge from *on* to *off* in Fig. 3.1 where $w = T$ and $\mathsf{k}_1 = \mathsf{k}_2 = \mathsf{T}_{\mathsf{th}_2}$).

Since the orchestrator controls the robots by using the digital observations made by sensors on humans and robots, the SHA modeling physical dynamics (i.e., humans and robots) feature dense-counter variables (set V_{dc}) as the discrete (i.e., digital) equivalents of real-valued variables. Dense counters are periodically updated every $\mathsf{T}_{\mathsf{poll}} \in K$ time units (where $\mathsf{T}_{\mathsf{poll}}$ corresponds to the refresh period of the specific sensor) through updates in $\Xi(W)$ that are compatible with the ODEs modeling the dynamics of the physical attributes in each location.

To this end, every SHA in the network uses a clock $t_{\mathrm{upd}} \in X$ to measure the time elapsed between two consecutive measurements and to trigger an update. Therefore, when $t_{\mathrm{upd}} = \mathsf{T}_{\mathsf{poll}}$ holds for an automaton $\mathcal{A}$, hence when time $\mathsf{T}_{\mathsf{poll}}$ has elapsed since the last measurement, then $\mathcal{A}$ uncontrollably switches to a *committed* location. A committed location is equivalent to an ordinary location with invariant $t \leq 0$ and all incoming edges with update $t = 0$ for some $t \in X$: therefore, time cannot elapse while in these locations [123]. In that location, the dense counters modeling the latest sensor readings are immediately notified to the orchestrator by firing an event over a dedicated channel that triggers the publishing routine.

Controllable switches realize the interactions among the automata and are obtained through synchronization channels. Since the orchestrator implements the control logic that governs the agents, the issuing of a command by the orchestrator is modeled through synchronization between the orchestrator and the automaton modeling the agent that reacts to the command. Hence, all the channels in (the automaton modeling) the orchestrator are labeled with !, whereas (the automata modeling) the humans and the battery are defined with edges having the channels marked with $?$.

The modeling has been realized by considering one robot serving one individual at a time, even if several agents (i.e., humans and robots) can participate in the scenario. Each pattern models the interaction between one pair of agents, a robot and a human, and missions are finite interactions. Since there is always only one active robot and a single served human at a time, channels representing synchronizing events between the orchestrator and the agents are not specific to a single instance of a human or a robot. A dense counter in the orchestrator, with a finite domain, identifies the human currently interacting with the robot. It is evaluated by every automaton modeling the humans to allow or deny the firing of the synchronization

events with the orchestrator.

In particular, all the edges in the automaton modeling a human agent include a condition that evaluates to true when the dense counter indicating the currently served human is equal to the value id uniquely identifying it (see Section 5.1) and, hence, the automaton. Moreover, even if modeling multiple robots serving multiple humans simultaneously is possible in theory, adding this feature would cause the models to increase in complexity.

The information flow that the orchestrator realizes by issuing commands to agents through events via channels is as follows. The orchestrator

- informs the human to start or stop walking via channels $\texttt{cmd_h}_{\mathrm{start}}$ and $\texttt{cmd_h}_{\mathrm{stop}}$;
- makes the robot move or stop via channels $\texttt{cmd_r}_{\mathrm{start}}$ and $\texttt{cmd_r}_{\mathrm{stop}}$. Moreover, it starts the battery charging through channel $\texttt{cmd_b}_{\mathrm{start}}$ and interrupts the charging with channel $\texttt{cmd_b}_{\mathrm{stop}}$, restoring the robot to the mission-defined interaction.

6.2 Robotic System Model

In the following, we describe the SHA modeling the robotic platform and the robot's battery. Both exploit modeling pattern $\langle\mathsf{op}\rangle_\mathsf{pub}_{\langle\mathsf{id}\rangle}$ (see Fig. 6.1), which captures the periodical publishing of sensor readings.

The $\langle\mathsf{op}\rangle_\mathsf{pub}_{\langle\mathsf{id}\rangle}$ pattern is applied to agents' operating states. Location $x_{\langle\mathsf{op}\rangle}$ in Fig. 6.1 corresponds to a generic operating condition of agent x. Ports start, stop, and hold mark the transitions that enter and leave $x_{\langle\mathsf{op}\rangle}$,

Table 6.1: *Mapping between SHA elements and real agent's features.*

Agent Feature	$\longrightarrow$	**Automaton Feature**
State	$\longrightarrow$	Location: $l \in L$
Physical Variable	$\longrightarrow$	Real-valued Variable: $w \in W \setminus (X \cup V_{\mathrm{dc}} \cup K)$
Sensor Output	$\longrightarrow$	Dense-Counter Variable: $v \in V_{\mathrm{dc}}$
Design Parameter	$\longrightarrow$	Numerical Constant: $k \in K$
Time-Dynamics	$\longrightarrow$	Flow Condition: $f \in \mathcal{F}(l), l \in L$
Uncontrollable Switch	$\longrightarrow$	Invariant-Edge pair $(i \in \mathcal{I}(l), e \in \mathcal{E})$ s.t. $e = (l, \epsilon, \gamma, \xi, l'), \gamma \in \Gamma(W), \xi \in \Xi(W)$
Controllable Switch	$\longrightarrow$	Edge $e \in \mathcal{E}$ s.t. $e = (l, \mathsf{c}, \top, \xi, l')$ and $\mathsf{c} \in C \setminus \{\epsilon\}$

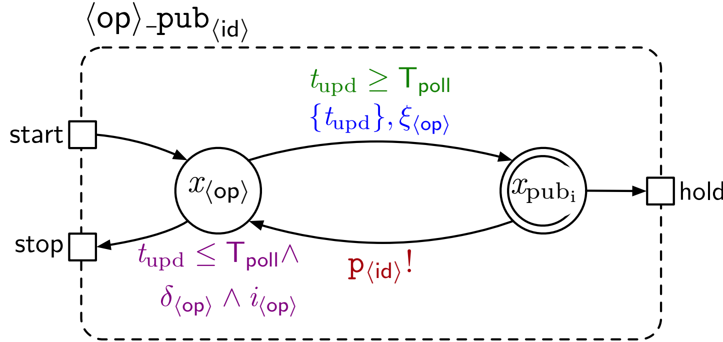


Figure 6.1: ⟨op⟩_pub$_{\langle\text{id}\rangle}$ *pattern, applied to all* $x_{\langle\text{op}\rangle}$ *locations in the network to model the message publishing mechanism on queue* ⟨id⟩*, as seen in [128].*

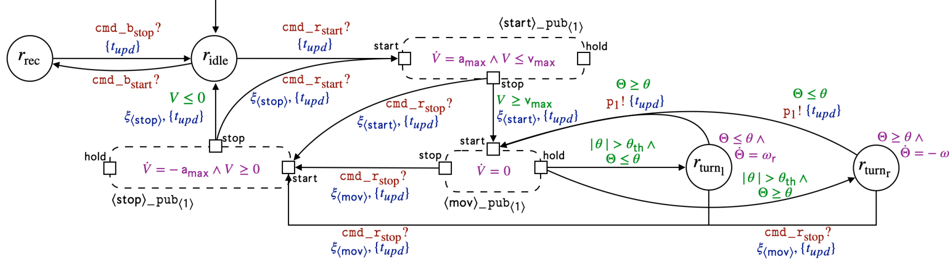


Figure 6.2: *SHA modeling the robotic platform, as seen in [128].*

whose characteristics change from case to case. Ports are not officially part of the SHA formalism but merely a visual expedient to represent edges entering and leaving the component.

While in $x_{\langle\text{op}\rangle}$, the dynamics of the system are constrained by differential equations marked by the generic symbol $\delta_{\langle\text{op}\rangle} \in \mathcal{F}(x_{\langle\text{op}\rangle})$. The symbol $i_{\langle\text{op}\rangle} \in \mathcal{I}(x_{\langle\text{op}\rangle})$ corresponds to the set of location-dependent invariants, in addition to constraint $t_{\text{upd}} \leq \mathsf{T}_{\text{poll}}$, which is common to all instances of ⟨op⟩_pub$_{\langle\text{id}\rangle}$. The dense counter representing the sensor reading is updated by instruction $\xi_{\langle\text{op}\rangle} \in \Xi(W)$. It is possible for one ⟨op⟩_pub$_{\langle\text{id}\rangle}$ instance to be in charge of publishing *multiple* sensor readings (e.g., the human shares data about fatigue and position). Thus, multiple dense-counter variables are simultaneously updated by $\xi_{\langle\text{op}\rangle}$.

6.2.1 Robotic Platform Model

As for the robot, there are three operating states corresponding to as many locations: r_{start}, r_{mov}, and r_{stop}. These are enclosed in as many ⟨op⟩_pub$_{\langle\text{id}\rangle}$ instances and represented as dashed boxes in Fig. 6.2 for ease of visualiza-

tion. Constraints within the dashed boxes in Fig. 6.2 represent flow condition and invariant associated with the corresponding $x_{\langle \mathsf{op} \rangle}$ location. On the other hand, locations r_{idle} and r_{rec} capturing the robotic platform being still or busy recharging, respectively, do not entail sensor data sharing and, as such, are not part of a $\langle \mathsf{op} \rangle$_pub$_{\langle \mathsf{id} \rangle}$ instance.

Real-valued variables V and Θ model the robot's velocity and orientation, respectively. As per the framework's assumption, velocity V evolves according to a trapezoidal profile [39], and it is, thus, constrained by the three flow conditions in Eq.6.1. Each flow condition in Eq.6.1 models a phase of the velocity profile: acceleration, travel, and deceleration.

$$\dot{V} = \begin{cases} \mathsf{a}_{\mathsf{max}} & \text{if } x_{\langle \mathsf{op} \rangle} = \mathrm{r}_{\mathrm{start}} \\ 0 & \text{if } x_{\langle \mathsf{op} \rangle} = \mathrm{r}_{\mathrm{mov}} \\ -\mathsf{a}_{\mathsf{max}} & \text{if } x_{\langle \mathsf{op} \rangle} = \mathrm{r}_{\mathrm{stop}} \end{cases} \tag{6.1}$$

Real-valued variable Θ models the orientation of the robot with respect to the x-axis, and the two locations $r_{\mathrm{turn_l}}$ and $r_{\mathrm{turn_r}}$ model the case in which the robot is rotating *left* or *right* with constant speed $\omega_{\mathsf{r}} \in K$. Variable Θ varies according to flow conditions $\dot{\Theta} = \omega_{\mathsf{r}}$ (in $r_{\mathrm{turn_l}}$) or $\dot{\Theta} = -\omega_{\mathsf{r}}$ (in $r_{\mathrm{turn_r}}$) until the desired orientation is reached. Parameters $\mathsf{v}_{\mathsf{max}}, \mathsf{a}_{\mathsf{max}}$, and ω_r are as introduced in Section 5.1.

Update instructions $\xi_{\langle \mathsf{start} \rangle}$, $\xi_{\langle \mathsf{mov} \rangle}$, and $\xi_{\langle \mathsf{stop} \rangle} \in \Xi(W)$ (also in Fig. 6.2) specific to the robot are made explicit in Eq.6.2. The Cartesian coordinates of the robot inside the building are measured and periodically updated through the dense-counter variables r_{pos_x} and r_{pos_y}.

$$\xi_{\langle \mathsf{start} \rangle, \langle \mathsf{mov} \rangle, \langle \mathsf{stop} \rangle} : \begin{cases} r'_{pos_x} = r_{pos_x} + V \mathsf{T}_{\mathsf{poll}} cos(\Theta) \\ r'_{pos_y} = r_{pos_y} + V \mathsf{T}_{\mathsf{poll}} sin(\Theta) \end{cases} \tag{6.2}$$

While in r_{mov} (within $\langle \mathsf{mov} \rangle$_pub$_{\langle 1 \rangle}$), every $\mathsf{T}_{\mathsf{poll}}$ seconds the automaton updates dense-counter variable $\theta \in V_{\mathrm{dc}}$, which represents the new set-point for orientation Θ. To make the path as smooth as possible, the robot starts turning only if the set-point θ is greater than a threshold $\theta_{\mathsf{th}} \in K$. As in Fig. 6.2, if $|\theta| > \theta_{\mathsf{th}}$ holds, the robot switches from $r_{\langle \mathsf{mov} \rangle}$ to $r_{\mathrm{turn_l}}$ or $r_{\mathrm{turn_r}}$ depending on whether $\Theta \lesseqgtr \theta$ holds. Finally, the robot switches back to r_{mov} and publishes the updated position values by sending an event through channel $\mathtt{p}_1$ (label $\mathtt{p}_1!$ in Fig. 6.2).

The robot starts or stops moving through controllable switches, labeled by channels $\mathtt{cmd_r}_{\mathrm{start}} \in C$ and $\mathtt{cmd_r}_{\mathrm{stop}} \in C$ (see Fig. 6.2).

Switch from r_{start} to r_{mov} and from r_{stop} to r_{idle} are uncontrollable as they capture the end of the acceleration and deceleration phases, respec-

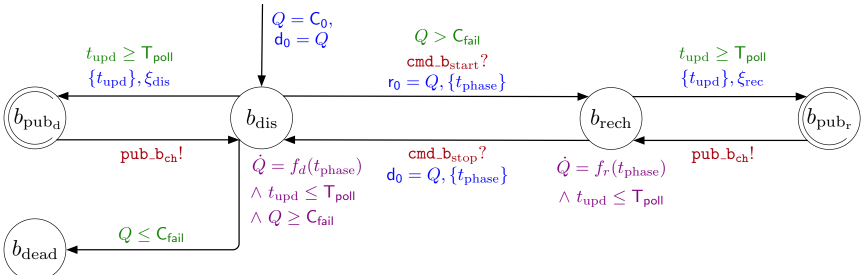


Figure 6.3: *SHA modeling the robot's battery.*

tively. Invariant $V \leq \mathsf{v}_{\mathsf{max}}$ on r_{start} (visible within $\langle$start$\rangle$_pub$_{\langle 1 \rangle}$) and the guard condition $V \geq \mathsf{v}_{\mathsf{max}}$ on the edge to r_{mov} ensure that the switch occurs exactly when velocity V equals $\mathsf{v}_{\mathsf{max}}$. Similar observations can be made about invariant $V \geq 0$ on r_{stop} and guard $V \leq 0$ on the edge to r_{idle}.

6.2.2 Battery Model

Mobile robots are typically powered by a lithium battery, which undergoes *charging* and *discharging* cycles. Therefore, the SHA $\mathcal{A}_{\text{b}}$ modeling the robot's battery features two ordinary non-committed locations b_{dis} and b_{rech} corresponding to the discharge and recharge phases, respectively, plus a deadlock location b_{dead} capturing the battery being fully discharged.

The main physical attribute of a battery is its voltage (representing the charge level), which is modeled by the real-valued variable Q. Variable Q is initialized with the initial voltage value C_0, i.e., the attribute of class Battery introduced in Section 5.1. The temporal dynamics of Q is determined by flow conditions in $\mathcal{F}(b_{\text{dis}})$ and $\mathcal{F}(b_{\text{rech}})$, whose integral is shown in Eq.6.3 and Eq.6.4, respectively.

Flow conditions match the behavior of lithium batteries for real robotic devices. As a matter of fact, the entire discharge cycle (from 100% of the voltage capacity to 0%) can be approximated by an exponential curve [211]. Nevertheless, the real device is not operational when the voltage drops below a certain threshold (which can vary depending on the specific battery type and the device it is powering), i.e., when the level of charge is not sufficient to power the wheel motors.

Letting the battery pack discharge to very low levels (close to 0%) may permanently damage it [41]. Therefore, we identify equations governing the evolution of variable Q (shown in Eq.6.3 and Eq.6.4) by fitting the discharge/charge curve when the robotic device is operative and can carry out the assigned mission. A cubic function showed a high fit to the real dynam-

ics. Parameters $\mathsf{d}_{0,1,2,3}, \mathsf{r}_{0,1,2,3} \in K$ determining the discharge and recharge curves are fitted based on sensor measurements collected during charge/discharge cycles of the same robotic device. Parameter d_0 is always set to C_0 at the beginning of the scenario.

$$Q(t) = -\mathsf{d}_3 t^3 - 2\mathsf{d}_2 t^2 - \mathsf{d}_1 t + \mathsf{d}_0 \tag{6.3}$$

$$Q(t) = \mathsf{r}_3 t^3 + 2\mathsf{r}_2 t^2 + \mathsf{r}_1 t + \mathsf{r}_0 \tag{6.4}$$

Edges between b_{dis} and b_{rech} model controllable switches, triggered when the orchestrator fires an event through channels $\texttt{cmd_b}_{\mathrm{start}}$ and $\texttt{cmd_b}_{\mathrm{stop}}$ instructing the robot to start or stop recharging. On the other hand, the switch from b_{dis} to b_{dead} is uncontrollable as it occurs when $Q = \mathsf{C}_{\mathsf{fail}}$, due to invariant $Q \geq \mathsf{C}_{\mathsf{fail}}$ on b_{dis} and guard $Q \leq \mathsf{C}_{\mathsf{fail}}$ on the edge to b_{dead}. The edge from b_{dis} to b_{rech} is also guarded by $Q > \mathsf{C}_{\mathsf{fail}}$ since the automaton must enter deadlock location b_{dead} when $Q = \mathsf{C}_{\mathsf{fail}}$ holds. The switches between b_{dis} and b_{rech} are equipped with updates that initialize the terms d_0 and r_0 of the equations Eq.6.3 with the residual charge values from the phase just ended that is $\mathrm{b}_{\mathrm{chg}}$ and reset clock t_{phase}.

The battery model features dense counter $\mathrm{b}_{\mathrm{chg}}$, representing the digital counterpart of Q, and a modeling pattern to periodically publish the latest charge measurement governed by clock t_{upd}. In $\mathcal{A}_{\mathrm{b}}$, this occurs while in b_{dis} and b_{rech} by switching to committed locations $b_{\mathrm{pub_d}}$ and $b_{\mathrm{pub_r}}$, respectively, when $t_{\mathrm{upd}} = \mathsf{T}_{\mathsf{poll}}$ holds. Upon these switches, clock t_{upd} is reset, to begin a new sensor refresh phase, and dense counter $\mathrm{b}_{\mathrm{chg}}$ is updated through updates ξ_{dis} and ξ_{rec}. The new value of the battery charge depends on how many phases lasting $\mathsf{T}_{\mathsf{poll}}$ time units have been executed so far, hence how many measurements have been collected.

For this reason, automaton $\mathcal{A}_{\mathrm{b}}$ features a dense counter $\mathrm{k} \in V_{\mathrm{dc}}$ that keeps track of the number of readings that have been done since the beginning of the scenario. Updates ξ_{dis} and ξ_{rech} compute the battery charge at the k-th refresh cycle $Q(\mathrm{k}\mathsf{T}_{\mathsf{poll}})$. They are obtained by expanding Eq.6.3 and Eq.6.4 when t is equal to $(\mathrm{k} - 1)\mathsf{T}_{\mathsf{poll}} + \mathsf{T}_{\mathsf{poll}}$.

Unlike the updates in automata modeling humans, dependency on index k cannot be removed in the equations defining $\mathrm{b}'_{\mathrm{chg}}$. At every sensor refresh, dense counter k at every sensor refresh is incremented. Updates are shown in Table 6.2 (where $\mathrm{b}'_{\mathrm{chg}}$ is the new value of $\mathrm{b}_{\mathrm{chg}}$ after the update). The updated value of $\mathrm{b}_{\mathrm{chg}}$ is then published by firing an event through channel $\texttt{pub_b}_{\texttt{ch}}$.

Table 6.2: *Updates for the $\mathcal{A}_{\mathrm{b}}$ SHA modeling the robot's battery.*

Symbol	Updates	Description
ξ_{dis}	$\mathrm{b}'_{\mathrm{chg}} = \mathrm{b}_{\mathrm{chg}} - \mathsf{T}_{\mathsf{poll}}((\mathsf{d}_3 + 6\mathsf{d}_3\mathrm{k})\mathsf{T}_{\mathsf{poll}}{}^2 + (\mathsf{d}_2 + 2\mathsf{d}_2\mathrm{k})\mathsf{T}_{\mathsf{poll}} + \mathsf{d}_1); \mathrm{k} = \mathrm{k} + 1;$	Discharge phase
ξ_{rec}	$\mathrm{b}'_{\mathrm{chg}} = \mathrm{b}_{\mathrm{chg}} + \mathsf{T}_{\mathsf{poll}}((\mathsf{r}_3 + 6\mathsf{r}_3\mathrm{k})\mathsf{T}_{\mathsf{poll}}{}^2 + (\mathsf{r}_2 + 2\mathsf{r}_2\mathrm{k})\mathsf{T}_{\mathsf{poll}} + \mathsf{r}_1); \mathrm{k} = \mathrm{k} + 1;$	Recharge phase

6.3 Human Behavior Modeling Approach

In all interaction patterns, the SHA modeling humans differentiate between operating states based on how fatigue evolves (i.e., whether individuals are recovering or not) and how humans are interacting with the robot (e.g., they are leading the action or waiting for a robot's action). Hence, all SHA modeling humans feature real-valued variable $F \in W$, capturing physical fatigue, and a dense counter $\mathrm{f} \in V_{\mathrm{dc}}$ capturing the digital counterpart of F. Besides physical fatigue, for each human, suitable sensors also periodically refresh their position within the building. The position is modeled by dense counters $\mathrm{h}_{\mathrm{pos}_\mathrm{x}}$ and $\mathrm{h}_{\mathrm{pos}_\mathrm{y}}$ capturing a pair of Cartesian coordinates. Therefore, the portion of SHA modeling the update of periodic sensor readings is present in all the operating states of the human, generically indicated as op, and is hereinafter referred to as $\langle \mathsf{op} \rangle_\mathrm{pub}$. In the following sections, for a clock $t \in X$, we use notation $\{t\}$ to represent update $t = 0$ (e.g., we write $\{t_{\mathrm{upd}}\}$ instead of $t_{\mathrm{upd}} = 0$).

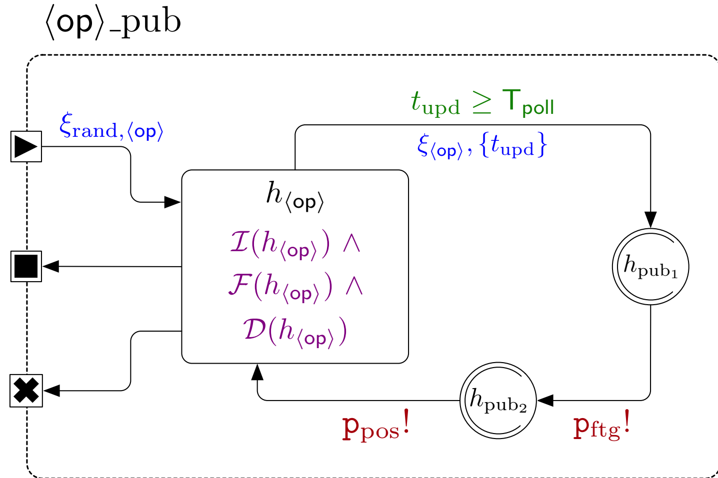


Figure 6.4: *SHA modeling the $\langle \mathsf{op} \rangle_\mathrm{pub}$ pattern, color-coded as in Fig. 3.1. Ports are marked by symbols "▶", "■", and "×".*

The $\langle \mathsf{op} \rangle_\mathrm{pub}$ pattern (i.e., a version of $\langle \mathsf{op} \rangle_\mathsf{pub}_{\langle \mathsf{id} \rangle}$ extended with features specifically tailored to modeling human behavior) is shown in Fig. 6.4. In the following, we use label op when describing the high-level structure of the pattern, while it is replaced by descriptive labels when referring to a specific instance of the pattern (e.g., $\langle \mathsf{stand} \rangle_\mathrm{pub}$ and $\langle \mathsf{walk} \rangle_\mathrm{pub}$). The automaton features three locations: an ordinary location $h_{\langle \mathsf{op} \rangle}$ and two committed locations h_{pub_1} and h_{pub_2}. Location $h_{\langle \mathsf{op} \rangle}$ captures the human's behavior while in a specific state (e.g., $h_{\langle \mathsf{stand} \rangle}$ and $h_{\langle \mathsf{walk} \rangle}$). To this end, $h_{\langle \mathsf{op} \rangle}$ is endowed with invariants $\mathcal{I}(h_{\langle \mathsf{op} \rangle})$, flow conditions $\mathcal{F}(h_{\langle \mathsf{op} \rangle})$, and probability distributions $\mathcal{D}(h_{\langle \mathsf{op} \rangle})$.

For all instances of $\langle \mathsf{op} \rangle_\mathrm{pub}$, $(t_{\mathrm{upd}} \leq \mathsf{T}_{\mathsf{poll}}) \in \mathcal{I}(h_{\langle \mathsf{op} \rangle})$ holds. The combination of this invariant with condition $t_{\mathrm{upd}} \geq \mathsf{T}_{\mathsf{poll}}$ on the edge to h_{pub_1} forces the SHA to switch to the committed location when $t_{\mathrm{upd}} = \mathsf{T}_{\mathsf{poll}}$ holds. Upon switching, the set of updates $\xi_{\langle \mathsf{op} \rangle} \subset \Xi(W)$ (e.g., $\xi_{\langle \mathsf{stand} \rangle}$ and $\xi_{\langle \mathsf{walk} \rangle}$) updates dense counters f, $\mathrm{h}_{\mathrm{pos}_\mathrm{x}}$, and $\mathrm{h}_{\mathrm{pos}_\mathrm{y}}$. The effect of $\xi_{\langle \mathsf{op} \rangle}$ varies depending on the specific state of the human. Since h_{pub_1} and h_{pub_2} are committed, the new values are immediately shared with the orchestrator by firing an event through channels $\mathsf{p}_{\mathrm{ftg}}$ first and $\mathsf{p}_{\mathrm{pos}}$ right after.

Edges entering and leaving the $\langle \mathsf{op} \rangle_\mathrm{pub}$ pattern are represented through *ports* (coherently with [128, 130]). SHA *enter* a submachine through the port marked by the symbol "▶" (i.e., *start*) and *leave* a submachine through ports marked by symbols "■" (i.e., *end*) and "×" (i.e., *fail*), indicating whether the operating state op ended (or stopped momentarily) or the entire mission ended with failure (e.g., because the human is too fatigued), respectively.

Edge conditions, channels, and updates characterizing edges through ports vary depending on the specific $\langle \mathsf{op} \rangle_\mathrm{pub}$ instance. The only exception is update $\xi_{\mathrm{rand},\langle \mathsf{op} \rangle}$ on the edge through the start port. Update $\xi_{\mathrm{rand},\langle \mathsf{op} \rangle}$ is featured by *all* instances of $\langle \mathsf{op} \rangle_\mathrm{pub}$ since it determines the stochastic properties of human fatigue of the human while in a specific operating state $\langle \mathsf{op} \rangle$ and the way these properties are determined is the same for every instance of $\langle \mathsf{op} \rangle_\mathrm{pub}$.

Human fatigue is a complex phenomenon driven by a wide range of factors: our approach focuses on muscular fatigue due to physical strain. As discussed by Liu et al. [141], a muscle can be seen as a reservoir of motor units. When physical exertion is required, motor units progressively activate and eventually cause *fatigue* due to biochemical processes. The muscle can, then, *recover* from fatigue if it is put to rest [118, 141].

Our approach exploits the model proposed by Konz [81, 118], described by Eq.6.5, for which human action undergoes alternate fatigue and recovery

cycles, each one modeled by an exponential function. Fatigue and recovery are expressed by means of function parameters, called fatigue rates, which depend on several factors such as the age of the subject that the model represents, their health condition, etc.

Each cycle is associated with an index i uniquely identified, given time t, by function $j : \mathbb{R}_+ \to \mathbb{N}$ (thus, $i = j(t)$ holds). We indicate the timestamp at which cycle i ends by t_i. During both fatigue and recovery, fatigue $F(t)$ for cycle i depends on the residual value $F(\mathsf{t}_{i-1})$ from the previous cycle ended at time t_{i-1}. Parameters λ_i and ρ_i are the fatigue and recovery rates for cycle i.

$$F(t) = \begin{cases} 1 - (1 - F(\mathsf{t}_{i-1})) \cdot e^{-\lambda_i (t - \mathsf{t}_{i-1})} & \text{(fatigue)} \\ F(\mathsf{t}_{i-1}) \cdot e^{-\rho_i (t - \mathsf{t}_{i-1})} & \text{(recovery)} \end{cases} \tag{6.5}$$

Complete recovery occurs when $F(t) = 0$ holds, whereas condition $F(t) = 1$ models the case in which the muscle has reached the maximum level of *endurance*. Liu et al. [141] argue that the fatigue $F(t)$ can be seen as ratio $M_F(t)/\mathrm{M}_0$, where M_0 is the total amount of motor units, and $M_F(t)$ is the number of fatigued units at time t. Therefore, $F(t) = M_F(t)/\mathrm{M}_0 = 1$ holds when *every* unit composing a muscle is fatigued.

Running experiments on a pool of subjects has shown how a Normal distribution is a good fit to capture the variability of rates λ_i and ρ_i in the fatigue model [140]. Furthermore, the variability of the fatigue rates for an individual subject between different exertion cycles has been observed in [184]. The SHA modeling the human in a scenario embed this variability by means of probability distributions, as the automaton is not representative of a single specific individual. Still, it represents a set of subjects with similar physical characteristics. Therefore, we approximate the complexity of the fatigue phenomenon by considering each λ_i (resp., ρ_i) as a sample of distribution $\mathcal{N}(\mu_\lambda, \sigma_\lambda^2)$ (resp., $\mathcal{N}(\mu_\rho, \sigma_\rho^2)$), whose mean and variance depend on the fatigue profile that characterizes the class of humans under analysis.

By construction, every operating state of a human agent is associated with a specific fatigue profile, i.e., it is either a fatigue state or a recovery state. Hence, for every instance of $\langle \mathsf{op} \rangle$_pub function $\mathcal{D}(h_{\langle \mathsf{op} \rangle})$ is defined.

Upon entering an $\langle \mathsf{op} \rangle$_pub, update $\xi_{\mathrm{rand},\langle \mathsf{op} \rangle}$ computes the fatigue/recovery rate to be considered while the automaton is in location $h_{\langle \mathsf{op} \rangle}$. To this end, every automaton modeling a human features two dense counters $\lambda, \rho \in V_{\mathrm{dc}}$, which store the current fatigue/recovery rates. Every time update $\xi_{\mathrm{rand},\langle \mathsf{op} \rangle}$ is executed, it generates a new sample of $\mathcal{D}(h_{\langle \mathsf{op} \rangle})$ and assigns it to ρ, if $h_{\langle \mathsf{op} \rangle}$ is a recovery state, otherwise to λ. The sample is generated

through the Box-Müller algorithm [46]. Update $\xi_{\mathrm{rand},\langle\mathsf{op}\rangle}$ is given in Eq.6.6, where rate equals λ if $h_{\langle\mathsf{op}\rangle}$ is a fatigue state and ρ otherwise; $\mu_{\langle\mathsf{op}\rangle}$ and $\sigma_{\langle\mathsf{op}\rangle}$ are the mean and standard deviation of $\mathcal{D}(h_{\langle\mathsf{op}\rangle})$; u_1 and u_2 are independent realizations of uniform distribution $\mathcal{U}(0,1)$.

$$\xi_{\mathrm{rand},\langle\mathsf{op}\rangle}: \quad \mathsf{rate} \;=\; \mu_{\langle\mathsf{op}\rangle} + \sigma_{\langle\mathsf{op}\rangle}\sqrt{-2\ln(u_2)}\cos(2\pi u_1) \tag{6.6}$$

The values of ρ and λ determine the temporal evolution of the real-valued variable F and its digital counterpart f while the automaton is in $h_{\langle\mathsf{op}\rangle}$.

In a given operating state $\langle\mathsf{op}\rangle$, if fatigue increases, flow condition $\mathcal{F}(h_{\langle\mathsf{op}\rangle})$ corresponds to the derivative of Eq.6.5(fatigue), indicated as f_{ftg} in Eq.6.7; otherwise, fatigue decreases and $\mathcal{F}(h_{\langle\mathsf{op}\rangle})$ is equal to the derivative of Eq.6.5 (recovery), indicated as f_{rec} in Eq.6.8.

Both equations depend on two terms other than ρ and λ, i.e., clock $t_{\mathrm{phase}} \in X$ and dense counter $\mathrm{F_p} \in V_{\mathrm{dc}}$. Clock $t_{\mathrm{phase}} \in X$ measures the total amount of time the automaton spends in location $h_{\langle\mathsf{op}\rangle}$ and dense counter $\mathrm{F_p} \in V_{\mathrm{dc}}$ is the residual value of fatigue at the end of the previous fatigue/recovery cycle, realized by a different $\langle\mathsf{op}\rangle$_pub instance. Both are updated when a new fatigue/recovery cycle begins, i.e., every time the SHA modeling the human enters an instance of $\langle\mathsf{op}\rangle$_pub and $\xi_{\mathrm{rand},\langle\mathsf{op}\rangle}$ is carried out: clock t_{phase} is reset and variable $\mathrm{F_p}$ is updated with F.

$$\dot{F} = f_{\mathrm{ftg}}(t_{\mathrm{phase}}, \lambda) = \mathrm{F_p}\lambda e^{-\lambda t_{\mathrm{phase}}} \tag{6.7}$$

$$\dot{F} = f_{\mathrm{rec}}(t_{\mathrm{phase}}, \rho) = -\mathrm{F_p}\rho e^{-\rho t_{\mathrm{phase}}} \tag{6.8}$$

On the other hand, dense counter f is not associated with a flow in location $h_{\langle\mathsf{op}\rangle}$ because it models the digital equivalent of the physical attribute F. For this reason, the temporal evolution of f is calculated explicitly via the update $\xi_{\langle\mathsf{op}\rangle}$, which computes a new value for f by applying the update in Eq.6.9, every $\mathsf{T_{poll}}$ time units. The primed version f' indicates the new value of f after the computation of the expression, which depends on the operating state $\langle\mathsf{op}\rangle$.

Unlike Eq.6.5, the equations in Eq.6.9 model fatigue in a single cycle and are expressed in terms of the amount of time elapsed from the beginning of the current fatigue/recovery cycle. Conversely, Eq.6.5 depends on the absolute time t and instant $\mathtt{t}_{i-1}$, the latter indicating the end of the cycle that precedes the current one. Hence, if τ is the amount of time elapsed from the beginning of a cycle, Eq.6.5 can be rewritten in terms of τ by applying the identity $t - \mathtt{t}_{i-1} = \tau$, and fatigue after τ time units from the

beginning of the current fatigue/recovery cycle is $\bar{F}(\tau) = F(\mathtt{t}_{i-1} + \tau)$. For $\tau = 0$, fatigue $\bar{F}(0)$ is equal to the residual value $F(\mathtt{t}_{i-1})$, which is $\mathrm{F_p}$.

At the end of the $(k+1)$-th sensor refresh, lasting $\mathsf{T_{poll}}$ time units each, the fatigue is $F(k\mathsf{T_{poll}} + \mathsf{T_{poll}})$. The final expressions are obtained by considering that, before computing $\xi_{\langle \mathsf{op} \rangle}$, f amounts to $\mathrm{F_p} e^{-\rho k \mathsf{T_{poll}}}$, in case of recovery, and to $1 - (1 - \mathrm{F_p}) e^{-\lambda k \mathsf{T_{poll}}}$ otherwise (i.e., the fatigue after k refresh cycles).

$$\mathrm{f}' = \bar{F}(k\mathsf{T_{poll}} + \mathsf{T_{poll}}) = \begin{cases} 1 - (1 - \mathrm{F_p}) e^{-\lambda(k\mathsf{T_{poll}} + \mathsf{T_{poll}})} & = \\ \qquad 1 - (1 - \mathrm{F_p}) e^{-\lambda k \mathsf{T_{poll}}} e^{-\lambda \mathsf{T_{poll}}} & = \\ \qquad 1 - (1 - \mathrm{f}) e^{-\lambda \mathsf{T_{poll}}} & \text{(fatigue)} \\ \mathrm{F_p} e^{-\rho(k\mathsf{T_{poll}} + \mathsf{T_{poll}})} & = \\ \qquad \mathrm{F_p} e^{-\rho k \mathsf{T_{poll}}} e^{-\rho \mathsf{T_{poll}}} & = \\ \qquad \mathrm{f} e^{-\rho \mathsf{T_{poll}}} & \text{(recovery)} \end{cases} \tag{6.9}$$

The introduction of distribution $\mathcal{D}(h_{\langle \mathsf{op} \rangle})$, $\xi_{\langle \mathsf{op} \rangle}$, and $\xi_{\mathrm{rand}, \langle \mathsf{op} \rangle}$ in all $\langle \mathsf{op} \rangle_\mathrm{pub}$ instances strengthens the results obtained with SMC as they account not only for the uncertainty due to human autonomy but also for the natural variability of the fatigue phenomenon. The extension, therefore, leads to more reliable estimations of the fatigue levels reached by subjects involved in the scenario, including an estimation of their variability ranges.

In the following, we present the individual SHA modeling the three interaction patterns, all featuring multiple instances of the hereby presented $\langle \mathsf{op} \rangle_\mathrm{pub}$ pattern.

6.3.1 Human Follower

An instance of the SHA modeling the human follower pattern is generated for each service specified through the DSL with $\mathsf{ptrn} = \mathsf{HumanFollower}$. The SHA, hereinafter referred to as $\mathcal{A}_{\mathrm{hf}}$ and shown in Fig. 6.5, features two instances of the $\langle \mathsf{op} \rangle_\mathrm{pub}$ pattern: one capturing the recovery phase while standing ($\langle \mathsf{stand} \rangle_\mathrm{pub}$) and one for the fatigue phase while walking ($\langle \mathsf{walk} \rangle_\mathrm{pub}$).

Fatigue decreases while resting (in $\langle \mathsf{stand} \rangle_\mathrm{pub}$) and increases while walking (while in $\langle \mathsf{walk} \rangle_\mathrm{pub}$). Therefore, $\mathcal{F}(h_{\langle \mathsf{stand} \rangle})$ equals $f_{\mathrm{rec}}(t, \rho)$ (see Eq.6.8) and $\mathcal{F}(h_{\langle \mathsf{walk} \rangle})$ equals $f_{\mathrm{ftg}}(t, \lambda)$ (see Eq.6.7). Values ρ and λ are realizations of $\mathcal{N}(\mu_{\mathsf{stand}}, \sigma^2_{\mathsf{stand}})$ and $\mathcal{N}(\mu_{\mathsf{walk}}, \sigma^2_{\mathsf{walk}})$, respectively. Table 6.3 shows the internal updates, ξ_{stand} and ξ_{walk}, respectively, later described in detail. $\mathcal{A}_{\mathrm{hf}}$ also features a deadlock location h_{faint} capturing the case in which the human reaches complete exhaustion causing the failure of the

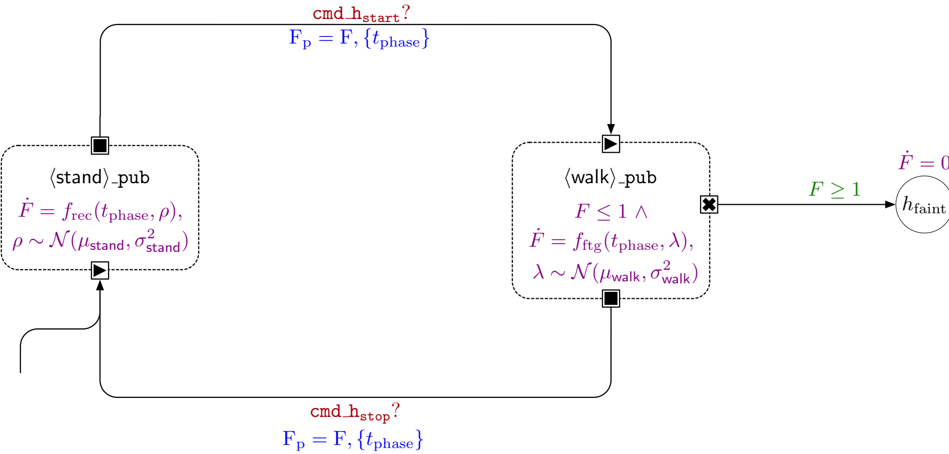


Figure 6.5: *SHA modeling human behavior when adhering to the* HumanFollower *pattern*

mission. If the mission fails because the human has reached location h_{faint}, modeling the evolution of fatigue is no longer relevant. Therefore, location h_{faint} is endowed with flow condition $\dot{F} = 0$.

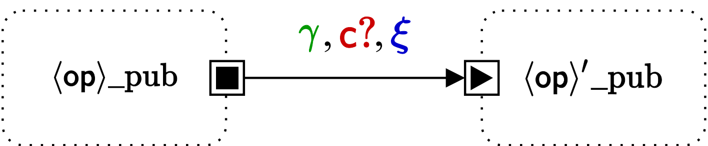
While walking (thus, while in location h_{walk}), the SHA periodically updates variables $\mathrm{h}_{\mathrm{pos}_\mathrm{x}}$ and $\mathrm{h}_{\mathrm{pos}_\mathrm{y}}$. As described in Section 5.1, we assume that humans walk at constant speed v. Dense counter h_γ captures the human's orientation with respect to the *x*-axis. We assume that humans can rotate instantly while following their trajectory; thus, no location is necessary to capture the delay caused by rotation.

Variable h_γ is updated while walking through function upd_orientation, which computes the new orientation required (primed dense counter h'_γ) to head towards the following point of the trajectory. Therefore, every $\mathsf{T}_{\mathsf{poll}}$ time instants, the *x*-*y* coordinates increase by $\mathsf{vT}_{\mathsf{poll}} \cos(\mathrm{h}_\gamma)$ along the *x*-axis and $\mathsf{vT}_{\mathsf{poll}} \sin(\mathrm{h}_\gamma)$ along the *y*-axis. While standing (in h_{stand}), as per Table 6.3, the human does not move, thus the values of $\mathrm{h}_{\mathrm{pos}_\mathrm{x}}$ and $\mathrm{h}_{\mathrm{pos}_\mathrm{y}}$ do not change. The periodic sensor refresh mechanism does not apply to location h_{faint} (which is not, thus, part of an $\langle$op$\rangle$_pub instance) since, once the mission has failed, the orchestrator no longer requires up-to-date sensor measurements.

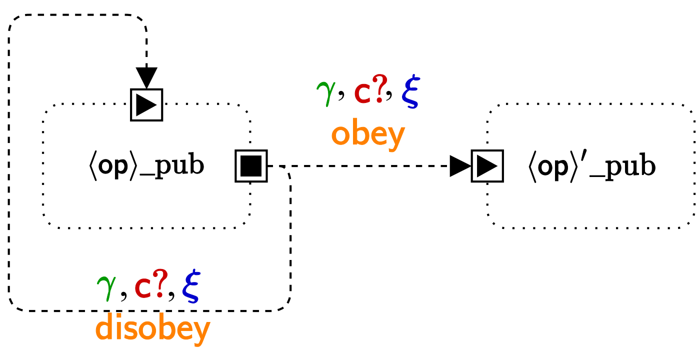
The switch between h_{stand} and h_{walk} (and vice versa) is controllable and triggered by events through channels $\texttt{cmd_h}_{\text{start}}$ and $\texttt{cmd_h}_{\text{stop}}$. The orchestrator sends to the SHA modeling the human events through these channels when it detects that the interaction between the human and the robot must start ($\texttt{cmd_h}_{\text{start}}$) or stop ($\texttt{cmd_h}_{\text{stop}}$). Upon switching between h_{stand} and

Table 6.3: *Updates for the SHA modeling the* HumanFollower *and* HumanLeader *patterns.*

Symbol	Updates	Description
ξ_{stand}	$\mathrm{f}' = \mathrm{f}\, e^{\rho \mathsf{T}_{\mathsf{poll}}}$; $\mathrm{fw}' = \mathsf{roll_dice}()$;	Resting phase
ξ_{walk}	$\mathrm{f}' = 1 - (1-\mathrm{f})\, e^{-\lambda \mathsf{T}_{\mathsf{poll}}}$; $\mathrm{h}'_{\gamma} = \mathsf{upd_orientation}()$; $\mathrm{h}'_{\mathrm{pos_x}} = \mathrm{h}_{\mathrm{pos_x}} + \mathsf{v}\mathsf{T}_{\mathsf{poll}} \cos(\mathrm{h}_{\gamma})$; $\mathrm{h}'_{\mathrm{pos_y}} = \mathrm{h}_{\mathrm{pos_y}} + \mathsf{v}\mathsf{T}_{\mathsf{poll}} \sin(\mathrm{h}_{\gamma})$; $\mathrm{fw}' = \mathsf{roll_dice}()$;	Walking phase



(a) *SHA modeling the non-erroneous behavior, with the solid line representing a deterministic edge.*



(b) Disobey/Obey *SHA add-on: as in Fig. 3.1, dashed lines represent probabilistic edges, with weights highlighted in orange.*

Figure 6.6: *SHA showing the standard behavior, and the* Disobey/Obey *add-on.*

h_{walk}, the SHA updates the value of variable $\mathrm{F_p}$ (see the updates on entering ⟨op⟩_pub instances).

To capture the unpredictability of human behavior, the edges between h_{stand} and h_{walk} and back are extended with features modeling human free will whose formal model is presented in the following and further detailed in Chapter 8.

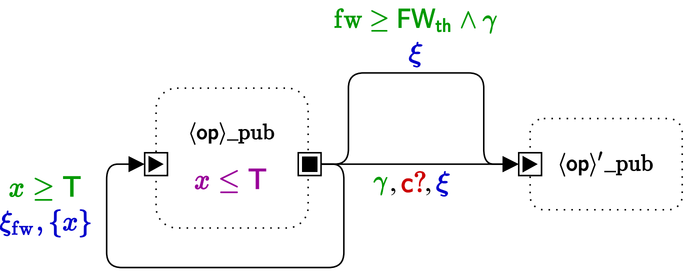
Disobey/Obey Add-On

The Disobey/Obey SHA add-on formally models the situation in which the orchestrator issues an instruction for the human, and the human ignores it (thus, "*disobeys*") and protracts the action they were previously performing. We do not further investigate or formally model whether ignoring the instruction is intentional since, as previously discussed, the formal model captures the *manifestation* of the erroneous behavior and not its cognitive source.

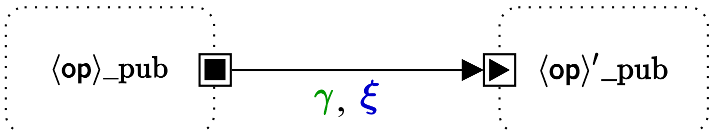
The standard behavior is captured by the SHA in Fig. 6.6a. Subautomaton $\langle \mathsf{op} \rangle_\mathrm{pub}$ represents the current state of the human (e.g., *standing* or *walking*), while $\langle \mathsf{op}' \rangle_\mathrm{pub}$ represents the following state in the sequence. The switch from $\langle \mathsf{op} \rangle_\mathrm{pub}$ to $\langle \mathsf{op}' \rangle_\mathrm{pub}$ occurs through an edge, which fires when an instruction is *sent* through channel $\mathsf{c} \in C$ (thus, the edge is labeled with $\mathsf{c}?$). Optionally, the edge may also be labeled with guard condition γ and update ξ.

Consider, for instance, the running example of the action sequence envisaged by the HumanFollower pattern, where the orchestrator fires an instruction through channel $\mathtt{cmd_h}_{\mathrm{start}}$ to instruct the human to start walking and follow the robot. In this case, the human acts *erroneously* as they do not abide by the instruction, which is captured by the SHA add-on in Fig. 6.6b. The deterministic edge from $\langle \mathsf{op} \rangle_\mathrm{pub}$ to $\langle \mathsf{op}' \rangle_\mathrm{pub}$ in Fig. 6.6a is changed into two *probabilistic* edges with weights $\mathsf{obey}, \mathsf{disobey} \in K$, and the same labels $\gamma, \mathsf{c}?$, and ξ as the original edge. The edge with weight obey reaches $\langle \mathsf{op}' \rangle_\mathrm{pub}$, thus capturing the human following the instruction and changing their state when $\mathsf{c}?$ fires. The edge with weight disobey is a self-loop on $\langle \mathsf{op} \rangle_\mathrm{pub}$, capturing the human ignoring the instruction and staying in the state modeled by $\langle \mathsf{op} \rangle_\mathrm{pub}$ when $\mathsf{c}?$ fires. The observed behavior when introducing this add-on is that the human performs the required action *when* instructed by the orchestrator with probability $\mathsf{p} = \mathsf{obey}/(\mathsf{obey} + \mathsf{disobey})$ and does *not* perform the required action with probability $1 - \mathsf{p}$.

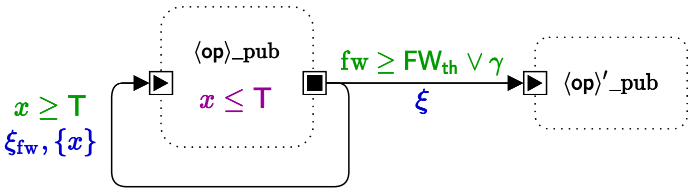
We remark that even if the SHA modeling the human takes the disobey edge, an event is still received through channel c due to label $\mathsf{c}?$. Nevertheless, the behavior of the SHA network that is *effectively* observed is that the SHA modeling the human does *not* initiate the action semantically associated with channel c. When reporting examples of erroneous action sequences (also for upcoming add-ons), we recall that the orchestrator checks the state of the system and issues one or multiple instructions, if necessary, every $\mathsf{T}_{\mathsf{int}} \in K$ time units.



(a) *Free Will SHA add-on corresponding to the standard behavior in Fig. 6.6a, when the action is initiated by the robot.*



(b) *SHA modeling the non-erroneous behavior, when the action is initiated by the human.*



(c) *Free Will SHA add-on when the action is initiated by the human.*

Figure 6.7: *Free Will SHA add-on.*

Free Will Add-On

The Free Will SHA add-on, shown in Fig. 6.7, captures the situation in which the human performs an action independently of the orchestrator's instructions (if the action is initiated by the robot) or when the system does not meet the pre-conditions for the actions (if the action is initiated by the human). In both cases, the manifestation of this erroneous behavior depends on dense counter fw that approximates the free will phenomenon through a random distribution [31] and whose underlying mechanism is explained below.

If the action is initiated by the robot, the planned behavior is shown in Fig. 6.6a: the orchestrator instructs the human to perform the next required action through channel c, triggering them to switch to $\langle \mathsf{op} \rangle$_pub. The SHA add-on modeling the erroneous behavior (shown in Fig. 6.7a) features an

additional edge between $\langle\mathsf{op}\rangle$_pub and $\langle\mathsf{op}'\rangle$_pub with update ξ, no channel label, and whose guard is a conjunction between the original guard γ and condition $\mathrm{fw} \geq \mathsf{FW_{th}}$ (explained in detail below). The purpose of the self-loop $\langle\mathsf{op}\rangle$_pub, for both cases, is also explained below contextually to the update of variable fw. The firing of this edge represents the human erroneously starting the action represented by channel c when no instruction has been issued by the orchestrator.

In the second case (i.e., the action initiated by the human), the standard behavior is shown in Fig. 6.7b. Subautomata $\langle\mathsf{op}\rangle$_pub and $\langle\mathsf{op}'\rangle$_pub represent the current and the next operational states. The switch between the two subautomata does not depend on robot instructions (there is no channel label on the edge) but is entirely up to the human to perform the action when a certain condition γ holds. The modeling approach assumes that no SHA other than the orchestrator fire an event through a channel (thus, with label $\mathsf{c}!$) representing the start of an action. The reason is that, in the real system, the human cannot *actively* signal the start of each action for the sake of practicality. When the human initiates an action, the orchestrator infers from sensor measurements that such an event occurred (e.g., that the human started moving because their position changed). The erroneous behavior, shown in Fig. 6.7c, captures the human potentially performing the action even if the required pre-conditions (represented by γ) do not hold, due to guard $\mathrm{fw} \geq \mathsf{FW_{th}} \vee \gamma$.

The mechanism determining free will, i.e., how new values are assigned to dense counter fw, is stochastic. Let $x \in X$ be a clock of the SHA the add-on is applied to, subautomaton $\langle\mathsf{op}\rangle$_pub is then endowed with invariant $x \leq \mathsf{T}$, where $\mathsf{T} \in K$ is a constant. Both in the Disobey/Obey and Free Will add-ons, subautomata $\langle\mathsf{op}\rangle$_pub and $\langle\mathsf{op}'\rangle$_pub may be endowed with further flow conditions, probability distributions, and invariants—i.e., $\mathcal{F}(h_{\langle\mathsf{op}\rangle})$, $\mathcal{D}(h_{\langle\mathsf{op}\rangle})$, $\mathcal{I}(h_{\langle\mathsf{op}\rangle})$, $\mathcal{F}(h_{\langle\mathsf{op}\rangle'})$, $\mathcal{D}(h_{\langle\mathsf{op}\rangle'})$, and $\mathcal{I}(h_{\langle\mathsf{op}\rangle'})$ can be non-empty. Nevertheless, since they do not directly impact the erroneous behavior like invariant $x \leq \mathsf{T}$, these labels are not shown in Fig. 6.6 nor Fig. 6.7 to ease the visualization of the add-ons' essential elements.

Subautomaton $\langle\mathsf{op}\rangle$_pub features a self-loop with guard $x \leq \mathsf{T}$ and update ξ_{fw}. The joint presence of the invariant and the guard condition enforces update ξ_{fw} to be executed every T time units. Simultaneously, clock x is reset (indicated as $\{x\}$) to ensure that the invariant holds after the self-loop fires. Update ξ_{fw} assigns a new value to dense counter $\mathrm{fw} \in V_{\mathrm{dc}}$. Specifically, the update yields a new sample of Uniform distribution $\mathcal{U}_{[0,\mathsf{FW_{max}})}$, where $\mathsf{FW_{max}} \in K$ is a numerical constant. Guard $\mathrm{fw} \geq \mathsf{FW_{th}}$ on the Free Will edge (in conjunction or disjunction with γ

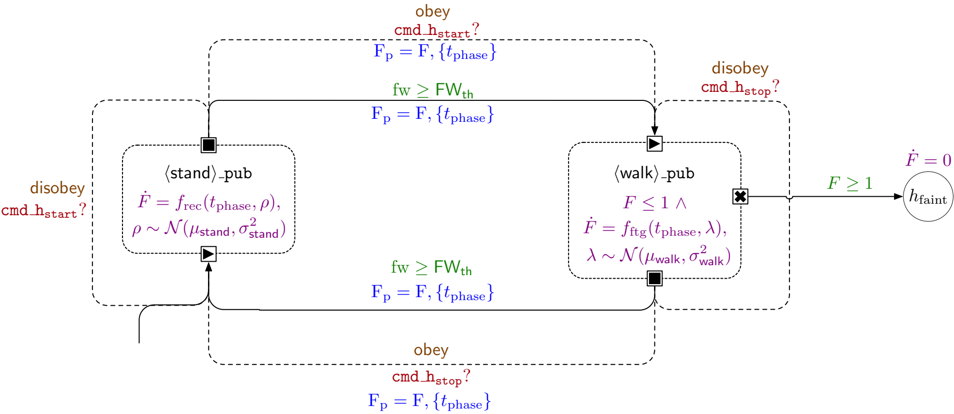


Figure 6.8: *SHA modeling human behavior when adhering to the* HumanFollower *pattern extended with the* Disobey/Obey *and* FreeWill *add-ons.*

in Fig. 6.7a and Fig. 6.7c, respectively) ensures that the erroneous behavior occurs only if the last value drawn for variable fw belongs to range $[\mathsf{FW_{th}}, \mathsf{FW_{max}})$ where $\mathsf{FW_{th}} \in K$ is a constant such that $\mathsf{FW_{th}} \leq \mathsf{FW_{max}}$ holds.

The HumanFollower extended with Disobey/Obey and FreeWill add-ons is shown in Fig. 6.8. In the following sections, the other interaction patterns are also presented directly in their extended version.

6.3.2 Human Leader

The SHA modeling the leader pattern, shown in Fig. 6.9, shares most features with the model described in Section 6.3.1. Locations h_{stand} and h_{walk} (within ⟨stand⟩_pub and ⟨walk⟩_pub) capture the human resting and walking constraining real-valued variable F through flow conditions in Eq.6.8 and Eq.6.7. While in these locations, sensor readings are periodically modified by updates ξ_{stand} and ξ_{walk} in Table 6.3. When fatigue exceeds the maximum threshold, the SHA switches to deadlock location h_{faint}.

The distinguishing feature of this pattern is that the switch from h_{stand} to h_{walk} is purely based on the free will mechanism and not on the orchestrator's instructions. The leader *autonomously* decides when to start the action. Therefore, the edge to h_{walk} is not tied to any event fired through any channel. Variable fw appearing in the guard condition is periodically randomly updated as described in Section 6.3.1. On the other hand, while the leader is free also to stop walking at any time irrespective of the robot's decisions (through the solid edge from h_{walk} to h_{stand}), the orchestrator may

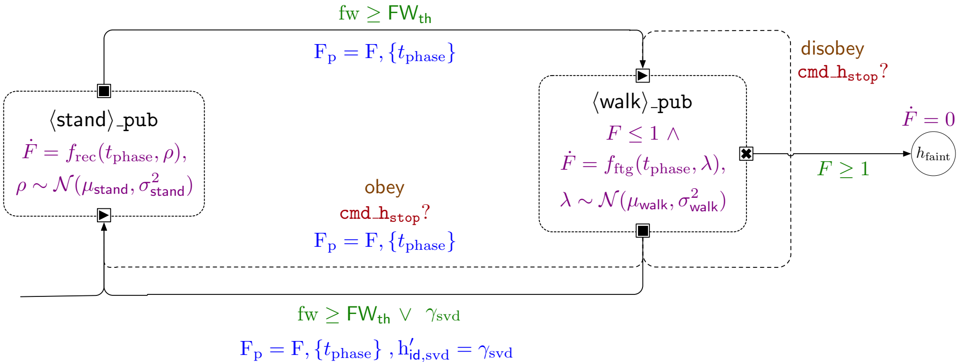


Figure 6.9: *SHA modeling the HumanLeader pattern.*

exceptionally instruct the human to stop walking through channel $\texttt{cmd_h}_{\text{stop}}$ when their fatigue reaches an alarming value. As with all other orchestrator commands, the edges triggered by such events are probabilistic and labeled with weights obey and disobey (see Fig. 6.9) governing whether the human abides by the instruction or ignores it and stays in the same location.

Finally, unlike in the follower pattern, the leader is in charge of declaring when the service is complete (thus, the robot may move on to serve the following human or stop if the mission is complete). The condition that determines whether the service is complete is indicated as γ_{svd} and corresponds to Formula 6.10 (see also Fig. 6.9). The service is considered complete if both the human and the robot are within a specific range of the destination, corresponding to attribute target of the class Service in Fig. 5.1. Dense counters $\mathrm{r}_{\text{pos}_\text{x}}$ and $\mathrm{r}_{\text{pos}_\text{y}}$ represent the Cartesian coordinates of the robot within the layout [128].

$$\sqrt{(\mathrm{h}_{\text{pos}_\text{x}} - \mathsf{target.x})^2 + (\mathrm{h}_{\text{pos}_\text{y}} - \mathsf{target.y})^2} \leq \mathsf{vT}_{\text{poll}} \wedge \qquad (6.10)$$
$$\sqrt{(\mathrm{h}_{\text{pos}_\text{x}} - \mathrm{r}_{\text{pos}_\text{x}})^2 + (\mathrm{h}_{\text{pos}_\text{y}} - \mathrm{r}_{\text{pos}_\text{y}})^2} \leq \mathsf{vT}_{\text{poll}}$$

As per Fig. 6.9, when condition γ_{svd} holds, dense counter $\mathrm{h}_{\mathsf{id,svd}}$ is updated to mark that the service is complete.

6.3.3 Human Recipient

The recipient pattern captures a human needing the robot to fetch an object and deliver it back to their current location. While the robot moves to the object's physical location (i.e., attribute target of the corresponding

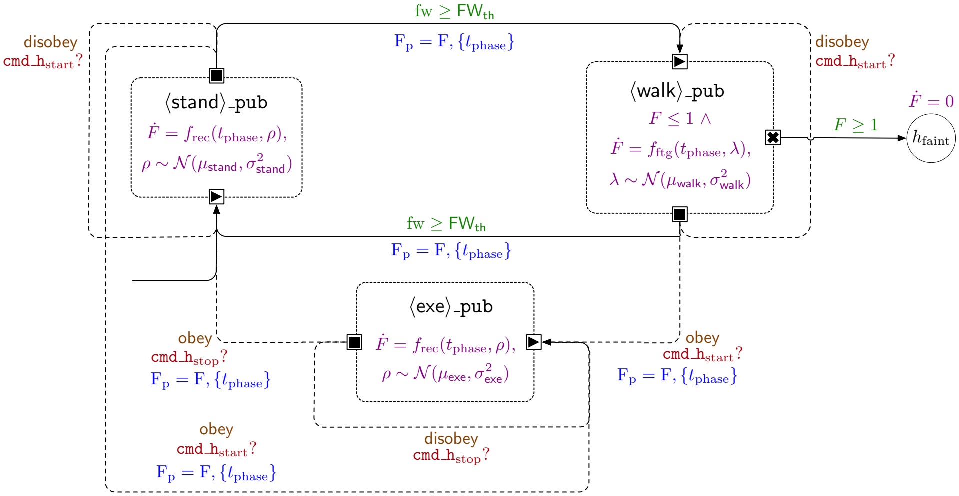


Figure 6.10: *SHA modeling the HumanRecipient pattern.*

Service) and travels back, the human is free to move around. Therefore, the SHA modeling human behavior for this pattern (shown in Fig. 6.10) features three operational states, corresponding to as many instances of the ⟨op⟩_pub pattern. Instance ⟨stand⟩_pub captures the human standing still, as described in Section 6.3.1 and Section 6.3.2.

Similarly, ⟨walk⟩_pub captures the human walking out of free will while waiting for the robot. Additionally, the recipient pattern features location h_{exe} (within pattern ⟨exe⟩_pub), representing that the robot has reached the human while carrying the object, and the human has to collect it. During the synchronization phase, neither the robot nor the human can move; thus $\mathcal{F}(h_{\langle exe \rangle})$ equals $f_{rec}(t, \rho)$. Ordinary location h_{faint} captures the human having reached the maximum fatigue level and, as in previously presented patterns, it is endowed with flow condition $\dot{F} = 0$.

While the robot is busy fetching the object, the human can autonomously decide to move at any time. Therefore, the edges from h_{stand} to h_{walk} and back depend on variable fw, which is periodically updated as described in Section 6.3.1. The robot triggers the synchronization with the human when it is ready to deliver the object by firing an event through channel $\texttt{cmd_h}_{start}$.

In this case, whether the human is walking (thus, in ⟨walk⟩_pub) or idle (in ⟨stand⟩_pub), they receive the instruction through channel $\texttt{cmd_h}_{start}$ to switch to ⟨exe⟩_pub for the synchronization phase. As with other SHA modeling human behavior, there is a certain probability that the human ignores the orchestrator's commands as dictated by weights obey and disobey.

The orchestrator gives the human time to pick up the object and then

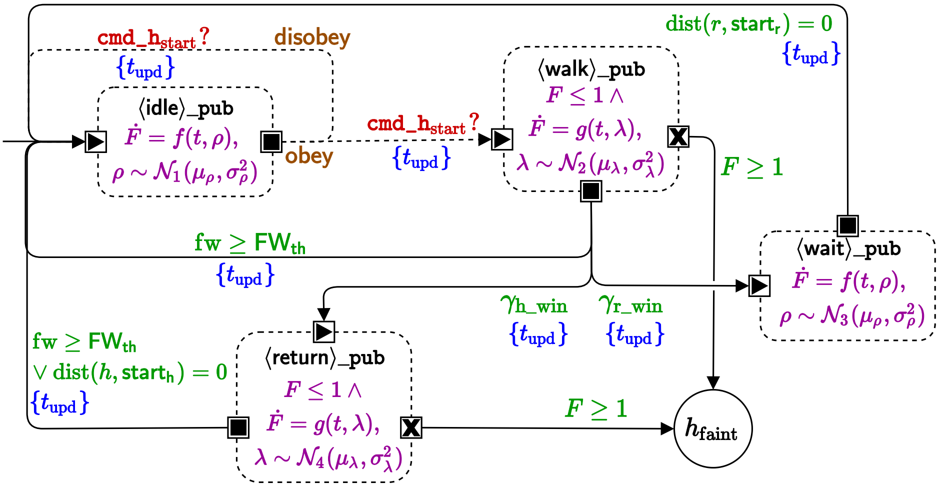


Figure 6.11: *SHA modeling the HumanCompetitor pattern, as seen in [130].*

fires an event through $\mathtt{cmd_h}_{\mathrm{stop}}$ to conclude the service, which the human may follow or ignore. On the other hand, no edge governed by variable fw enters or leaves ⟨exe⟩_pub since it would not capture "*rational*" behaviors. Such edge entering ⟨exe⟩_pub would capture the human collecting the object before the robot is sufficiently close. Similarly, a free-will edge leaving ⟨exe⟩_pub would capture the human deliberately suspending the synchronization phase, possibly dropping the item.

The possibility that the human still needs time to complete the synchronization after command $\mathtt{cmd_h}_{\mathrm{stop}}$ is issued by the robot (for example, if the item is particularly delicate or bulky) is modeled by the self-loop (i.e., the robot instructs the human to conclude the phase, but they ignore it and prolong the action).

6.3.4 Human Competitor

The pattern (shown in Fig. 6.11) captures a *competitive* type of interaction. Specifically, the human and the robot must fetch a resource from a specific physical location as quickly as possible. The location of the resource is captured by parameter dest and corresponds to attribute target of class Service presented in Section 5.1.

The human and the robot both move to the destination, and the winner returns to their initial physical location, captured by variables $\mathsf{start}_{\mathsf{h}}$ and $\mathsf{start}_{\mathsf{r}}$ representing pairs of Cartesian coordinates. Like the other SHAmodeling human behavior, this pattern also features the possibility that the hu-

man may autonomously decide to interrupt their current action.

The human starts in $\langle\mathsf{idle}\rangle$_pub. It switches to $\langle\mathsf{walk}\rangle$_pub when the orchestrator sends the instruction to start moving through channel $\mathtt{cmd_h}_{\mathrm{start}}$. As shown in Fig. 6.11, this edge is probabilistic: the human follows the orchestrator's instruction with probability $\mathrm{p} = \mathsf{obey}/(\mathsf{obey} + \mathsf{disobey})$ and ignores it with probability $1 - \mathrm{p}$.

Submachine $\langle\mathsf{walk}\rangle$_pub captures the situation in which the human is moving towards POI dest. Therefore, in this phase, fatigue increases and may reach the maximum threshold, resulting in the human being fully exhausted, which is modeled by deadlock location h_{faint}.

Human free will is also modeled through variable $\mathrm{fw} \in V_{\mathrm{dc}}$, which is updated with a random value every $\mathsf{T}_{\mathsf{poll}}$ time instants. When $\mathrm{fw} \geq \mathsf{FW}_{\mathsf{th}}$ holds, where $\mathsf{FW}_{\mathsf{th}}$ is a constant, the human autonomously decides to stop walking and switch back to $\langle\mathsf{idle}\rangle$_pub.

Guards $\gamma_{\mathrm{h_win}}$ (formalized by Formula (6.11)) and $\gamma_{\mathrm{r_win}}$ (Formula (6.12)) capture the "victory" conditions for the human and the robot, respectively. The human wins the competition if they reach destination dest before the robot and vice versa. If the agents arrive simultaneously, priority is given to the human since Formula (6.12) does not hold, but Formula (6.11) does.

$$\mathrm{dist}(h, \mathsf{dest}) = 0 \tag{6.11}$$

$$\mathrm{dist}(r, \mathsf{dest}) = 0 \wedge \mathrm{dist}(h, \mathsf{dest}) > 0 \tag{6.12}$$

If the robot arrives first, the human switches to submachine $\langle\mathsf{wait}\rangle$_pub while they wait for the robot to return to its initial position ($\mathrm{dist}(r, \mathsf{start_r}) = 0$ holds) and complete the interaction. Otherwise, the human switches to $\langle\mathsf{return}\rangle$_pub capturing them walking back to their initial physical location $\mathsf{start_h}$. In this case, the human may reach complete exhaustion or haphazardly decide to stop walking.

When the human returns to its initial physical location ($\mathrm{dist}(h, \mathsf{start_h}) = 0$ holds), the SHA switches back to submachine $\langle\mathsf{idle}\rangle$_pub if there are other humans to serve, or to o_{scs} if all humans in the scenario have been served and the mission has been completed.

6.3.5 Human Rescuer

The human rescuer pattern, shown in Fig. 6.12, captures the case in which the robot requires human support to perform an action, such as entering an elevator or crossing a closed door. Therefore, the human has to notice that the robot is requesting their assistance (e.g., through sound or visual signals), move to the robot's location, and perform the requested action.

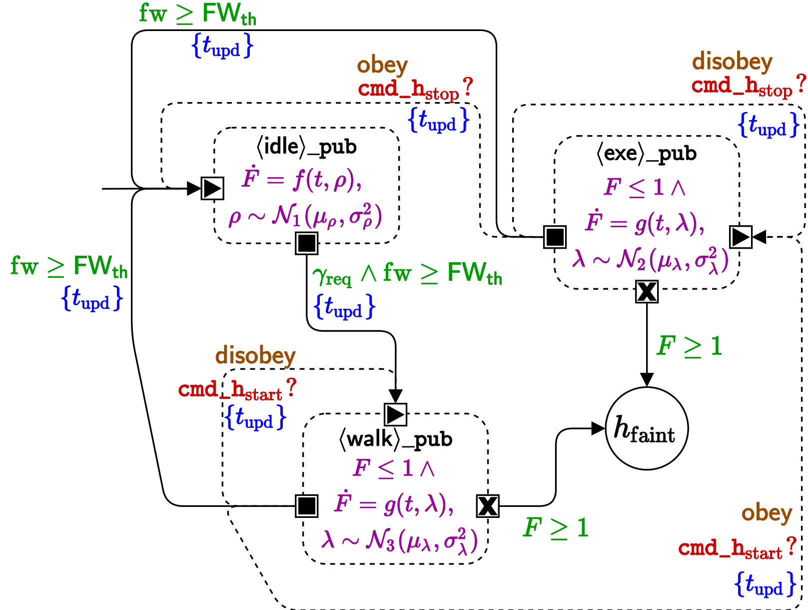


Figure 6.12: *SHA modeling the* HumanRescuer *pattern, as seen in [130].*

The human starts in the ⟨idle⟩_pub submachine, where it waits for the robot's request. Once the request signal is active (γ_{req} holds), the human autonomously decides when to start moving, which is captured by previously explained variable fw and explains why the outgoing edge from ⟨idle⟩_pub is not labeled with any channel.

When the human starts walking, the SHA switches to ⟨walk⟩_pub, capturing the phase in which they are approaching the robot. While walking, fatigue increases, thus, also, in this case, the SHA switches to deadlock location h_{faint} if $F \geq 1$ holds, causing the failure of the mission.

When the human is sufficiently close to the robot, the orchestrator sends an instruction through channel $\mathtt{cmd_h}_{start}$, which the human may follow or not with the same probabilities described for the competitor pattern. At any time, humans may stop walking out of free will.

If the human abides by the orchestrator's instruction, they switch to submachine ⟨exe⟩_pub to perform the required task.

The duration of the task, modeled by constant T_{task}, determines how long the SHA stays in ⟨exe⟩_pub. When configuring the scenario, the designer can specify constant parameter dext for the human that models their ability to perform the task and determines the time required to complete it. Dense-counter t_{task} measures the progress since the start of the action.

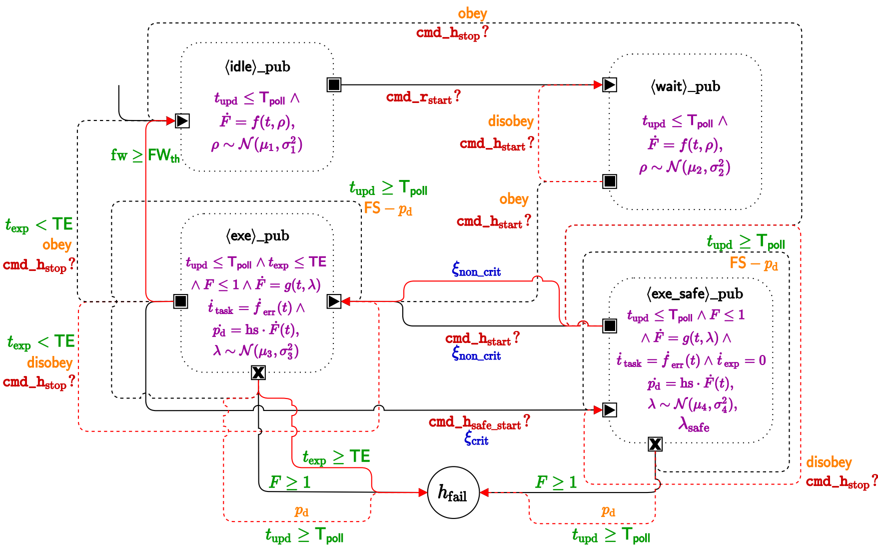


Figure 6.13: *SHA modeling the* HumanApplicant *pattern, as seen in [130].*

To capture the unpredictability of the human progressing through the action, every $\mathsf{T}_{\mathsf{poll}}$ instants, t_{task} increases with probability $(100 - \mathsf{T}_{\mathsf{th}})/100$ as per update instruction $\xi_{\langle \mathsf{exe} \rangle}$ (see Formula (6.13)), where $\mathsf{T}_{\mathsf{th}} \leq 100$ is a constant.

$$t_{\text{task}} = t_{\text{task}} + \mathsf{dext} * (\text{rand}(0, 100) \geq \mathsf{T}_{\mathsf{th}}) \tag{6.13}$$

Condition $t_{\text{task}} \geq \mathsf{T}_{\mathsf{task}}$ holds when the action is complete. Guard γ_{stop} (see Formula (6.14)) holds if either the action is done or the human has stopped executing it out of free will (i.e., system-wide variable $\text{h}_{\text{exe}} \in \text{V}_{\text{dc}}$ evaluates to false).

$$t_{\text{task}} \geq \mathsf{T}_{\mathsf{task}} \vee \neg \text{h}_{\text{exe}} \tag{6.14}$$

When γ_{stop} holds, the orchestrator instructs both the human and the robot to stop through the respective channels. If the human follows the instruction, the corresponding SHA enters ⟨idle⟩_pub.

6.3.6 Human Applicant

The applicant pattern captures the case in which the human requires the robot's support in performing an action. This covers applications such as a patient needing to be fed by the robot or a doctor needing the robot's help performing a medical test. In this case, the human waits for the robot to reach them, and then they start working on the task simultaneously.

The SHAmodeling the human, shown in Fig. 6.13, starts in submachine $\langle\mathsf{idle}\rangle$_pub. The orchestrator instructs the robot to start moving by firing an event through channel $\mathtt{cmd_r}_{\mathrm{start}}$, thus the human switches to subautomaton $\langle\mathsf{wait}\rangle$_pub, as they wait for the robot to reach their current location.

Once the robot reaches the human location, the pair begins the task; thus the human switches to submachine $\langle\mathsf{exe}\rangle$_pub. The instruction to start is sent through channel $\mathtt{cmd_h}_{\mathrm{start}}$, which the human can follow or ignore probabilistically. In reality, this could capture the potential additional delay due to the human requiring further preparation before they can start.

The task is modeled like in the rescuer pattern: its duration corresponds to constant $\mathsf{T}_{\mathsf{task}}$, and its current progress is measured by dense counter t_{task}, which is updated with a certain probability according to update instruction $\xi_{\langle\mathsf{exe}\rangle}$ of Formula (6.13). At any time, the human may autonomously decide to pause the action, which is approximated by the mechanism involving variable fw.

Like in the rescuer pattern, since actions captured by this pattern imply physical involvement on the human's side, fatigue increases while in $\langle\mathsf{exe}\rangle$_pub. Therefore, it features an outgoing edge to deadlock location h_{faint}. When γ_{stop} (see Formula (6.14)) holds, the orchestrator sends the instruction to stop, and the human concludes the interaction by switching back to submachine $\langle\mathsf{idle}\rangle$_pub.

6.4 Orchestrator Model

The orchestrator controls the robot's behavior based on the current state of the system to drive the mission to success. As described in previous sections, the humans, the robot, and its battery share sensor readings with the orchestrator, which checks these values against given policies to determine whether a particular event has to be fired. Specifically, the orchestrator is entirely in control of the mobile robot's behavior (i.e., it issues every instruction to start or stop moving), while it issues *suggestions* for the human, e.g., to stop moving when they reach an alarming value of fatigue, which might be dismissed due to human free will.

An abstract representation of the orchestrator SHA is shown in Fig. 6.14. The orchestrator operational states (the dashed boxes in Fig. 6.14) are modeled as submachines. All the edges connecting them are labeled with events of the form c! with $\mathsf{c} \in C$ as the orchestrator proactively triggers suitable actions to govern the evolution of the entire scenario.

The orchestrator's operational states, i.e., the submachines in it, are:

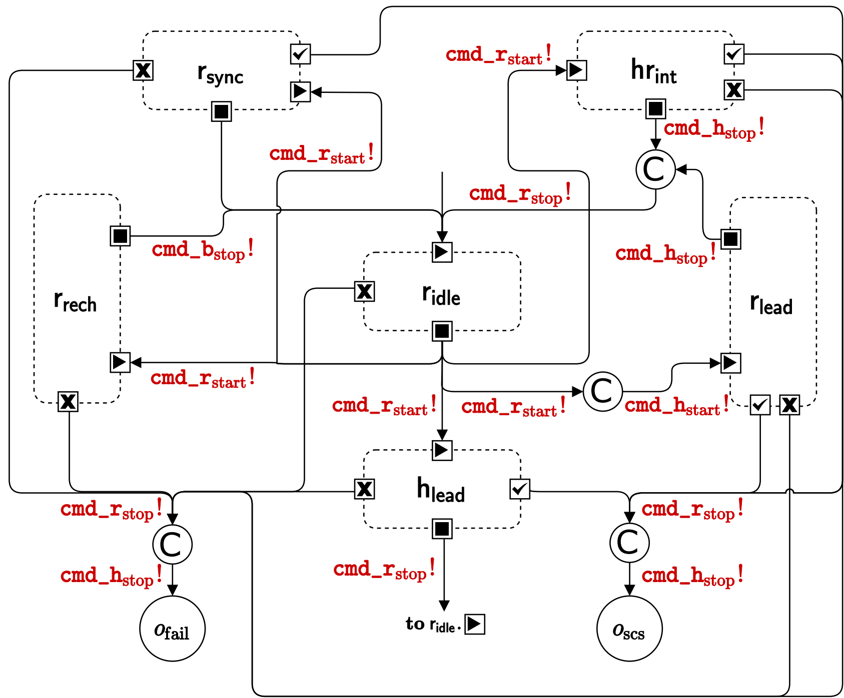


Figure 6.14: *High-Level representation of the SHA modeling the orchestrator, as seen in [130]. Submachines are represented as dashed boxes, with ports marked by symbols "▶", "■", "✓", and "×".*

1. $\mathsf{r}_{\mathsf{idle}}$: given the system's state, no action can start, and thus, the robot is waiting;
2. $\mathsf{r}_{\mathsf{rech}}$: the robot is moving to the recharge station or recharging;
3. $\mathsf{r}_{\mathsf{lead}}$: based on the interaction pattern characterizing the service underway, the robot *leads* the action;
4. $\mathsf{h}_{\mathsf{lead}}$: action is initiated by the human;
5. $\mathsf{hr}_{\mathsf{int}}$: the robot is providing a service that requires precise or close-distance synchronization with the human;
6. $\mathsf{r}_{\mathsf{sync}}$: the robot synchronizes with another robot in the fleet to hand over its current task.

The orchestrator relies on a recurring modeling pattern denominated as ⟨op⟩_chk and shown in Fig. 6.15. The ⟨op⟩_chk automaton portion is

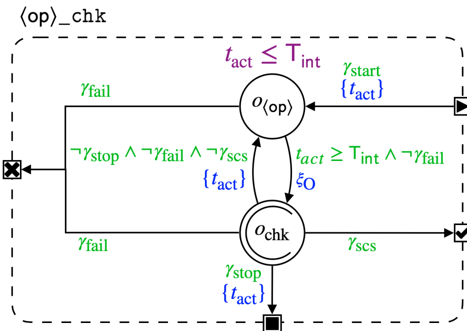


Figure 6.15: *SHA portion modeling the* ⟨op⟩_chk *pattern.*

necessary to *periodically* check the latest set of sensor readings against the orchestrator's policies and issue a command for the agents, if necessary.

Instances of ⟨op⟩_chk in Fig. 6.14 are endowed with *ports*, intended as in the ⟨op⟩_pub pattern. The orchestrator *enters* a submachine through the start port (marked by the symbol "▶"), and *exits* through the stop, fail, and scs ports (marked by symbols "■", "×", and "✓", respectively), indicating whether the action has ended (or is momentarily suspended), the mission has ended with failure or with success. Ports highlight the transitions entering and leaving each submachine, guarded by conditions γ_{start}, γ_{stop}, γ_{fail}, and γ_{scs}, each associated with a component-specific formula. The orchestrator enters a submachine when the corresponding γ_{start} condition is true. If either of γ_{stop}, γ_{fail}, or γ_{scs} holds, the orchestrator exits the submachine.

Location $o_{\langle \mathsf{op} \rangle}$ models the current operational state of the system, where ⟨op⟩ is a generic identifier. Committed location o_{chk} captures the orchestrator on the verge of either issuing a command (thus, exiting the submachine through one of the ports) or returning to $o_{\langle \mathsf{op} \rangle}$ if no instruction is required.

The orchestrator periodically switches from $o_{\langle \mathsf{op} \rangle}$ to o_{chk} every $\mathsf{T}_{\mathsf{int}}$ time instants, with the delay measured through clock t_{act}. Upon entering o_{chk}, the orchestrator checks the sensor readings against the component-specific policies through update instruction ξ_{O}. If either one of γ_{stop}, γ_{fail}, or γ_{scs} holds, the orchestrator exits the submachine; otherwise, it switches back to $o_{\langle \mathsf{op} \rangle}$.

Locations o_{fail} and o_{scs} of Fig. 6.14 correspond to the end of the mission with failure or success, respectively, and are reached when either γ_{fail} (see Formula 6.15) or γ_{scs} (see Formula 6.16) holds. Failure occurs if, for at least one of the subjects, human fatigue exceeds 1 (i.e., $\mathrm{f}_i \geq 1$ holds for some i) or one of the robots' charge drops to a neighborhood of $\mathsf{C}_{\mathsf{fail}}$ (i.e., $|\mathrm{b}_{\mathrm{j,chg}} - \mathsf{C}_{\mathsf{fail}}| \leq \epsilon$ holds for some j). The latter condition accounts for small

fluctuations in the estimated discharge curve.

$$\left(\bigvee_{i=1}^{\mathsf{N_h}} \mathrm{f}_i \geq 1\right) \vee \left(\bigvee_{j=1}^{\mathsf{N_r}} |\mathrm{b_{j,chg}} - \mathsf{C_{fail}}| \leq \epsilon\right) \tag{6.15}$$

$$\bigwedge_{i=1}^{\mathsf{N_h}} \mathrm{h_{i,svd}} \tag{6.16}$$

Location o_{scs} is reached when the mission has been successfully completed—i.e., when all humans in the scenario have been served: when a human in the scenario with $\mathsf{id} = i$ is served, boolean $\mathrm{h_{i,svd}}$ is set to true. We recall that the primary expression whose value we calculate through SMC is $\mathbb{P}_{\mathrm{M}}(\diamond_{\leq\tau}\ \mathsf{scs})$, where dense Boolean counter scs is set to true upon entering location o_{scs} (thus when the condition in 6.16 holds). As per Fig. 6.14, failure is possible for all submachines. On the other hand, only $\mathsf{r_{lead}}$, $\mathsf{h_{lead}}$, $\mathsf{r_{sync}}$, and $\mathsf{hr_{int}}$ have outgoing transitions towards o_{scs}, since recharging the robot does not impact service provision (thus, progress towards mission completion).

Table 6.4 contains the formulae for the start (γ_{start}) and stop (γ_{stop}) conditions of the submachines. Fig. 6.14 highlights the channels through which the orchestrator fires instructions when entering or leaving a submachine. For the sake of clarity, if a is a submachine, e.g., $\mathsf{r_{lead}}$, and g is the guard associated with the edge through a port, e.g., γ_{start}, then we refer to g by writing $a.g$.

6.4.1 $\mathsf{r_{idle}}$ Submachine

The first instance of the pattern is idle_chk in Fig. 6.16a, which models the situation in which the system is idle and periodically checks whether an action can start.

The system enters this component first when the execution starts and returns to it whenever an action stops (and the corresponding sub-component is left). Similarly, as per Table 6.4, the orchestrator exits this component if one of the γ_{start} conditions for the other sub-machines holds.

6.4.2 $\mathsf{r_{rech}}$ Submachine

The recharging routine in Fig. 6.16d starts when battery charge c is below a threshold $\mathsf{C_{rech}}$.

Submachine $\mathsf{r_{rech}}$ models two operating conditions: the robot's movement towards the charging station (move_chk), and the robot recharging its battery (rech_chk).

Table 6.4: *Orchestrator start and stop conditions (γ_{start} and γ_{stop}, respectively) for each submachine (sub.m.) of the orchestrator. Guard condition γ characterizing a submachine x is indicated with notation $\mathsf{x}.\gamma$ (e.g., $\mathsf{r}_{\mathsf{rech}}.\gamma_{\mathrm{start}}$).*

Sub.m.	**Start condition** (γ_{start})	**Stop condition** (γ_{stop})
$\mathsf{r}_{\mathsf{idle}}$	$\mathsf{r}_{\mathsf{rech}}.\gamma_{\mathrm{stop}} \vee \mathsf{r}_{\mathsf{lead}}.\gamma_{\mathrm{stop}} \vee \mathsf{h}_{\mathsf{lead}}.\gamma_{\mathrm{stop}} \vee \mathsf{hr}_{\mathsf{int}}.\gamma_{\mathrm{stop}} \vee \mathsf{r}_{\mathsf{sync}}.\gamma_{\mathrm{stop}}$	$\mathsf{r}_{\mathsf{rech}}.\gamma_{\mathrm{start}} \vee \mathsf{r}_{\mathsf{lead}}.\gamma_{\mathrm{start}} \vee \mathsf{h}_{\mathsf{lead}}.\gamma_{\mathrm{start}} \vee \mathsf{hr}_{\mathsf{int}}.\gamma_{\mathrm{start}} \vee \mathsf{r}_{\mathsf{sync}}.\gamma_{\mathrm{start}}$
$\mathsf{r}_{\mathsf{rech}}$	$\mathrm{b}_{\mathrm{chg}} \leq \mathsf{C}_{\mathsf{rech}}$	$\mathrm{b}_{\mathrm{chg}} \geq \mathsf{C}_{\mathsf{restart}}$
$\mathsf{r}_{\mathsf{lead}}$	$\mathrm{h}_{\mathrm{curr,ptrn}} \in \{\mathsf{follower}, \mathsf{recipient}, \mathsf{competitor}\} \wedge \neg \mathrm{h}_{\mathrm{curr,svd}} \wedge \mathrm{f}_{\mathrm{curr}} \leq \mathsf{F}_{\mathsf{restart}} \wedge \mathrm{b}_{\mathrm{chg}} \geq \mathsf{C}_{\mathsf{rech}} \wedge \mathrm{dist}(\mathrm{r}_{\mathrm{pos}}, \mathrm{h}_{\mathrm{pos}}) \leq \mathsf{D}_{\mathsf{restart}}$	$\mathrm{f}_{\mathrm{curr}} \geq \mathsf{F}_{\mathsf{stop}} \vee \mathrm{b}_{\mathrm{chg}} \leq \mathsf{C}_{\mathsf{rech}} \vee \left(\mathrm{h}_{\mathrm{curr,svd}} \wedge \bigvee_{i=1}^{\mathsf{N}_{\mathsf{h}}} \neg \mathrm{h}_{\mathrm{i,svd}}\right) \vee \mathrm{dist}(\mathrm{r}_{\mathrm{pos}}, \mathrm{h}_{\mathrm{curr,pos}}) \geq \mathsf{D}_{\mathsf{stop}}$
$\mathsf{h}_{\mathsf{lead}}$	$\mathrm{h}_{\mathrm{curr,ptrn}} \in \{\mathsf{leader}\} \wedge \neg \mathrm{h}_{\mathrm{curr,svd}} \wedge \mathrm{b}_{\mathrm{chg}} \geq \mathsf{C}_{\mathsf{rech}} \wedge \mathrm{h}'_{\mathrm{curr,pos}} \neq \mathrm{h}_{\mathrm{curr,pos}}$	$\mathrm{f}_{\mathrm{curr}} \geq \mathsf{F}_{\mathsf{stop}} \vee \mathrm{b}_{\mathrm{chg}} \leq \mathsf{C}_{\mathsf{low}} \vee \left(\mathrm{h}_{\mathrm{curr,svd}} \wedge \bigvee_{i=1}^{\mathsf{N}_{\mathsf{h}}} \neg \mathrm{h}_{\mathrm{i,svd}}\right) \vee \mathrm{h}'_{\mathrm{curr,pos}} = \mathrm{h}_{\mathrm{curr,pos}}$
$\mathsf{hr}_{\mathsf{int}}$	$\mathrm{h}_{\mathrm{curr,ptrn}} \in \{\mathsf{rescuer}, \mathsf{assistant}\} \wedge \neg \mathrm{h}_{\mathrm{curr,svd}} \wedge \mathrm{f}_{\mathrm{curr}} \leq \mathsf{F}_{\mathsf{restart}} \wedge \mathrm{b}_{\mathrm{chg}} \geq \mathsf{C}_{\mathsf{rech}}$	$\mathrm{f}_{\mathrm{curr}} \geq \mathsf{F}_{\mathsf{stop}} \vee \mathrm{b}_{\mathrm{chg}} \leq \mathsf{C}_{\mathsf{rech}} \vee \left(\mathrm{h}_{\mathrm{curr,svd}} \wedge \bigvee_{i=1}^{\mathsf{N}_{\mathsf{h}}} \neg \mathrm{h}_{\mathrm{i,svd}}\right) \vee \neg \mathrm{h}_{\mathrm{curr,exe}}$
$\mathsf{r}_{\mathsf{sync}}$	$\gamma_{\mathrm{handover}} \wedge \left(\bigvee_{j \in [1, \mathsf{N}_{\mathsf{r}}]} j \neq i \wedge \neg \mathrm{busy}_j \wedge \mathrm{b}_{\mathrm{j,chg}} \geq \mathsf{C}_{\mathsf{rech}}\right) \wedge \neg\left(\bigvee_{k \in [1, \mathsf{N}_{\mathsf{r}}]} (k \neq j \neq i \wedge \mathrm{dist}(\mathrm{r}_i, \mathrm{r}_k) < \mathrm{dist}(\mathrm{r}_i, \mathrm{r}_j)\right)$	$\mathrm{t}_{\mathrm{upd}} \geq \mathsf{T}_{\mathsf{sync}}$

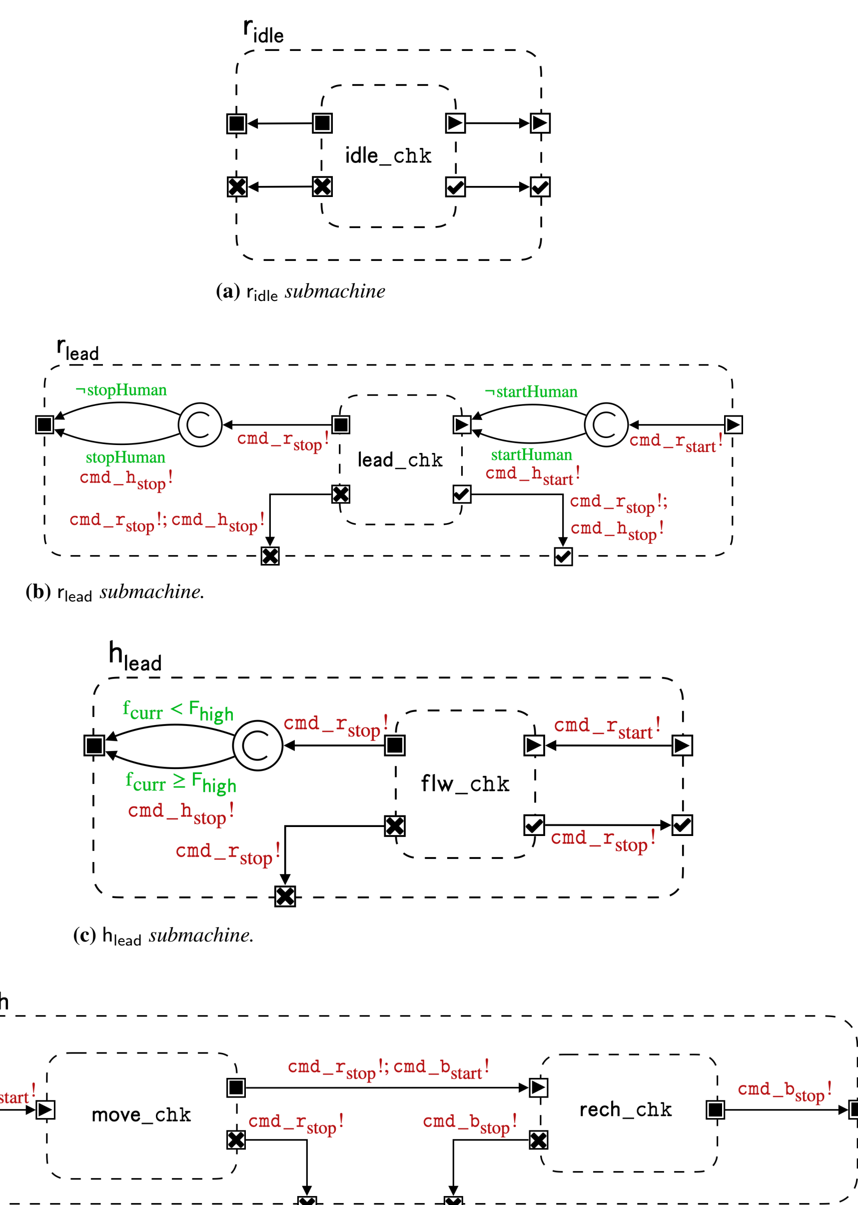


Figure 6.16: *Submachines of the orchestrator SHA.*

Upon entering $\mathsf{r}_{\mathsf{rech}}$, the orchestrator fires $\texttt{cmd_r}_{\text{start}}$ to instruct the robot to reach the charging station, then $\texttt{cmd_r}_{\text{stop}}$ when the dock has been reached. The robot can, thus, start recharging, and the orchestrator fires $\texttt{cmd_b}_{\text{start}}$.

The robot stops recharging ($\mathtt{cmd_b}_{\mathrm{stop}}$ is fired) when $\mathrm{b}_{\mathrm{chg}}$ is above threshold $\mathsf{C}_{\mathsf{restart}}$. In this case, the orchestrator switches back to $\mathsf{r}_{\mathsf{idle}}$.

6.4.3 $\mathsf{r}_{\mathsf{lead}}$ Submachine

The orchestrator enters submachine $\mathsf{r}_{\mathsf{lead}}$ (see Fig. 6.16b) to initiate the robot movement when the robot *leading* the action. As per Table 6.4, the start condition holds for the follower, recipient, and competitor patterns. The robot begins assisting the currently served human (indicated with index $\mathrm{curr} \in [1, \mathsf{N}_{\mathsf{h}}]$) if they are sufficiently close and the service has yet to be completed. For safety purposes, the action can start only if human fatigue is sufficiently low and the robot has sufficient charge.

Upon entering $\mathsf{r}_{\mathsf{lead}}$, the orchestrator fires $\mathtt{cmd_r}_{\mathrm{start}}$ and $\mathtt{cmd_h}_{\mathrm{start}}$ for the robot to start moving and the human to follow. The only operating condition modeled by this sub-component is the robot movement (component lead_chk in Fig. 6.16b). The robot stops moving (events $\mathtt{cmd_r}_{\mathrm{stop}}$ and $\mathtt{cmd_h}_{\mathrm{stop}}$ fire) if either one of the following conditions holds:

1. human fatigue f exceeds a maximum tolerable value $\mathsf{F}_{\mathsf{stop}}$;
2. battery charge drops below a value $\mathsf{C}_{\mathsf{rech}}$ that calls for recharging;
3. the human has been served, but they were not the last one (if they were the last one, the mission would be complete);
4. the distance between the robot and the human is too larger (greater than $\mathsf{D}_{\mathsf{stop}}$), indicating that the human has stayed behind and needs to get closer to the robot to proceed with the service.

6.4.4 $\mathsf{h}_{\mathsf{lead}}$ Submachine

The $\mathsf{h}_{\mathsf{lead}}$ submachine controls the robot's behavior when the human is leading the action (i.e., with the *leader* interaction pattern). As per Table 6.4, the orchestrator enters $\mathsf{h}_{\mathsf{lead}}$ (Fig. 6.16c) if:

1. the currently assisted human conforms to the HumanLeader pattern;
2. the service has not been completed;
3. the robot is sufficiently charged;
4. the human is moving (their current position $\mathrm{h}'_{\mathrm{curr,pos}}$ is different from the previous sensor reading $\mathrm{h}_{\mathrm{curr,pos}}$).

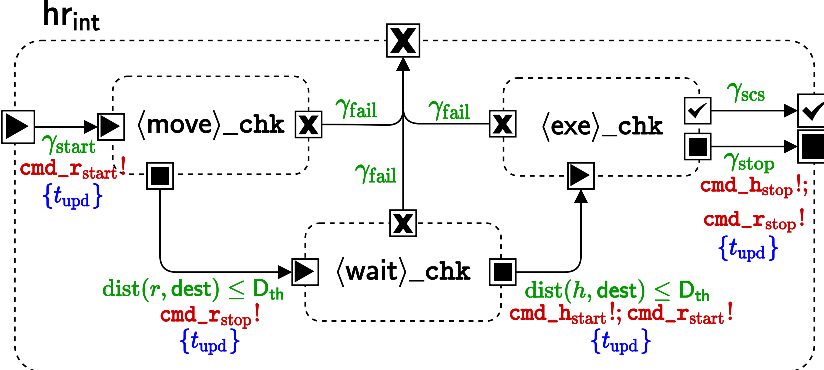


Figure 6.17: $\mathsf{hr}_{\mathsf{int}}$ *submachine, as seen in [130].*

Upon entering $\mathsf{h}_{\mathsf{lead}}$, $\mathtt{cmd_r}_{\mathrm{start}}$ is triggered, and the robot starts moving to follow the human (submachine $\mathsf{flw_chk}$). The orchestrator exits $\mathsf{h}_{\mathsf{lead}}$ when:

1. the human has reached an excessive fatigue level;
2. the robot's battery charge has dropped below the recharge threshold;
3. the human has set themselves as served, but there are other humans to serve in the scenario;
4. the human has stopped moving (their current position $\mathrm{h}'_{\mathrm{curr,pos}}$ is the same as the previous sensor reading).

As shown in Fig. 6.16c, when γ_{stop} holds, the orchestrator stops the robot through channel $\mathtt{cmd_r}_{\mathrm{stop}}$ and instructs the human to stop walking only if they are excessively fatigued. As explained in Section 6.3.2, the human may ignore the orchestrator's instruction due to free will.

6.4.5 $\mathsf{hr}_{\mathsf{int}}$ Submachine

The orchestrator handles the rescuer and applicant patterns through the $\mathsf{hr}_{\mathsf{int}}$ submachine, shown in Fig. 6.17. These two patterns require the robot and human to perform an action simultaneously and in close proximity; therefore, this submachine captures the following three phases modeled by as many $\langle\mathsf{op}\rangle\mathtt{_chk}$ instances:

1. the robot approaching the location where the action has to be performed (parameter dest);
2. the robot waiting until also the human reaches the destination;

3. the human-robot pair performing the required action.

The robot's approaching phase is captured by submachine $\langle\mathsf{move}\rangle$`_chk`. The service starts (see $\mathsf{hr_{int}}.\gamma_{\mathrm{start}}$ in Table 6.4) if the interaction pattern associated with the service is rescuer or applicant, the human is sufficiently rested and the active robot has sufficient charge. If these conditions hold, the orchestrator instructs the robot to start moving through channel $\mathtt{cmd_r}_{\mathrm{start}}$ upon entering such submachine and to stop moving through $\mathtt{cmd_r}_{\mathrm{stop}}$ when $\mathrm{dist}(r, \mathsf{dest} \leq \mathsf{D_{th}})$ holds.

Subsequently, the orchestrator stays in submachine $\langle\mathsf{wait}\rangle$`_chk` until the human reaches the destination as well ($\mathrm{dist}(h, \mathsf{dest} \leq \mathsf{D_{th}})$ holds).

Once both the human and the robot are in the right location, the action starts as instructed by the orchestrator through channels $\mathtt{cmd_h}_{\mathrm{start}}$ and $\mathtt{cmd_r}_{\mathrm{start}}$. The orchestrator then switches to submachine $\langle\mathsf{exe}\rangle$`_chk`.

If no condition for failure is met, the orchestrator exits $\mathsf{hr_{int}}$ if:

1. the action needs to be put on hold or is complete, but there are still services to provide (γ_{stop} holds); or
2. the action is complete and all services have been provided; thus, the mission has ended with success (γ_{scs} holds).

As per Fig. 6.17, all submachines have an outgoing edge to the *fail* port with guard condition γ_{fail} (see Formula (6.15)) since the mission may fail during all the three phases.

6.4.6 $\mathsf{r_{sync}}$ Submachine

The $\mathsf{r_{sync}}$ submachine, shown in Fig. 6.18, handles the interaction between two robots. The orchestrator enters this submachine when, while executing an action, it becomes necessary to **hand over** the task to another robot (e.g., if the active robot needs recharging). Upon interrupting the action, the active orchestrator synchronizes with the orchestrator controlling the closest free robot to hand over the current task. This robot-robot interaction pattern reduces the mission duration when a robot needs recharging, but other robots can replace it.

The entry condition γ_{start} for this submachine is given in Formula (6.17). Constant $\mathsf{N_r}$ represents the number of robots in the fleet. Index $i \in V_{\mathrm{dc}}$ indicates the active robot requesting the task handover, while index $j \in V_{\mathrm{dc}}$ indicates the robot selected to replace it. The value of index j is chosen based on: which robots are currently available (i.e., dense counter busy_j evaluates to false), which robots have a sufficient charge (i.e., $\mathrm{b_{j,chg}} \geq \mathsf{C_{rech}}$ holds), which robot is the closest to robot i.

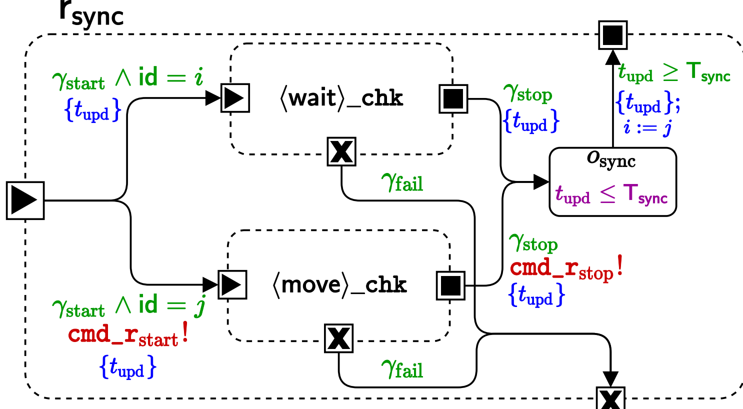


Figure 6.18: $\mathsf{r_{sync}}$ *submachine, as seen in [130].*

Condition γ_{handover} represents the condition triggering the task handover: a possible example is $\mathrm{b_{i,chg}} \leq \mathsf{C_{rech}}$, i.e., if robot i needs recharging. Nevertheless, the pattern is open to more complex conditions to be plugged into Formula (6.17) related to the system's efficiency.

$$
\begin{aligned}
&\gamma_{\text{handover}} \wedge \left(\bigvee_{j\in[1,\mathsf{N_r}]} j \neq i \wedge \neg \text{busy}_j \wedge \mathrm{b_{j,chg}} \geq \mathsf{C_{rech}} \right) \wedge \\
&\neg \left(\bigvee_{k\in[1,\mathsf{N_r}]} (k \neq j \neq i \wedge \text{dist}(\mathrm{r}_i, \mathrm{r}_k) < \text{dist}(\mathrm{r}_i, \mathrm{r}_j) \right)
\end{aligned}
\tag{6.17}
$$

For each orchestrator instance, $\mathsf{id} \in [1, \mathsf{N_r}]$ (see Fig. 6.18) is a constant value identifying such instance and the associated robot. The task handover pattern only involves orchestrators i (the one requesting the swap) and j (the one selected for the swap).

Before proceeding with the actual handover, robot j might have to move to get sufficiently close to robot i. More specifically, orchestrator i enters submachine ⟨wait⟩_chk (i.e., an instance of pattern ⟨op⟩_chk), where it waits for robot j. Orchestrator j instead enters submachine ⟨move⟩_chk (also an instance of ⟨op⟩_chk), which captures robot j moving to robot i's location to take over the task.

Since motion is required, upon entering ⟨move⟩_chk, orchestrator j instructs robot j to start moving by sending a message through channel $\mathtt{cmd_r}_{\text{start}}$. Orchestrator j enters ⟨move⟩_chk even if robot j is already sufficiently close to i, and, therefore, γ_{stop} will already hold after $\mathsf{T_{int}}$ time

instants (see Fig. 6.15). Despite the back-to-back start and stop commands, lower-level functions in the robot model prevent jolting and unnecessary movements if they are already close to the destination.

Guard condition γ_{stop} in Formula (6.18) determines when both orchestrators can exit their submachines.

$$\text{dist}(\text{r}_i, \text{r}_j) \leq \mathsf{D}_{\mathsf{restart}} \tag{6.18}$$

Once the distance between the two robots is smaller than threshold $\mathsf{D}_{\mathsf{restart}}$, both orchestrators switch to location o_{sync} (see Fig. 6.18), capturing the actual task handover phase. Upon leaving submachine move_chk, orchestrator j instructs its robot to stop moving (through channel $\texttt{cmd_r}_{\text{stop}}$) as it has reached its destination.

The ⟨op⟩_chk pattern ensures that both orchestrators enter submachine $\mathsf{r}_{\mathsf{sync}}$ and switch to location o_{sync} synchronously. Condition γ_{stop} in Formula (6.18) holds simultaneously for both orchestrators, and they are forced to leave their submachines (i.e., ⟨wait⟩_chk and ⟨move⟩_chk) as soon as γ_{stop} holds because location o_{chk} is committed. For the same reason, both orchestrators leave $\mathsf{r}_{\mathsf{idle}}$ (see Fig. 6.14) and enter $\mathsf{r}_{\mathsf{sync}}$ simultaneously as soon as the condition in Formula (6.17) holds.

At any time during the approaching phase, mission failure can occur: in this case, both orchestrators exit their current submachines through the *fail* port, switching to location o_{fail}.

The orchestrators stay in location o_{sync} for $\mathsf{T}_{\mathsf{sync}}$ time units, where $\mathsf{T}_{\mathsf{sync}}$ models the delay due to robot i handing over the task to robot j. The value of $\mathsf{T}_{\mathsf{sync}}$, hence the duration of the delay, depends on the nature of the task. For example, if no physical interaction is required (e.g., robot j has to follow or lead the human), the swap occurs instantly ($\mathsf{T}_{\mathsf{sync}} = 0$ holds). Otherwise, if the ongoing task requires handing over a physical object (e.g., if robot j has to deliver an object), the swap lasts $\mathsf{T}_{\mathsf{sync}} > 0$ time units. After time $\mathsf{T}_{\mathsf{sync}}$, the swap is finalized by having robot j becoming the new active robot (instruction $i := j$ in Fig. 6.18), which concludes the task handover.

Once the swap is complete, both orchestrators switch back to $\mathsf{r}_{\mathsf{idle}}$ (see Fig. 6.14). The previously active orchestrator detects that the associated robot meets the condition for recharging. Thus it switches to submachine $\mathsf{r}_{\mathsf{rech}}$ after time $\mathsf{T}_{\mathsf{int}}$ (see Fig. 6.15). The active orchestrator resumes the previously interrupted service by switching to the corresponding submachine (i.e., either $\mathsf{hr}_{\mathsf{int}}$, $\mathsf{r}_{\mathsf{lead}}$, or $\mathsf{h}_{\mathsf{lead}}$ depending on the interaction pattern).

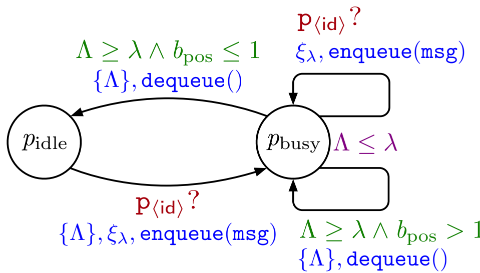


Figure 6.19: *SHA modeling the ROS publisher queue, also referred to as* ros_pub$_{\langle id \rangle}$*, as seen in [128].*

6.5 ROS Publisher Queue Model

For the design time and deployment results to be comparable, the SHA network includes a model of ROS publisher queues (i.e., instances of class ROSPubNode described in Section 5.1).

For each sensor, a ROS node periodically shares the latest readings with the orchestrator. Agents are responsible for *publishing* new data, whereas the orchestrator *subscribes* to such data. ROS handles all messages received by subscribers and shared by publishers through independent queues. The rate at which subscriber queues are emptied is easier to control and anticipate than publisher queues. As a matter of fact, the first one depends either on the time it takes to execute a single iteration of the callback function, or on the rate explicitly set for the execution of callbacks through the ros :: spinOnce() function. On the other hand, the time necessary to process a publisher queue depends on how quickly the message can reach the subscribers, which is not fully controllable [165].

As argued by Halder et al. [86], a formal model of this mechanism should include the following parameters: the publisher's publishing rate (T_{pub}), the subscriber's *spin* rate (T_{sub}), the time required to transmit messages over channels (T_{min} and T_{max}) and the time required to process callbacks (CB_{min} and CB_{max}). Sensor readings are shared with frequency $1/T_{poll}$ with $T_{poll} \in K$, therefore $T_{pub} = T_{poll}$ holds.

Subscriber nodes, within this framework, perform no other action besides data subscription, thus we assume $T_{sub} = 0$ holds, and the only delay attributable to the subscriber is the callback execution time. Given that callbacks consist of update instruction sets (see, for example, Eq.6.2), we assume that CB_{min} and CB_{max} are negligible compared to the timeline of a mission (usually in the order of *minutes*).

The ros_pub$_{\langle id \rangle}$ SHA, shown in Fig. 6.19, models independent pub-

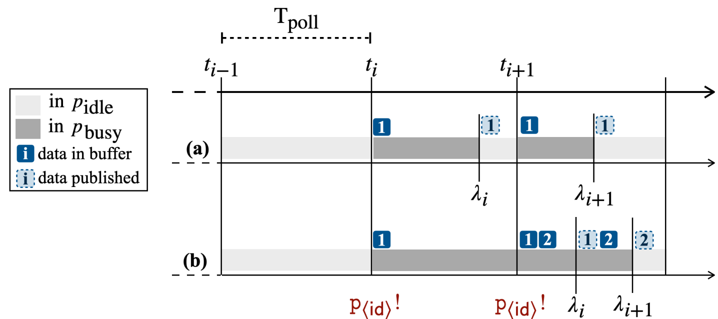


Figure 6.20: *Diagram of the synchronization contingencies between an* ⟨op⟩_pub$_{\langle id \rangle}$ *pattern and the corresponding* `ros_pub`$_{\langle id \rangle}$ *automaton, as seen in [128].*

lisher queues. This element of the SHA network captures the delays to transmit messages stored in queues over ROS topics.

The time required to publish a message is not fully predictable and constitutes a source of uncertainty. Therefore, a probability distribution approximates this delay. Instead of having a defined interval $\mathsf{T}_{\mathsf{min}}$ and $\mathsf{T}_{\mathsf{max}}$, we model the delay through variable $\lambda \in V_{\mathrm{dc}}$, whose values are randomly generated from a Normal distribution $\mathcal{N}(\lambda_{\mathrm{mean}}, \lambda_{\mathrm{var}})$ as shown in previous studies [208]. This feature incorporates the impact of delays due to ROS latency into the formal analysis.

As in Fig. 6.19, a publisher queue is *empty* (location p_{idle}) until an agent requests the publication of a message through channel $\mathsf{p}_{\langle \mathsf{id} \rangle} \in C$. Parameter ⟨id⟩ identifies the selected queue and semantically corresponds to a ROS topic. The publication request is captured by edge label $\mathsf{p}_{\langle \mathsf{id} \rangle}!$, either in the component in Fig. 6.1) or as seen in Fig. 6.2 (label $\mathsf{p}_1!$).

Queues are modeled as fixed-length arrays, and variable $b_{\mathrm{pos}} \in V_{\mathrm{dc}}$ keeps track of the first available position's index inside the queue. As the publication request is issued, the message is added to the buffer through update `enqueue(msg)`, a new value of λ is generated through update ξ_λ, and the automaton switches to location p_{busy}.

The automaton stays in p_{busy} as long as $\Lambda \leq \lambda$ holds, where clock $\Lambda \in X$ models the message publication latency. When time λ elapses, the first element of the queue is published to the orchestrator through instruction `dequeue()` (see Fig. 6.19). The latter subsumes the *subscriber* (e.g., the orchestrator) behavior, which, as already argued, is not explicitly modeled to limit complexity.

Fig. 6.20 displays possible contingencies (cases **(a)** and **(b)**) resulting from the combination of a ⟨op⟩_pub$_{\langle id \rangle}$ pattern with the corresponding

$\texttt{ros_pub}_{\langle\mathsf{id}\rangle}$ automaton. Case **(a)** occurs when $\forall i, \lambda_i \leq \mathsf{T}_{\mathsf{poll}}$ holds, therefore a message is always successfully published before the new reading and the queue never holds more than one element. If, on the other hand, $\exists i, \mathrm{s.t.} \lambda_i > \mathsf{T}_{\mathsf{poll}}$ holds, which corresponds to case **(b)** in Fig. 6.20, more than one position in the queue will be simultaneously occupied and two messages will be published back-to-back without switching back to p_{idle}. Case **(b)** is enabled by the two self-loops on p_{busy} (Fig. 6.19) and guard conditions on variable b_{pos}.

CHAPTER 7

Formal Modeling Approach Validation

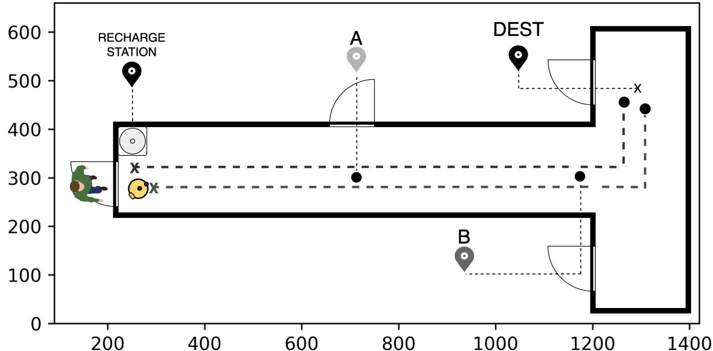
In this chapter, we present two case studies from the healthcare domain to test the formal modeling and analysis approach presented in Chapter 6.[a]

The first case study highlights how different fatigue profiles for the human subjects involved in the scenario impact the formal verification results (thus, on the outcome of the scenario).

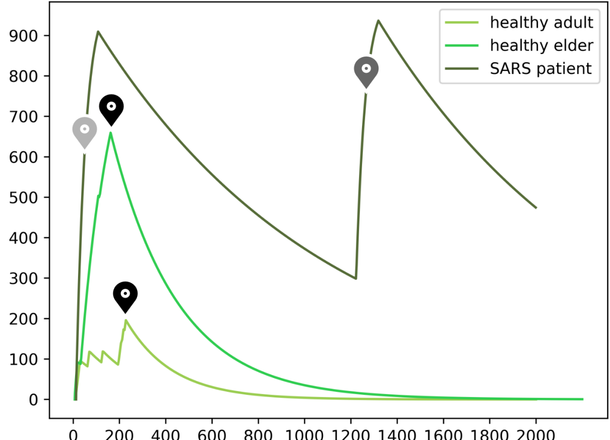
The second case study showcases how the framework can be exploited to analyze scenarios with multiple humans, multiple robots in the fleet, and alternative mission plans depending on the outcome of a specific service.

Finally, the results of a scalability analysis are reported to discuss how verification times increase approximately linearly with the size of the SHA network under analysis.

[a] The content presented in this chapter also appears in [125] and [130]. The author of this thesis declares to have also authored the reproduced text, figures, and data and to have the right to reproduce such content in a dissertation according to the license under which both articles are published.



(a) *Floor layout for case study 1, with initial positions and destination marked by a × symbol, and intermediate points by • symbols.*



(b) *Fatigue curves for all the profiles tested in the first case study (colors as in the legend). Location markers indicate the time each human has reached the corresponding point of the trajectory.*

Figure 7.1: *Experimental Setup and Results for Case Study 1, as seen in [125].*

7.1 Case Study 1: Fatigue Profile Impact Analysis

The first scenario involves a human patient who needs to reach a doctor's office. The mobile robot is aware of the floor plan and the patient's characteristics and can guide the human toward the destination. When the service is successfully provided, the robot successfully completes its mission.

This case study aims to assess the impact of different fatigue profiles on the mission's outcome. The experimental setting features a mobile robot with $\mathsf{v}_{\mathsf{max}} = 20\mathsf{cm/s}$, $\mathsf{a}_{\mathsf{max}} = 5\mathsf{cm/s}^2$, with a fully charged battery ($\mathsf{C}_0 = 100\%$) (corresponding to approximately 2.5h to full discharge).

The layout used for this experiment, depicted in Fig. 7.1a, reproduces a T-shaped hospital hallway, with doors leading to different offices. The entrance and starting point for both agents is on the left-end side (POI $(200, 300)$) close to the charging station (POI $(250, 375)$).

The robot's mission is to lead the human subject to their destination in $(1300, 500)$. Therefore, the interaction pattern is HumanFollower. To test the effectiveness of the fatigue-related policies of the orchestrator, we replicate formal analysis with different parameters for the human, i.e., different fatigue profiles and walking speed.

For the first configuration, the human is young and in fine health ($\mathsf{p_f} =$ Young/Healthy). Their Maximum Endurance Time (MET)—i.e., how long they can walk non-stop before $F = 1$—is on average 23 minutes (with reference to Eq.6.5, it leads to mean $\lambda = \rho = 0.005$) and their walking speed is v = 18cm/s. In the second configuration, the human is an elder in good health ($\mathsf{p_f} =$ Elderly/Healthy), with MET = 14min ($\lambda = 0.008$, $\rho = 0.0035$) and v $= 8cm/s$. In the third and final case, the human is affected by a severe respiratory disease ($\mathsf{p_f} =$ SARSPatient), v = 5cm/s) with MET = 4.6min ($\lambda = 0.025$, $\rho = 0.001$).

For these experiments, we use Uppaal version $4.1.24$ to implement the automata and run SMC experiments. Experiments are performed on a machine with 128 cores, 515GB of RAM, and Debian Linux version 10 with the default set of statistical parameters.

Each experiment yields the probability for mission success with time bound τ (the value of expression $\mathbb{P}_{\mathrm{M}}(\diamond_{\leq\tau}\mathsf{scs})$).

For the first configuration, we estimate that $\mathbb{P}_{\mathrm{M}}(\diamond_{\leq\tau}\ \mathsf{scs}) = [0.717, 0.817]$ with $\tau = 300s$. The second configuration leads to a success probability estimate of $\mathbb{P}_{\mathrm{M}}(\diamond_{\leq\tau}\ \mathsf{scs}) = [0.8, 0.9]$ with $\tau = 300s$. In the last case, we obtain that $\mathbb{P}_{\mathrm{M}}(\diamond_{\leq\tau}\ \mathsf{scs}) = [0, 0.098]$ with $\tau = 1000s$. These results show that in the first two cases, the destination can be reached in approximately 5min with a high degree of confidence. In the last case, instead, the mission cannot be completed even in 16min.

The motivation behind this result is highlighted by the simulations in Fig. 7.1: Fig. 7.1a shows the trajectories of the two agents, whereas Fig. 7.1b shows the fatigue curves for the three test cases and the time instants in which the destination or intermediate points of the trajectory have been reached within the simulation.

The orchestrator instructs every agent to stop walking if human fatigue exceeds a certain threshold, set to $\mathsf{F_{stop}} = 0.9$. In the last test case, motion stops at $t = 200s$, it resumes at $t = 1200s$ when fatigue drops to an acceptable value ($\mathsf{F_{restart}} = 0.3$) and stops again at $t = 1400s$ when the destination

Table 7.1: *Verification performance data for the first case study.*

Exp.	**States**	**Time** [min]	**Virtual Memory** [KiB]	**Resident Memory** [KiB]
$\mathsf{p_f}$ =Young/Healthy	212570	≈ 1	166488	120800
$\mathsf{p_f}$ =Elderly/Healthy	391311	≈ 1.5	166484	122376
$\mathsf{p_f}$ =SARS Patient	2927009	≈ 11.5	166484	123068

is yet to be reached. This behavior is caused by the orchestrator trying to prevent human exhaustion, which inevitably slows down the entire mission. In the other two cases, thanks to the different $\mathsf{p_f}$ parameter values, when the human reaches the destination the fatigue level is still acceptable, thus they are not stopped by the orchestrator.

Performance data for each experiment can be found in Table 7.1.

This case study constitutes a preliminary step towards assessing the soundness of the models presented in Chapter 6, specifically the ability of the orchestrator to enable corrective actions if required by the state of the system.

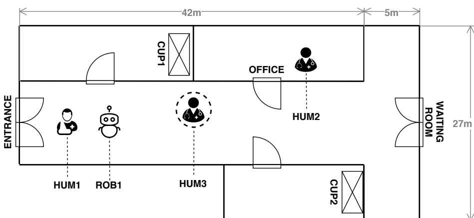
7.2 Case Study 2: Multi-Human Multi-Robot Scenario

The second case study showcases the formal modeling approach versatility through a broader range of human-robot interaction patterns. The scenario envisages the availability of a fleet of mobile robots to provide the services. The scalability of the approach with increasing fleet size is then assessed.

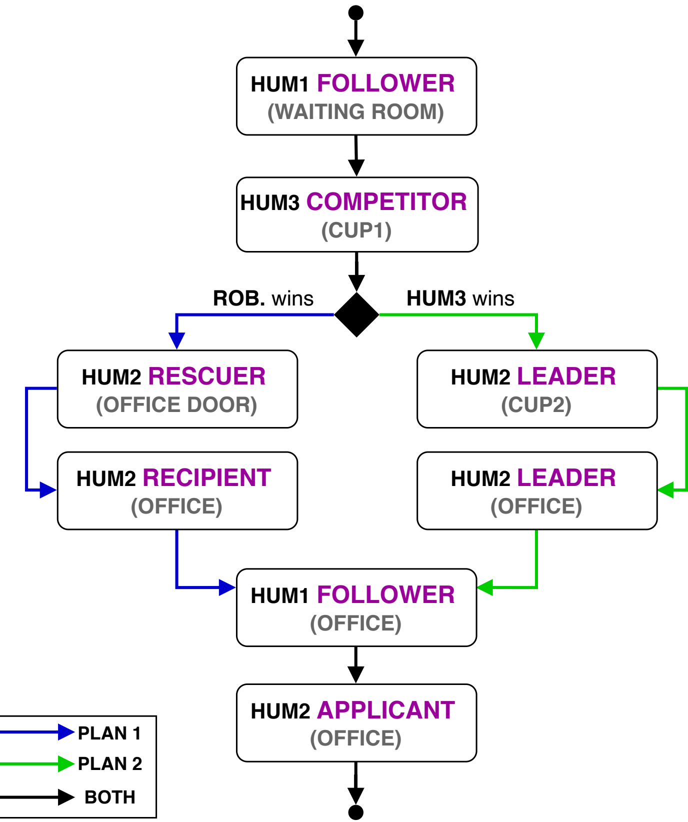
The second case study is also set in a T-shaped hospital corridor with cupboards containing medical equipment (**CUP1** and **CUP2**), doors leading to offices, and a waiting room. The scenario features a patient (**HUM1**) in need of medical treatment and a doctor (**HUM2**) to administer it.

When the scenario begins, the robot *accompanies* the patient to the waiting room. The robot should then fetch the equipment required for the treatment from **CUP1**: we assume that a second doctor (**HUM3**) simultaneously requires the same item, thus they *compete* to grab it. If the robot wins, it requires **HUM2**'s *assistance* to open the office door, and then it *delivers* them the object. If **HUM3** wins, the robot *follows* **HUM2** to the second cupboard (**CUP2**) and carries the item back to the office.

Since the outcome of the competitor pattern is not known beforehand, there are two possible paths if the robot or the third human wins the competition, represented in Fig. 7.2b as blue and green arrows, respectively. For testing purposes, **HUM3** starts from a random location in the corridor to



(a) *Floor layout for the case study with sizes ([m]) in grey. Humans* **1** *and* **2** *are represented in their starting locations, while **HUM3**'s starting location is random.*



(b) *Flowchart representing the requested service sequence. For each service, we highlight the human (in black), the interaction pattern (in purple), and the location (in grey). The diamond represents a decision node, and differently colored arrows represent alternative plans.*

Figure 7.2: *Floor layout and workflow for the second case study, as seen in [130].*

keep the outcome of the competition unpredictable (hence, their location in Fig. 7.2a is purely indicative). Once the doctor has the required equipment, the robot returns to the waiting room and *accompanies* the patient to the

office, where it *supports* the doctor in administering the treatment.

Doctors belong to different age groups (*young* and *elderly*) and are in a good state of health, while the patient exhibits a more critical fatigue profile. By default, the robot with $\mathsf{id} = 1$ (**ROB1**) starts providing the first service of the sequence; therefore, the outcome of the scenario varies based on the robot's initial state of charge (i.e., parameter C_0).

We run the SMC experiments through Uppaal v.4.1.24 on a machine with 4 cores and 8GB of RAM.

Table 7.2 reports the SMC results for the design-time analysis of the scenario, split by competition outcome (whether the robot or the doctor wins, i.e., parameter PLAN), level of charge (C_0), time-bound (τ), and the number of robots ($\mathsf{N_r}$). We perform experiments with $\tau = 400s$ or $\tau = 1200s$, and $\mathsf{C}_0 = 90\%$ or $\mathsf{C}_0 = 30\%$.

The robot's discharge curve is parameterized to match the battery life of commercially available mobile robotic devices (approximately $2.5\mathsf{h}$). When **ROB1** starts with $\mathsf{C_{start}} = 30\%$, it has approximately $5\mathsf{m}$ of battery life left due to the non-linearity of charge time-dynamics. The robots' maximum linear speed is also set to match devices available on the market ($1\mathsf{m/s}$), whereas humans' walking speeds range from $0.8\mathsf{m/s}$ (for the patient) to $1\mathsf{m/s}$ (for the doctors).

If only one robot is available ($\mathsf{N_r} = 1$ holds) and $\mathsf{C}_0 = 90\%$ holds, the estimated probability of success within $400\mathsf{s}$ is greater than 90% independently of the competition outcome.

When $\mathsf{C}_0 = 30\%$ holds, in all cases, the robot reaches the recharge threshold $\mathsf{C_{rech}} = 10\%$ when the mission is still executing. If there is no other robot to replace it ($\mathsf{N_r} = 1$ holds), the mission cannot resume until the robot recharges, thus the probability of success within $\tau = 400\mathsf{s}$ is null (≤ 0.098). With $\tau = 1200\mathsf{s}$, we obtain a higher success probability (approximately 50%) with both plans since the robot has time to recharge and complete the mission at least in some of the cases.

On the other hand, if $\mathsf{N_r} = 2$ holds, the first robot's orchestrator triggers the task handover routine to synchronize with the second robot. Since robot 2 resumes the service as soon as it reaches robot 1's location, it is not necessary to wait until robot 1 recharges to proceed and the probability of completing the mission within $400\mathsf{s}$ is greater than 90% with both plans.

The estimated maximum fatigue levels of **HUM1** (reported in Table 7.2) indicate that plan 2 puts more strain on the patient's side. Indeed, it takes less for the robot to move from **CUP1** to the office than for **HUM2** to wait for the robot, walk to **CUP2** and back to the office: thus, on average, plan 2 lasts longer than plan 1. Results in Table 7.2 confirm this since the

Table 7.2: *SMC results split by competition outcome (*PLAN*), ROB1's initial charge (*C_0*), time bound (*τ*), and fleet size (*N_r*). The table contains the success probability (*$\mathbb{P}_\mathrm{M}(\diamond_{\leq\tau}\ \mathsf{scs})$*) and* ***HUM1****'s maximum fatigue (*$\mathrm{E}_{\leq\tau}[\max(\mathrm{f}_1)]$*) estimations.*

Scenario Parameters				SMC Results	
PLAN	C_0	τ	N_{robs}	$\mathbb{P}_\mathrm{M}(\diamond_{\leq\tau}\ \mathsf{scs})$	$\mathrm{E}_{\leq\tau}[\max(\mathrm{f}_1)]$
1	90%	400s	1	$[0.902, 1]$	0.120 ± 0.058
	30%	400s	1	$[0, 0.0981]$	0.103 ± 0.053
	30%	1200s	1	$[0.491, 0.591]$	0.557 ± 0.043
	30%	400s	2	$[0.902, 1]$	0.148 ± 0.019
2	90%	400s	1	$[0.902, 1]$	0.156 ± 0.042
	30%	400s	1	$[0, 0.0981]$	0.102 ± 0.052
	30%	1200s	1	$[0.412, 0.512]$	0.667 ± 0.052
	30%	400s	2	$[0.902, 1]$	0.197 ± 0.092

estimated probability of success within 1200s when the robot has time to recharge is higher for plan 1 than plan 2 (the mean value of the probability interval is 54% compared to 46%). A longer mission duration implies a higher chance that humans make haphazard decisions (hence, their fatigue increases) out of free will, leading to higher fatigue levels for plan 2 than plan 1, all other parameters being equal.

Similarly, when $\mathsf{C}_0 = 30\%$, the estimated fatigue level for **HUM1** is higher with $\tau = 1200\mathsf{s}$ and $\mathsf{N}_r = 1$ than $\tau = 400\mathsf{s}$ and $\mathsf{N}_r = 2$ since, in the first case, the patient is free to move in the waiting room for a long time accumulating more fatigue. The same phenomenon, although on a smaller scale, explains the difference between the fatigue estimations obtained with $\mathsf{C}_0 = 90\%$ and those obtained with $\mathsf{C}_0 = 30\%$ and $\mathsf{N}_r = 2$. In this case, the mission lasts longer since the two robots require 1m to swap.

In conclusion, these results indicate that plan 1 is slightly preferable in terms of success probability and patients' physical effort. Therefore, the analysis may lead to a retuning of the facility's policies, if possible, to give the robot priority when accessing **CUP1** when a patient is waiting. Furthermore, the formal verification results corroborate the intuition that at least a second robot is necessary to treat the patient as quickly as possible when it is not feasible for a single robot to complete all tasks without recharging.

7.2.1 Scalability Analysis Results

Development approaches based on model checking often become impractical as the size of the problem grows due to excessive verification times.

Table 7.3: *Performance data of SMC experiments with* $C_0 = 30\%$*,* $\tau = 400$s*, and increasing robot fleet size (*N_r*). The table reports on explored states (*$[\times 10^6]$*), duration of the experiment (*[min]*), virtual and resident memory used (*[MB]*).*

N_{robs}	States	Time	Virtual Memory	Resident Memory
1	1.112M	2m 28s	4,424 MB	88.508 MB
2	1.117M	3m 46s	4,424 MB	92.560 MB
3	1.149M	3m 47s	4,432 MB	96.716 MB
5	1.404M	4m 11s	4,442 MB	106.416 MB
10	5.395M	12m 55s	4,460 MB	127.352 MB
20	6.725M	18m 28s	4,499 MB	171.080 MB

SMC partially eliminates this issue at the cost of approximate results and high dependency on the application designer's choices (see Section 3.1.2 and Section 4.2 for a more detailed discussion of these issues). Nevertheless, given that the technique relies on simulation, in some cases, converging to a confidence interval that is small enough may still require a large number of runs, thus significant computational load and verification times.

Given that the development approach is meant to be accessible and convenient, assessing whether it is still applicable to more significant problems is a relevant issue. Although the second case study features more and a broader range of interaction patterns, besides the management of alternative mission plans, performance data reported in Table 7.3 suggest an adequate degree of applicability of the approach as verification times with 1 or 2 robots do not exceed 4m.

Data in Table 7.3 refer to a single SMC experiment (i.e., for a single configuration). Thus, it takes approximately 30m to test all configurations described in Table 7.2.

We performed additional SMC experiments with the same scenario but increasing values of N_r. These experiments are all performed with $C_0 = 30\%$ to trigger the task handover, $\tau = 400$s, and N_r up to 20. For testing purposes, we simulate the possibility that some robots may be busy and randomize their starting physical location to make the replacement robot selection (i.e., robot j in Section 6.4.6) unpredictable.

Per Table 7.3, verification time and memory usage grow approximately linearly with the robot fleet size.

Although performance data show that the approach applies to increasingly complex scenarios, it is worth pointing out some limitations of this scalability assessment. Even if the model features larger robot fleets, they

are not all actively involved in the scenario. As described in Section 6.4.6, only the first robot and the one that takes over (even if its selection is not predictable) contribute to the mission.

More generally, the model does not support parallel execution of multiple tasks (e.g., multiple robots serving multiple humans *simultaneously*) due to limitations of the modeling tool. Therefore, design-time analysis results do not consider issues due to robots carrying out conflicting tasks, for example, causing their trajectories to overlap, as we assume these are handled by lower-level safety measures embedded into the robot devices. In the future, this limitation could be dampened by extending the deployment approach presented in Chapter 10 and incorporating parallel mission execution into the runtime analysis.

CHAPTER 8

Erroneous Human Behaviors Model

This chapter presents the formal model of human behavior extended with manifestations of erroneous behaviors. Firstly, we present the set of phenotypes of erroneous human behavior, how we have mapped the phenotypes to the service setting, and how we have modeled the mapped phenotypes as SHA add-ons. Secondly, we introduce the extensions of the SHA modeling human-robot interaction patterns presented in Chapter 6 with the erroneous behavior models.

8.1 Phenotypes of Erroneous Human Behavior

The issue of defining and categorizing manifestations of human *erroneous* behavior has been largely investigated in the field of human-computer interaction. Hollnagel's human error taxonomy [92] is among the best-established works in the field addressing how unexpected human behavior can cause the interaction with a machine to fail.

Specifically, [92] focuses on the cases in which, although the interaction plan is adequate, the performed actions stray from the plan: such actions are referred to as "*human errors*". Although the form and frequency of

Table 8.1: *Phenotypes of human erroneous behavior identified by Hollnagel's taxonomy [92]. For each phenotype, the table reports the corresponding error mode, an informative description, and an example of an action sequence displaying the erroneous behavior (highlighted in red).*

Error Mode (Action:)	**Phenotype**	**Description**	**Action Sequence**
in the wrong place	Repetition	An action is a repetition of the previous contiguous one.	A, B, C, C, D, E
	Reversal	The next two actions in the expected sequence are reversed.	A, B, D, C, E
in the wrong place/at the wrong time	Omission	An action has been omitted from the sequence.	$(t_1, A), (t_2, B), (t_3, \perp), (t_4, D), (t_5, E)$
at the wrong time	Delay	An action is performed, but not when required.	$(t_1, A), (t_2, B), (t_3, \perp), (t'_3, C), (t_4, D), (t_5, E)$
	Premature Action	An action is performed, but not when expected.	$(t_1, A), (t_2, B), (t'_3, C), (t_3, \perp), (t_4, D), (t_5, E)$
of the wrong type	Replacement	An action is performed as a substitute of another, but the two are functionally equivalent (i.e., C is equivalent to C′).	A, B, C', D, E
not included in current plan	Insertion	An action is performed that is not in the original plan, but does not disrupt it.	A, B, Y, C, D, E
	Intrusion	An action is performed that is not in the original plan and causes disruption (the goal cannot be achieved).	A, B, $Y \downarrow$

human errors cannot be fully predicted, they are bound to take place in a complex interactive system and tend to occur in patterns.

Error patterns consist of a *phenotype*, i.e., the manifestation of the erroneous action (e.g., a user failing to press a button in time), and a *genotype*, i.e., the cognitive process that causes the erroneous action (e.g., forgetting the intention of pressing the button). As our work does not capture the cognitive sources of human actions but their observations, throughout the thesis, we focus on phenotypes, while genotypes are investigated in [91].

The taxonomy proposed by Hollnagel features four macro-categories of human errors, referred to as "*error modes*" in Table 8.1: actions in the wrong place (i.e., the position of the action within the sequence is not correct), at the wrong time (i.e., the timing of the action is not as planned), of the wrong type (i.e., the action is not planned but does not disrupt the plan), not included in the current plan (i.e., the action is not in the planned sequence).

An error mode groups one or multiple phenotypes, each capturing a deviation (i.e., the "*error*") with respect to a *plan*, where a plan is intended as "*a representation of both a goal [...] and the possible actions required to achieve it*" [183].

For each phenotype, Table 8.1 reports action sequences displaying the corresponding erroneous behavior with respect to a planned sequence of generic actions indicated with symbols $[\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}]$. When timing is necessary to characterize the phenotype, timestamps of the form $\mathsf{t_i}$ indicating the expected time of occurrence of an action are also reported. The notation $(\mathsf{t_i}, \perp)$ indicates that *no* action takes place at time $\mathsf{t_i}$ contrary to what the plan envisages. In contrast, symbol $\downarrow$ indicates an unexpected termination of the sequence of actions.

The formal model in our framework captures human behavior while interacting with a robot, and the possible interactions are referred to as *patterns*. Since each pattern represents the service requested by the human to the robot, the *goal* of a pattern is to provide the service and conclude the interaction successfully. The condition determining whether a service is completed successfully is specific to the pattern and indicated in the following as $\gamma_{i,\text{scs}} \in \Gamma(W)$, where $i \in [1, \mathsf{N_h}]$ identifies a specific human (equivalently, a specific pattern, as each SHA modeling a human is an instance of a single pattern) and $\mathsf{N_h} \in K$ is the total number of humans involved in the scenario. The goal of a pattern i is then expressed through formula $\diamond_{\leq\tau}(\gamma_{i,\text{scs}})$, where τ is the time-bound of the SMC experiment, as explained in Section 3.1.2.

As described in Section 6.3, SHA modeling human behavior feature

multiple instances of the $\langle \mathsf{op} \rangle_\mathrm{pub}$ subautomaton, each capturing an operational state, such as walking or standing. Similarly, the SHA modeling the robot features multiple instances of the $\langle \mathsf{op} \rangle_\mathsf{pub}_{\langle \mathsf{id} \rangle}$ subautomaton presented in Section 6.2 modeling the operational state of the robot.

To achieve the goal of a service, the human and the robot perform a sequence of actions, i.e., the "*plan*" in our framework. The *occurrence* of an action in our modeling approach is captured through a *change* in the operational state (i.e., $\langle \mathsf{op} \rangle_\mathrm{pub}$ instance) of the SHA modeling the human or a change of location in the SHA modeling the robot. Given a SHA involved in a plan and indicating its location at a given time t as l, an atomic element $(\mathsf{t_i}, l')$ of the plan such that $\mathsf{t_i} > t$ holds indicates that the SHA should switch from l to l' at time $\mathsf{t_i}$. On the other hand, element $(\mathsf{t_i}, \perp)$ indicates that the change of operational state planned at time $\mathsf{t_i}$ does not take place due to a human error, thus, the SHA modeling the human remains in its current state.

For example, let us consider the HumanFollower pattern. The goal (expressed in informal terms) is that the human follows the robot to a certain location (not known a priori by the human). To this end, the plan in informal terms is the following: *robot starts moving*, *human starts walking*, *robot stops moving*, *human stops walking*. The corresponding plan reporting the operational state changes is shown in sequence (8.1) (the human's initial operational state is $\langle \mathsf{stand} \rangle_\mathrm{pub}$, while the robot starts in r_{idle}). Note that, to ease the distinction between changes related to the robot and to the human when presenting sequences, we report the full $\langle \mathsf{op} \rangle_\mathrm{pub}$ label for humans and only the label of the ordinary location within the $\langle \mathsf{op} \rangle_\mathsf{pub}_{\langle \mathsf{id} \rangle}$ subautomaton for the robot (i.e., r_{start} and r_{stop}).

$$[(\mathsf{t}_1, r_{\mathrm{start}}), (\mathsf{t}_2, \langle \mathsf{walk} \rangle_\mathrm{pub}_h), (\mathsf{t}_3, r_{\mathrm{stop}}), (\mathsf{t}_4, \langle \mathsf{stand} \rangle_\mathrm{pub}_h)] \tag{8.1}$$

In reality, human behavior in a service setting is barely constrained, and the decision-making process is highly susceptible to free will; therefore, numerous deviations from the plan are possible. In our framework, SHA modeling human behavior capture this aspect by embedding a formalization of human haphazard behavior.

In the following, we present the developed SHA add-ons modeling the phenotypes as summarized by Table 8.2, providing examples of the corresponding erroneous behaviors within the domain of our framework. For each add-on, we present its features and show the corresponding SHA portion modeling the "standard" (i.e., non-erroneous) behavior.

We remark that we indicate as "add-ons" portions of SHA (thus, sub-

Table 8.2: *Mapping between the developed add-ons and the phenotypes identified by Hollnagel [92].*

Add-On	Phenotype(s)
Disobey/Obey	Delay, →Intrusion
Free Will	Premature Action, Reversal, Insertion
Timer Expired	Omission
Safety Violation	Insertion, Intrusion, Premature Action
Critical Status	Intrusion

tuples of the one defined in Definition 1) that can be flexibly incorporated into other SHA. This feature increases the extensibility of the modeling approach for two reasons: if new add-ons are developed in the future, these can be easily plugged into existing human-robot interaction patterns; if new human-robot interaction patterns are developed in the future, these can be easily extended with the developed add-ons.

Note that, given the *stochastic* nature of the employed formalism, it is not possible for the developed add-ons to be purely non-deterministic like the original phenotypes. A probabilistic formalization is necessary and described in more detail when introducing the various add-ons in the next sections. Indeed, quantifying the probability of human errors' manifestations is a long-standing approach in probabilistic Human Reliability Analysis techniques such as CREAM [93], HEART [217], THERP [206], and THEA [174].

As per Table 8.2, developed add-ons cover 6 out of 8 phenotypes from Hollnagel's taxonomy. The reason why Repetition and Replacement are not supported is due to the range of human actions currently covered by the modeling approach. In the first case, supported actions cannot physically or logically be repeated: for instance, the human cannot "*start*" walking twice in a row since they either stop and restart (counting as two separate actions) or simply keep walking. Concerning Replacement, the approach does not include sets of *functionally equivalent* actions (e.g., a human subject picking up the wrong object). Therefore, performing a substitute for the correct action is not supported. Introducing functionally equivalent action sets will be investigated in future work with an extension of the available set of add-ons to model the Replacement error.

8.1.1 Disobey/Obey Add-On

In the following, we describe how the Disobey/Obey add-on described in Section 6.3.1 is mapped to Hollnagel's phenotypes. Specifically, referring to the HumanFollower running example, if the human erroneously behaves according to the Disobey/Obey add-on and no other error occurs throughout the pattern, the observed action sequence is shown in sequence (8.2), where $k \in \mathbb{N}$ is the number of times the self-loop on $\langle \mathsf{op} \rangle_\mathrm{pub}_h$ is taken in favor of the edge to $\langle \mathsf{op} \rangle'_\mathrm{pub}_h$.

$$\begin{aligned} &[(t_1, r_{\mathrm{start}}), (t_2, \bot), (t_2 + T_{\mathrm{int}}, \bot), \dots, \\ &(t_2 + kT_{\mathrm{int}}, \bot), (t_2 + (k+1)T_{\mathrm{int}}, \langle \mathsf{walk} \rangle_\mathrm{pub}_h), \\ &(t_3, r_{\mathrm{stop}}), (t_4, \langle \mathsf{stand} \rangle_\mathrm{pub}_h)] \end{aligned} \tag{8.2}$$

With $k = 1$, we obtain a Delay phenotype (see Table 8.1). All sequences observed with $k > 1$ are an iteration of said phenotype, resulting in a *longer* delay. The probability of choosing the disobey edge k times is $(1 - p)^k$. It is possible—in theory—that the obey edge is *never* chosen in favor of the disobey self-loop: the probability of this happening (i.e., $(1 - p)^k$ with $k \to \infty$) tends to 0 if $p < 1$ holds—i.e., if disobey is greater than 0. The action sequence observed in this corner case (marked by symbol $\to$ in Table 8.2) is given in sequence (8.3).

$$[(t_1, r_{\mathrm{start}}), (t_2, \bot), \dots] \tag{8.3}$$

Since the goal is never reached, this constitutes a special case of Intrusion with $Y = \bot$ (see Table 8.1).

8.1.2 Free Will Add-On

In the following, we describe which phenotypes are realized through the Free Will add-on introduced in Section 6.3.1. The HumanFollower pattern is eligible for the add-on in Fig. 6.7a since actions are initiated by the robot. While the intended plan is reported in sequence (8.1), if the Free Will edge fires (thus, the erroneous behavior occurs) at time $t'_2 < t_2$, the observed actions are those shown in sequence (8.4).

$$\begin{aligned} &[(t_1, r_{\mathrm{start}}), (t'_2, \langle \mathsf{walk} \rangle_\mathrm{pub}_h), \\ &(t_2, \bot), (t_3, r_{\mathrm{stop}}), (t_4, \langle \mathsf{stand} \rangle_\mathrm{pub}_h)] \end{aligned} \tag{8.4}$$

In this case, sequence (8.4) reports the switch to $\langle\mathsf{walk}\rangle_\mathrm{pub}_h$ (i.e., the human starting to walk) at time t_2' out of free will even if no event is fired through channel $\texttt{cmd_h}_{\mathrm{start}}$. Therefore, in our framework, the Free Will add-on realizes the Premature Action phenotype if it involves the correct action according to the sequence (like starting to walk, in this example), but is not performed at the expected time. For example, the human may start walking during the $\mathsf{T}_{\mathsf{int}}$ time range in which the orchestrator is processing data.

A possible corner case of this erroneous behavior is obtained by decreasing t_2' to the point that $\mathsf{t}_2' < \mathsf{t}_1$ holds. In this case, the observed actions are given in sequence (8.5), whose timestamps are not shown as they are not necessary to identify the error.

$$[\langle\mathsf{walk}\rangle_\mathrm{pub}_h, r_{\mathrm{start}}, r_{\mathrm{stop}}, \langle\mathsf{stand}\rangle_\mathrm{pub}_h] \tag{8.5}$$

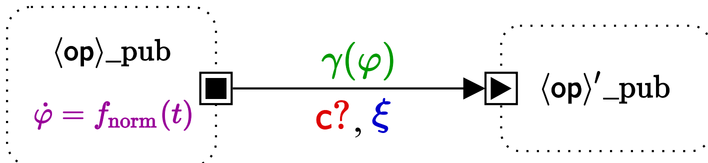
Therefore, in this case, the realized phenotype is a Reversal. We remark that sequence (8.5) is feasible in our framework since the orchestrator still issues instruction $\texttt{cmd_h}_{\mathrm{start}}$ *after* $\texttt{cmd_r}_{\mathrm{start}}$, but, since the human is already walking, no response to such instruction is observed on the human's side.

The third and final manifestation of the Free Will add-on is the human erroneously performing an action that is not envisaged by sequence (8.1), irrespectively of the time at which it is performed. For example, the human may abruptly stop walking while following the robot, resulting in sequence (8.6) (timestamps are not reported).

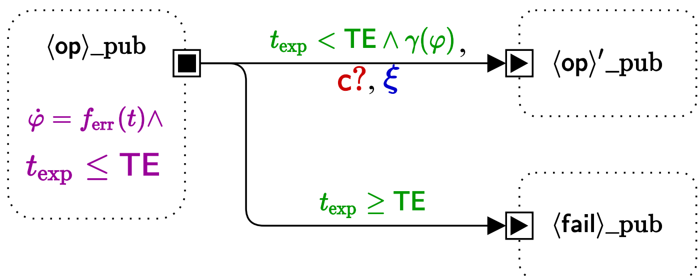
$$\begin{aligned}[&r_{\mathrm{start}}, \langle\mathsf{walk}\rangle_\mathrm{pub}_h, \langle\mathsf{stand}\rangle_\mathrm{pub}_h, \\ &\langle\mathsf{walk}\rangle_\mathrm{pub}_h, r_{\mathrm{stop}}, \langle\mathsf{stand}\rangle_\mathrm{pub}_h]\end{aligned} \tag{8.6}$$

This case realizes an Insertion phenotype since the human stopping is not expected but still allows the agents to reach the goal. In this situation, the orchestrator has to instruct the human to start walking again after the error occurs. Therefore, the additional $\texttt{cmd_h}_{\mathrm{start}}$ action is not expected, but it constitutes a *response* of the system to the error (thus, it is highlighted in blue and not in red).

The add-on in Fig. 6.7c, which captures actions initiated by the human, realizes the same phenotypes, with the difference that the point of reference is not the firing of channel c but condition γ being verified based on the system's state.



(a) *SHA representing the standard behavior.*



(b) *SHA representing the Timer Expired add-on, color-coded as in Fig. 3.1.*

Figure 8.1: *SHA representing a standard behavior and its version with the Timer Expired add-on representing the erroneous behavior, both color-coded as in Fig. 3.1.*

8.1.3 Timer Expired Add-On

The Timer Expired SHA add-on captures the human extremely delaying the completion of a task whose progress they are in charge of, to the point of being considered non-responsive. An example is the HumanLeader pattern, which is the dual case of the HumanFollower, in that the human is in charge of leading the robot to a certain destination. Similarly, the HumanApplicant pattern (described in detail in Section 8.2.1) features the human performing an action with the support of the robot, such as administering a treatment. In such cases, if the human performs impeccably, the action (i.e., walking or treating the patient) ends within a reasonable amount of time. However, unexpected time losses or the incumbency of an emergency may prevent the human from completing the task, leading the robot to consider them non-responsive and the service failed.

The standard behavior is shown in Fig. 8.1a. The SHA features two subautomata ⟨op⟩_pub$_h$ and ⟨op⟩′_pub$_h$ representing (as in Section 6.3.1 and Section 6.3.1) the current and the next operational state envisaged by the plan. Real-valued variable $\varphi \in W$ models the progress of the task the human performs while in ⟨op⟩_pub$_h$. The evolution of φ with time (e.g., the distance to the destination decreasing) is constrained by flow condition $f_{norm}(t)$. The switch from ⟨op⟩_pub$_h$ to ⟨op⟩′_pub$_h$ is realized through a

solid edge with guard $\gamma(\varphi)$, update ξ, and channel c: if the human initiates the switch, the channel is replaced by the internal action. Guard $\gamma(\varphi)$ is a condition on the value of φ evaluating to true when the task is complete.

The erroneous behavior modeled by the Timer Expired add-on, shown in Fig. 8.1b, captures the situation in which the progress of the task performed by the human is excessively delayed. The evolution of variable φ is, therefore, constrained by a different flow condition ($f_{\mathrm{err}}(t)$ in Fig. 8.1b). While function $f_{\mathrm{norm}}(t)$ models the human behaving normally, the function $f_{\mathrm{err}}(t)$ is such that condition $\gamma(\varphi)$ (capturing the completion of the task) may not be verified within a maximum time bound, corresponding to constant $\mathsf{TE} \in K$. A concrete example of how $f_{\mathrm{err}}(t)$ is implemented is given in Section 8.2.1 when describing in detail the HumanApplicant pattern. Subautomaton $\langle \mathsf{op} \rangle_\mathrm{pub}_h$ is further endowed with invariant $t_{\mathrm{exp}} \leq \mathsf{TE}$, where $t_{\mathrm{exp}} \in X$ is a clock that is reset upon entering $\langle \mathsf{op} \rangle_\mathrm{pub}_h$. If time bound TE is exceeded, the SHA switches to subautomaton $\langle \mathsf{fail} \rangle_\mathrm{pub}_h$. The edge from $\langle \mathsf{op} \rangle_\mathrm{pub}_h$ to $\langle \mathsf{fail} \rangle_\mathrm{pub}_h$ is labeled with guard condition $t_{\mathrm{exp}} \geq \mathsf{TE}$, which, in conjunction with the invariant on $\langle \mathsf{op} \rangle_\mathrm{pub}_h$, ensures that the transition occurs if and only if $t_{\mathrm{exp}} = \mathsf{TE}$ holds.

For each pattern eligible for this add-on, time-bound TE is estimated based on the characteristics of the human and the requested service. Eq.8.7 shows an example of how the value of TE is calculated in patterns requiring the human to move to a certain destination when initiating the movement is up to the human. In this case, variable φ corresponds to the distance between the human and the destination, and the completion of the task (i.e., condition $\gamma(\varphi)$) captures the distance being equal to 0. Function dist computes the distance between two points accounting for fixed obstacles (e.g., walls), $\mathrm{h}_{\mathrm{pos}} \in W$ is the Cartesian coordinate pair representing the human's position within the layout, $\mathsf{target} \in K$ is the Cartesian coordinates pair representing the destination of the service, $\mathsf{v} \in K$ is the human's walking speed, and $\delta \in K$ is the *allowance factor*.

$$\mathsf{TE} = \frac{\mathsf{dist}(\mathrm{h}_{\mathrm{pos}}(0), \mathsf{target})}{\mathsf{v}} \cdot (1 + \delta) \tag{8.7}$$

The ratio between the distance to be covered (thus, the distance between the human's starting position $\mathrm{h}_{\mathrm{pos}}(0)$ and the destination) and the walking speed represents the *expected* duration of the service in ordinary conditions, whereas δ determines how much the expected duration can be exceeded for the human to be considered non-responsive. Therefore, the higher the value of δ, the lower the likelihood of this erroneous behavior.

Let us refer to a [HumanLeader, HumanFollower] service sequence

to illustrate the manifestation of this erroneous behavior. The expected plan for the two services is shown in sequence (8.8), where the first 4 elements constitute the HumanLeader plan, while the last 4 constitute the HumanFollower plan.

$$[(\mathsf{t}_1, \langle\mathsf{walk}\rangle_\mathrm{pub}_h), (\mathsf{t}_2, r_{\mathrm{start}}), (\mathsf{t}_3, \langle\mathsf{stand}\rangle_\mathrm{pub}_h), (\mathsf{t}_4, r_{\mathrm{stop}}), (\mathsf{t}_5, r_{\mathrm{start}}), (\mathsf{t}_6, \langle\mathsf{walk}\rangle_\mathrm{pub}_h), (\mathsf{t}_7, r_{\mathrm{stop}}), (\mathsf{t}_8, \langle\mathsf{stand}\rangle_\mathrm{pub}_h)] \tag{8.8}$$

The erroneous behavior modeled by the Timer Expired add-on may occur while the human is *leading* the robot to the destination (captured by parameter target, as in Section 8.1.3) and they fail to reach it within time TE. In this case, the system behaves as in sequence (8.9).

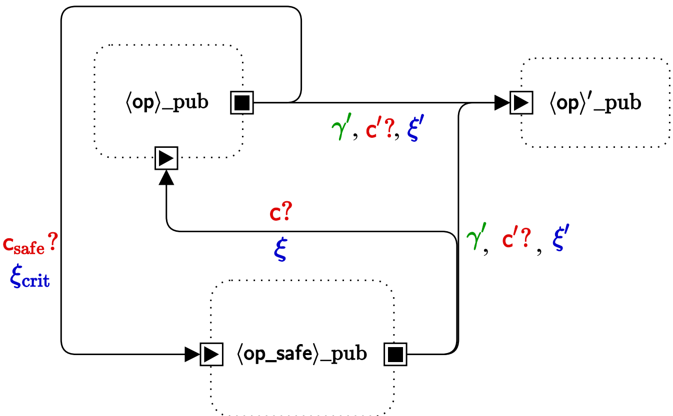
$$[(\mathsf{t}_1, \langle\mathsf{walk}\rangle_\mathrm{pub}_h), (\mathsf{t}_2, r_{\mathrm{start}}), (\mathsf{t}_3, \bot), \dots, (\mathsf{t}_3 + \mathsf{TE}, \bot), (\mathsf{t}'_4, r_{\mathrm{stop}}), (\mathsf{t}_5, r_{\mathrm{start}}), \dots] \tag{8.9}$$

We remark that, in sequence (8.9), t_3 is the expected duration of the task, and $\mathsf{t}'_4 \geq \mathsf{t}_3 + \mathsf{TE}$ holds since the robot stops serving the HumanLeader and starts serving the HumanFollower after the extra time allowed to complete the task has elapsed. Therefore, as per Table 8.1, this add-on realizes an Omission error phenotype. Note that, since the robot waits for the human to perform their task until time TE elapses, the switch to r_{stop} occurs at time $\mathsf{t}'_4 > \mathsf{t}_4$. However, this is not an error by itself but the system's response to the error made by the human (thus, it is highlighted in blue and not in red).

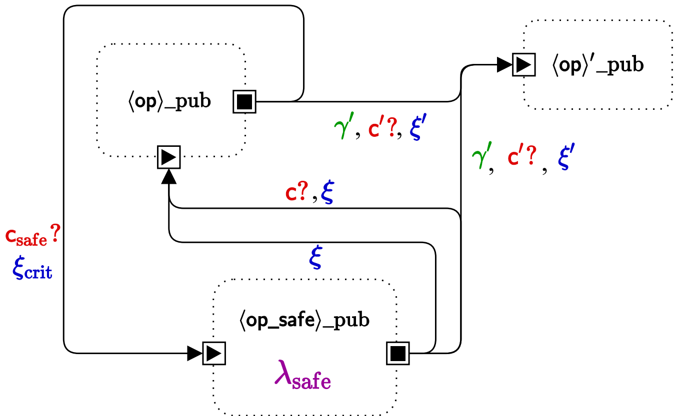
8.1.4 Safety Violation Add-On

The Safety Violation add-on captures the human entering a critical situation (e.g., moving too close to the robot), possibly causing a safety hazard [213]. Unlike operators in industrial settings, people in service settings do not wear protective devices nor receive systematic and thorough training in working alongside robots. Enforcing safety measures may be necessary throughout the interaction to prevent undesirable events, such as collisions. This add-on introduces a formalization of the human and the robot operating under a safety measure and the human violating such measure out of error.

The standard behavior, shown in Fig. 8.2a, models the situation in which the human is required to switch to a "*safe mode*" under specific circumstances while performing an action. As in previous cases, the current state



(a) *SHA add-on capturing the non-erroneous behavior.*



(b) *SHA add-on capturing the erroneous behavior: exponential rate λ_{safe} is color-coded like an invariant.*

Figure 8.2: *SHA depicting the standard behavior and the erroneous behavior captured by the Safety Violation add-on, color-coded as in Fig. 3.1.*

is modeled by subautomaton $\langle \mathsf{op} \rangle_\mathrm{pub}_h$, whereas the subsequent state in the plan is subautomaton $\langle \mathsf{op} \rangle'_\mathrm{pub}_h$. The orchestrator enforces the switch to $\langle \mathsf{op} \rangle'_\mathrm{pub}_h$ by firing an event through channel c' when condition γ' holds and, upon firing such event, update ξ' is executed. The condition determining whether the human should switch to the safety mode depends on the specific interaction pattern.

For example, referring to the HumanFollower pattern, the condition raising safety concerns is the human getting too close to the robot, expressed as $\mathsf{dist}(\mathrm{h_{pos}}, \mathrm{r_{pos}}) \leq \mathsf{d_{crit}}$, where function dist and variable $\mathrm{h_{pos}}$ are as described in Section 8.1.3, $\mathrm{r_{pos}} \in W$ is the Cartesian coordinate pair

representing the robot's position, and $\mathsf{d}_{\mathsf{crit}} \in K$ is a system-wide constant representing the maximum distance allowed before a safety measure is enforced. Therefore, the human should stay in $\langle \mathsf{op} \rangle_\mathrm{pub}_h$ only while the distance from the robot is greater than $\mathsf{d}_{\mathsf{crit}}$.

As soon as the safety-critical condition holds, the human receives an orchestrator instruction over channel $\mathsf{c}_{\mathsf{safe}}$ to switch to the safe mode, i.e., subautomaton $\langle \mathsf{op_safe} \rangle_\mathrm{pub}_h$, which captures the same operational state as $\langle \mathsf{op} \rangle_\mathrm{pub}_h$ but with the safety measure enforced. Realistic examples of this contingency would be the human being instructed to take a few steps to avoid the moving robot or walking at a slower pace to avoid colliding with the robot. Upon switching to $\langle \mathsf{op_safe} \rangle_\mathrm{pub}_h$, SHA variables are updated through ξ_{crit} to reflect the safety measure being enforced (e.g., setting a lower value of the human's walking speed). If the safety critical condition eventually holds, the human is instructed to switch back to $\langle \mathsf{op} \rangle_\mathrm{pub}_h$ through channel c. Note that the operational state modeled by $\langle \mathsf{op_safe} \rangle_\mathrm{pub}_h$ represents the *same* state as $\langle \mathsf{op} \rangle_\mathrm{pub}_h$ with different parameters (e.g., walking *at a lower pace*) and not a different functionally equivalent action (thus, it is not eligible for a Replacement phenotype).

The corresponding add-on modeling the erroneous behavior is shown in Fig. 8.2b. In this case, the modeled human error consists of arbitrarily *leaving* $\langle \mathsf{op_crit} \rangle_\mathrm{pub}_h$ even if the safety-critical condition is still in place (e.g., the human resuming walking at a full pace even if they are still too close to the robot). The human arbitrarily *leaving* $\langle \mathsf{op_safe} \rangle_\mathrm{pub}_h$ is captured by an additional edge back to $\langle \mathsf{op} \rangle_\mathrm{pub}_h$ without any channel, whose firing depends on exponential rate λ_{safe} added to subautomaton $\langle \mathsf{op_safe} \rangle_\mathrm{pub}_h$. The mechanism determining whether the SHA takes the new erroneous edge is stochastic rather than deterministic as in Fig. 8.2a. Specifically, the probability of leaving $\langle \mathsf{op_safe} \rangle_\mathrm{pub}_h$ t time units after entering it is $1 - \mathrm{e}^{-\lambda_{\mathrm{safe}} t}$. Therefore, the probability of switching back to $\langle \mathsf{op_safe} \rangle_\mathrm{pub}_h$ irrespective of the orchestrator's instructions increases with time. Note that, the longer the human stays in $\langle \mathsf{op_crit} \rangle_\mathrm{pub}_h$, which models the safe mode, the safer it is for the system. However, the higher λ_{safe} is, the more likely the human is to leave $\langle \mathsf{op_safe} \rangle_\mathrm{pub}_h$ shortly after entering it, when the safety-critical condition still holds.

The human erroneously not *entering* $\langle \mathsf{op_safe} \rangle_\mathrm{pub}_h$ upon receiving a message through $\mathsf{c}_{\mathsf{safe}}$ would be covered by applying the Disobey/Obey add-on to the edge from $\langle \mathsf{op} \rangle_\mathrm{pub}_h$ to $\langle \mathsf{op_crit} \rangle_\mathrm{pub}_h$.

This add-on gives rise to several phenotypes of erroneous behavior, thus different erroneous action sequences are illustrated in the following. Examples are provided taking as reference the HumanFollower pattern, specifi-

cally the sequence that envisages the orchestrator enforcing the safety measure, shown in sequence (8.10). Although the switch to $\langle\mathsf{walk_safe}\rangle_\mathrm{pub}_h$ is not envisaged by default by the HumanFollower pattern (see sequence (8.1)), it is not considered an error but a desired effect of the orchestrator's policies and the realization of the standard behavior in Fig. 8.2a. Therefore, the corresponding elements in sequence (8.10) are highlighted in blue and not in red.

$$\begin{aligned}&[r_{\mathrm{start}}, \left(\langle\mathsf{walk}\rangle_\mathrm{pub}_h, \langle\mathsf{walk_safe}\rangle_\mathrm{pub}_h\right)^{\mathsf{m}}, \\ &(\langle\mathsf{walk}\rangle_\mathrm{pub}_h)^{\mathsf{n}}, r_{\mathrm{stop}}, \langle\mathsf{stand}\rangle_\mathrm{pub}_h]\end{aligned} \tag{8.10}$$

Sequence (8.10) captures all the possible realizations of the standard behavior in Fig. 8.2a with $\mathsf{m} \geq 1$ and $\mathsf{n} \in \{0, 1\}$. In more detail, the human is instructed to enter the safe mode (i.e., walking at a slower pace) at least once. Afterwards, the human may be instructed to switch between $\langle\mathsf{walk}\rangle_\mathrm{pub}_h$ and $\langle\mathsf{walk_safe}\rangle_\mathrm{pub}_h$ $\mathsf{m} - 1$ times. Finally, in any case, the task can either terminate with the human in $\langle\mathsf{walk_safe}\rangle_\mathrm{pub}_h$ (if $\mathsf{n} = 0$ holds) or in $\langle\mathsf{walk}\rangle_\mathrm{pub}_h$ (if $\mathsf{n} = 1$ holds).

When applied to sequence (8.10), the Safety Violation add-on realizes different phenotypes based on the value of n. If $\mathsf{n} = 0$ holds (thus, the human should conclude the task in $\langle\mathsf{walk_safe}\rangle_\mathrm{pub}_h$) and the erroneous behavior in Fig. 8.2b occurs, sequence (8.11) is observed, which captures the human unexpectedly resuming walking at full speed.

$$\begin{aligned}&[r_{\mathrm{start}}, \langle\mathsf{walk}\rangle_\mathrm{pub}_h, \langle\mathsf{walk_safe}\rangle_\mathrm{pub}_h, \\ &\langle\mathsf{walk}\rangle_\mathrm{pub}_h, r_{\mathrm{stop}}, \langle\mathsf{stand}\rangle_\mathrm{pub}_h]\end{aligned} \tag{8.11}$$

Sequence (8.11) realizes two phenotypes from the same error mode (i.e., action "not included in current plan"): if the safety measure (even if active for a reduced amount of time) was successful in avoiding a hazard, the service can be completed successfully, resulting in an Insertion phenotype; otherwise (i.e., lifting the safety measure too early causes a hazard), the mission fails, effectively realizing an Intrusion phenotype. The same phenotypes are realized whether $\mathsf{m} = 1$ or $\mathsf{m} > 1$ hold.

If $\mathsf{n} = 1$ holds, it is sufficient to enforce the safety measure for a limited amount of time before all actions can resume in their normal mode (the human concludes the action in $\langle\mathsf{walk}\rangle_\mathrm{pub}_h$). In this case, the possible erroneous behavior is the human resuming the normal mode *too early*, as shown in sequence (8.12), which realizes a Premature Action phenotype with $\mathsf{m} = 1$.

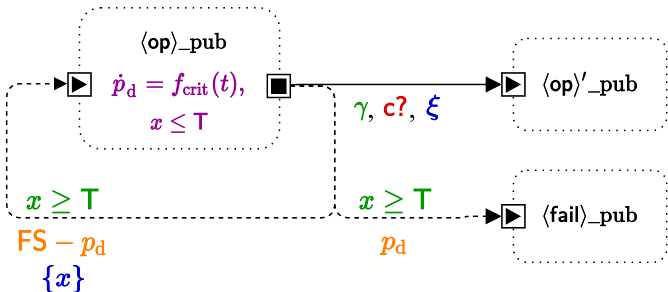


Figure 8.3: *SHA representing the* Critical Status *add-on, color-coded as in Fig. 3.1.*

$$[(\mathtt{t}_1, \mathtt{cmd_r}_{\mathrm{start}}), (\mathtt{t}_2, \mathtt{cmd_h}_{\mathrm{start}}), (\mathtt{t}_3, \mathtt{cmd_h}_{\mathrm{safe_start}}), (\mathtt{t}'_4, \mathtt{cmd_h}_{\mathrm{start}}), (\mathtt{t}_4, \perp), (\mathtt{t}_5, \mathtt{cmd_r}_{\mathrm{stop}}), \dots] \tag{8.12}$$

If m > 1 holds, the Premature Action phenotype can either refer to the last switch to $\langle \mathsf{walk} \rangle_\mathrm{pub}_h$ (like in sequence (8.12)) or any intermediate one.

8.1.5 Critical Status Add-On

Although our modeling framework is applicable to generic service settings, some of its features specifically target healthcare environments. In such cases, where people are often in pain or discomfort, robotic applications must safeguard humans' well-being and take into account *unexpected* (rather than purely erroneous) health-related accidents. This contingency is captured by the Critical Status add-on, capturing human subjects facing a sudden unexpected health issue (e.g., fainting) that requires immediate medical attention.

The standard behavior, in this case, may be captured by both Fig. 6.6a and Fig. 6.7b: in the following, we present the add-on as a variation of Fig. 6.6a. However, the same conclusions can be drawn on the SHA in Fig. 6.7b by replacing c with the internal action. The standard behavior envisages the human whose current state is modeled by subautomaton $\langle \mathsf{op} \rangle_\mathrm{pub}_h$ and upcoming state by $\langle \mathsf{op} \rangle'_\mathrm{pub}_h$. The switch from $\langle \mathsf{op} \rangle_\mathrm{pub}_h$ to $\langle \mathsf{op} \rangle'_\mathrm{pub}_h$ either depends on the orchestrator's instructions sent through channel c or the human's own initiative. The edge is enabled when guard γ holds and, upon firing, causes update ξ to execute.

As per Section 6.3, our modeling approach features a model of physical fatigue that increases when the human is actively performing an action

and decreases when they are resting. Human fatigue can only increase up to a maximum threshold (1 in our case, representing that 100% of muscle reservoir units have been activated) before the human can no longer move autonomously. The Critical Status add-on captures the possibility that the human faints or a similarly impairing accident occurs even if their current fatigue level is still below the maximum threshold. The SHA add-on representing this contingency is shown in Fig. 8.3: the additional location $\langle \mathsf{fail} \rangle_\mathrm{pub}_h$ represents the deadlock reached by the SHA if the accident occurs, causing the failure of the mission.

As shown in Fig. 8.3, location $\langle \mathsf{op} \rangle_\mathrm{pub}_h$ features invariant $x \leq \mathsf{T}$ as in Fig. 6.7, where $x \in X$ is a clock and $\mathsf{T} \in K$ is a constant. Compared to the standard behavior, the add-on has two additional edges leaving $\langle \mathsf{op} \rangle_\mathrm{pub}_h$, a self-loop and the edge to $\langle \mathsf{fail} \rangle_\mathrm{pub}_h$, both labeled with guard $x \geq \mathsf{T}$. Every T time units, there is a certain probability that the accident occurs, and the mission fails (the SHA switches to $\langle \mathsf{fail} \rangle_\mathrm{pub}_h$), or that the human remains in the same state (the self-loop on $\langle \mathsf{op} \rangle_\mathrm{pub}_h$ fires). Unlike in the Disobey/Obey add-on, probability weights are not constant, but their value changes with time (thus, they are real-valued variables). We indicate as $p_\mathrm{d} \in W$ the real-valued variable in question, whose derivative is constrained through a flow condition on $\langle \mathsf{op} \rangle_\mathrm{pub}_h$. In our specific case, the add-on envisages that the higher the level of fatigue, the higher the probability of an accident occurring. Therefore, the flow condition constraining p_d is indicated as $f_\mathrm{crit}(t)$ in Fig. 8.3 for the sake of generality, but in our specific case, it is a customizable function of fatigue, modeled by real-valued variable $F \in W$. An example, featured by the SHA presented later in this section, is $f_\mathrm{crit}(t) = \mathsf{hs} \cdot \dot{F}(t)$, where $\mathsf{hs} \in K$ is a customizable parameter determining how rapidly the probability of an accident increases with fatigue. In general, the probability weight for the self-loop on $\langle \mathsf{op} \rangle_\mathrm{pub}_h$ should evolve in time inversely with respect to fatigue (the higher the fatigue level, the lower the probability that the human does *not* have an accident and stay in the same state). A trivial example of expression determining the probability weight on the self-loop, also depicted in Fig. 8.3, is $\mathsf{FS} - p_\mathrm{d}$, where $\mathsf{FS} \in K$ is a constant such that $\mathsf{FS} \geq \sup(p_\mathrm{d})$ holds.

The action sequence observed if the behavior captured by this add-on occurs features, irrespective of the specific pattern, an unexpected action corresponding to the accident, this is clearly not part of the original plan and prevents the human-robot pair from achieving the goal, effectively realizing an Intrusion phenotype.

8.2 Human-Robot Interaction Patterns

Each SHA modeling an interaction pattern captures how the human behaves—either autonomously or in response to a robot's action—to achieve the goal of the specific service. In the following, we describe how the six SHA modeling human agents are extended through the presented erroneous behavior add-ons.

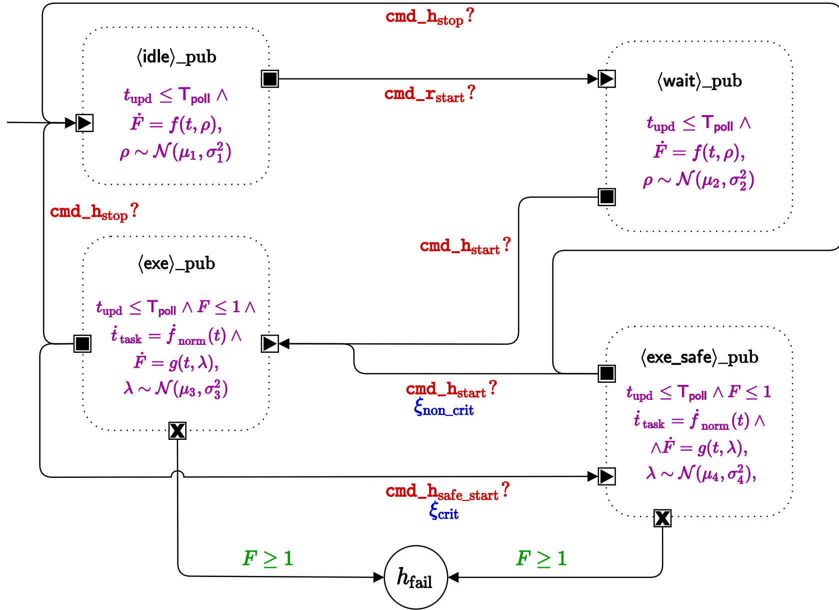
Not all add-ons apply to all patterns since we rule out unfeasible or unrealistic behaviors. In the following, we present the extended SHA modeling the HumanApplicant pattern in detail as an example of how add-ons are applied to HRI patterns. We then outline how a similar procedure has enriched the other five patterns.

8.2.1 HumanApplicant Pattern

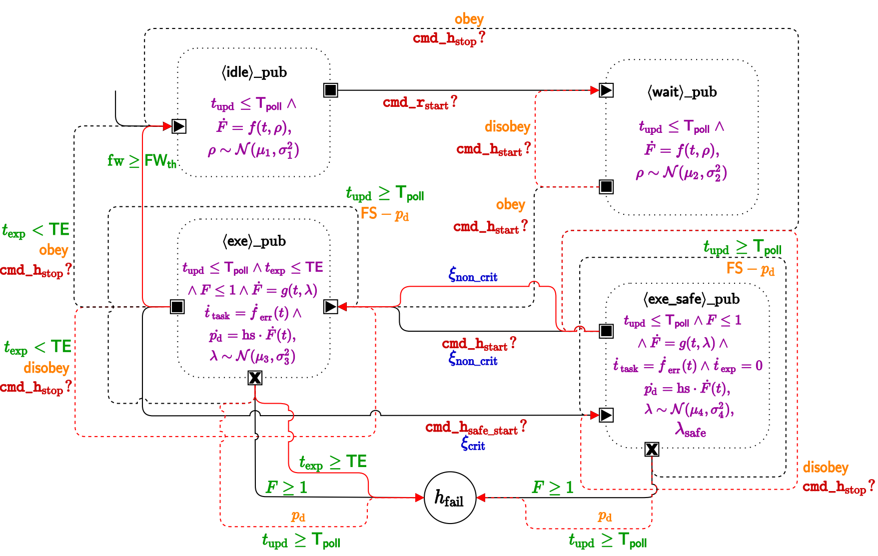
The HumanApplicant interaction pattern captures contingencies in which the human requests the robot's support to complete a task that requires working in a very close distance or sharp timely synchronization [130]. Example applications are robotic companions supporting a patient while feeding or healthcare professionals receiving the support of a service robot while administering medication. In the following, firstly, we recap the standard behavior of this pattern, shown in Fig. 8.4a; secondly, we describe the SHA (shown in Fig. 8.4b) extended with add-ons to incorporate erroneous behaviors.

The SHA modeling the HumanApplicant pattern features four instances of $\langle \mathsf{op} \rangle_\mathrm{pub}_h$ corresponding to the three phases of the service plus the operational state under critical conditions ($\langle \mathsf{op_safe} \rangle_\mathrm{pub}_h$ in Fig. 8.2b). When the service starts, the human is idle and resting, captured by subautomaton $\langle \mathsf{idle} \rangle_\mathrm{pub}_h$. The robot starts moving to approach the human when the orchestrator fires an event through channel $\mathtt{cmd_r}_{\mathrm{start}}$, causing the human to switch to $\langle \mathsf{wait} \rangle_\mathrm{pub}_h$, also a recovery state. The flow condition constraining F is, therefore, $f(t, \rho)$ (see Eq.6.5) both in $\langle \mathsf{idle} \rangle_\mathrm{pub}_h$ and $\langle \mathsf{wait} \rangle_\mathrm{pub}_h$. Normal distributions $\mathcal{N}(\mu_1, \sigma_1^2)$ and $\mathcal{N}(\mu_2, \sigma_2^2)$ determine the values of rate ρ in $\langle \mathsf{idle} \rangle_\mathrm{pub}_h$ and $\langle \mathsf{wait} \rangle_\mathrm{pub}_h$, respectively.

When the robot has reached the human's position, the orchestrator instructs the human to start performing the required task by firing an event through $\mathtt{cmd_h}_{\mathrm{start}}$, causing the human to switch to subautomaton $\langle \mathsf{exe} \rangle_\mathrm{pub}_h$. The standard duration of the task is modeled by a parameter $\mathsf{T}_{\mathsf{task}} \in K$, while $\mathsf{dext} \in K$ represents the human's *dexterity*, i.e., the rate at which they perform the specific task. The real-valued variable capturing the progress of the task is $t_{\mathrm{task}} \in W$ (φ in Fig. 8.1a), which evolves in time according



(a) *SHA representing the standard behavior of the* HumanApplicant *pattern.*



(b) *SHA representing the* HumanApplicant *enriched with erroneous behavior add-ons. Color-coding is as in Fig. 3.1 except for edges capturing human errors, highlighted in red for visualization purposes.*

Figure 8.4: *SHA modeling the* HumanApplicant *pattern.*

to flow condition $f_{\mathrm{norm}}(t) = \mathsf{dext} \cdot t$. In the ordinary case, the orchestrator instructs the human to stop by means of $\mathtt{cmd_h}_{\mathrm{stop}}$ and switch back to $\langle\mathsf{idle}\rangle_\mathrm{pub}_h$ when the human has spent $\frac{\mathsf{T_{task}}}{\mathsf{dext}}$ time units working on the task [130].

If, while performing the task, the orchestrator determines that the human and the robot are in a critical situation (i.e., their distance is below a certain threshold or human fatigue is above a critical level), it instructs the human to proceed cautiously. The human receives this instruction through channel $\mathtt{cmd_h}_{\mathrm{safe_start}}$ and switches to subautomaton $\langle\mathsf{exe_safe}\rangle_\mathrm{pub}_h$. Both $\langle\mathsf{exe}\rangle_\mathrm{pub}_h$ and $\langle\mathsf{exe_safe}\rangle_\mathrm{pub}_h$ subautomata are fatigue states, thus endowed with flow condition $g(t, \lambda)$ (see Eq.6.5). Distributions $\mathcal{N}(\mu_3, \sigma_3^2)$ and $\mathcal{N}(\mu_4, \sigma_4^2)$ determine the values of fatigue rate λ. Since $\langle\mathsf{exe_safe}\rangle_\mathrm{pub}_h$ captures the human working at a slower pace to avoid exhaustion or bumping against the robot (enforced through update ξ_{crit}, which reduces the value of parameter dext), $\mu_4 < \mu_3$ holds. If the safety measure is successful, the orchestrator instructs the human to switch back to $\langle\mathsf{exe}\rangle_\mathrm{pub}_h$ through channel $\mathtt{cmd_h}_{\mathrm{start}}$, and update $\xi_{\mathrm{non_crit}}$ restores the normal value of dext. Otherwise, if the task is completed while the human is in $\langle\mathsf{exe_safe}\rangle_\mathrm{pub}_h$, the orchestrator instructs it to switch back to $\langle\mathsf{idle}\rangle_\mathrm{pub}_h$ through channel $\mathtt{cmd_h}_{\mathrm{stop}}$.

Finally, deadlock location h_{fail} is reachable by the two fatigue states upon reaching the maximum endurable level of fatigue (guard $F \geq 1$ holds).

The edges modeling erroneous behaviors are highlighted in red in Fig. 8.4b, and the applied add-ons are individually described in the following.

Disobey/Obey Add-On Initially, the human may *delay* the start of the action and not respond to the $\mathtt{cmd_h}_{\mathrm{start}}$ command. Therefore, the edge from $\langle\mathsf{wait}\rangle_\mathrm{pub}_h$ to $\langle\mathsf{exe}\rangle_\mathrm{pub}_h$ is expanded into a Disobey/Obey add-on.

Similarly, both edges from $\langle\mathsf{exe}\rangle_\mathrm{pub}_h$ and $\langle\mathsf{exe_safe}\rangle_\mathrm{pub}_h$ to $\langle\mathsf{idle}\rangle_\mathrm{pub}_h$, marking the end of the action through channel $\mathtt{cmd_h}_{\mathrm{stop}}$, might be erroneously ignored by the human, and are thus expanded into a Disobey/Obey add-on.

Free Wil Add-On While in $\langle\mathsf{exe}\rangle_\mathrm{pub}_h$, the human may erroneously pause the task before it is complete (thus, before $\mathtt{cmd_h}_{\mathrm{stop}}$ fires).

Subautomata $\langle\mathsf{exe}\rangle_\mathrm{pub}_h$ and $\langle\mathsf{idle}\rangle_\mathrm{pub}_h$ are connected by an additional edge implementing the Free Will add-on, which fires when $\mathrm{fw} \geq \mathsf{FW_{th}}$ holds. Dense counter fw is updated every $\mathsf{T_{poll}}$ time instants by $\xi_{\langle\mathsf{exe}\rangle}$ (an instance of $\xi_{\langle\mathsf{op}\rangle}$ in Fig. 6.4, embedded into the $\langle\mathsf{exe}\rangle_\mathrm{pub}_h$ subautomaton) as described in Section 6.3.1.

Timer Expired Add-On While performing the task, the human might erroneously waste time and delay the completion of the action to the point that the robot considers them no longer responsive, as envisaged by the Timer Expired add-on. Therefore, variable t_{task}, capturing the progress of the task, is constrained by a different flow condition, i.e., $f_{\text{err}}(t)$ shown in Eq.8.13.

$$f_{\text{err}}(t) = (\mathsf{dext} \cdot (\text{rand}(0, \mathsf{T}_{\mathsf{max}}) \geq \mathsf{T}_{\mathsf{th}})) \cdot t \tag{8.13}$$

The progress of the task is governed by a stochastic mechanism as, for each time instant, the function rand draws a new sample from Uniform distribution $\mathcal{U}_{[0,\mathsf{T}_{\mathsf{max}})}$ and increases the value of t_{task} if the sample is greater than a customizable threshold T_{th} [130]. Therefore, the randomized evolution of variable t_{task} may lead to an unacceptable delay in the completion of the task.

Since, in this case, the human is not walking towards a target but performing a task for a certain amount of time, time-bound TE to deem the human non-responsive depends on the expected duration of the task (parameter $\mathsf{T}_{\mathsf{task}}$) and the rate at which the human performs it (dext) as per Eq.8.14, where δ is the allowance factor as illustrated in Section 8.1.3.

$$\mathsf{TE} = \frac{\mathsf{T}_{\mathsf{task}}}{\mathsf{dext}} \cdot (1 + \delta) \tag{8.14}$$

Subautomaton $\langle\mathsf{exe}\rangle_\text{pub}_h$ is endowed with invariant $t_{\text{exp}} \leq \mathsf{TE}$, where $t_{\text{exp}} \in X$ is a clock (as in Fig. 8.1b). As in Fig. 8.1b, the edge connecting subautomaton $\langle\mathsf{exe}\rangle_\text{pub}_h$ to deadlock location h_{fail} with guard condition $t_{\text{exp}} \geq \mathsf{TE}$ fires as soon as time TE elapses.

On the other hand, although the evolution of t_{task} is constrained by Eq.8.13 also while in $\langle\mathsf{exe_safe}\rangle_\text{pub}_h$, this operational state is not extended through the Timer Expired add-on since working at a slower pace is implied by the safety measure. Therefore, the time the human spends in $\langle\mathsf{exe_safe}\rangle_\text{pub}_h$ does not count towards upper bound TE ($\langle\mathsf{exe_safe}\rangle_\text{pub}_h$ is endowed with flow condition $\dot{t}_{\text{exp}} = 0$).

Safety Violation Add-On If the orchestrator finds that the human and the robot are in a safety-critical situation, it will instruct the human to switch to $\langle\mathsf{exe_safe}\rangle_\text{pub}_h$, corresponding to $\langle\mathsf{op_safe}\rangle_\text{pub}_h$ in Fig. 8.2a.

As envisaged by the Safety Violation add-on, the human may erroneously resume working at a normal pace, potentially causing a safety hazard. Therefore, while the orchestrator instructs the human to resume normal operations through channel $\texttt{cmd_h}_{\text{start}}$, an additional edge without

labels except update $\xi_{\text{non_crit}}$ restoring the standard value of dext connects $\langle\text{exe_safe}\rangle_\text{pub}_h$ to $\langle\text{exe}\rangle_\text{pub}_h$.

Subautomaton $\langle\text{exe_safe}\rangle_\text{pub}_h$ is endowed with parameter λ_{safe} representing the rate at which the probability of erroneously switching back to $\langle\text{exe}\rangle_\text{pub}_h$ increases with time.

Critical Status Add-On Since this pattern applies to patients and healthcare professionals who may find themselves in stressful situations or undiagnosed conditions, the modeled human subject is susceptible to accidents.

Therefore, both $\langle\text{exe}\rangle_\text{pub}_h$ and $\langle\text{exe_safe}\rangle_\text{pub}_h$ are extended through the Critical Status add-on. Specifically, at time t, where t is a multiple of $\mathsf{T}_{\mathsf{poll}}$ (T in Fig. 8.3), the probability of an accident occurring is p_{d}, where $p_{\text{d}} \in W$ is a real-valued variable. If this occurs, the SHA switches to h_{fail}. Otherwise, the probability of the SHA remaining in the same operational state depends on weight $\mathsf{FS} - p_{\text{d}}$, where $\mathsf{FS} \in K$ is a constant such that $\mathsf{FS} \geq \sup(p_{\text{d}})$ holds.

The probability of an accident occurring increases with the level of fatigue, as implied by flow condition $\dot{p_{\text{d}}} = \mathsf{hs} \cdot F(t)$, where $\mathsf{hs} \in K$ is a numerical constant.

8.2.2 HumanFollower Pattern

The HumanFollower pattern envisages the human following the robot to a certain destination. Operational states (corresponding to as many $\langle\text{op}\rangle_\text{pub}_h$ instances) capture the human *standing* (thus, recovering) and *walking*.

Upon receiving the instruction from the robot to start or stop walking (thus, either in the *standing* and *walking* states), the Disobey/Obey add-on introduces the possibility that the human ignores it. Similarly, the human may decide to start or stop walking irrespective of the robot's instructions through the Free Will add-on.

Since the Follower pattern does not envisage any motion initiated by the human, the Timer Expired add-on does not apply.

On the other hand, it is feasible for the human to walk too close to the robot, leading to the enforcement of a safety measure (i.e., the human walking slower). Moving in critical conditions is captured by a third $\langle\text{op}\rangle_\text{pub}_h$ instance, which is subject to the Safety Violation add-on.

Finally, to capture the possibility of unexpected accidents, all three operational states are extended through the Critical Status add-on: while walking (either normally or at a slower pace), probability p_{d} increases, while it decreases when the human is resting.

8.2.3 HumanLeader Pattern

The HumanLeader pattern captures the mirrored situation compared to the Follower, featuring the human leading the robot to a certain destination.

This SHA features the same $\langle \mathsf{op} \rangle_\mathrm{pub}_h$ instances as the Follower (*walking* and *standing*), although the actions of starting to walk and stopping are initiated by the human rather than the robot. Therefore, all edges between the two subautomata are extended through the Free Will add-on.

The robot only sends an instruction when the human's fatigue level rises to a critical threshold, advising them to stop and recover. This edge is extended through the Disobey/Obey add-on, as the human may erroneously ignore or miss the robot's suggestion.

Since this pattern captures a human freely operating on the floor, it is possible for them to get caught up in alternative tasks causing them to excessively delay the *walking* phase, which is captured by the Timer Expired add-on.

As with the Follower pattern, the Safety Violation add-on captures the situation in which the human erroneously starts walking at full pace while in a critical situation.

Finally, the Critical Status add-on captures the possibility of unexpected accidents.

8.2.4 HumanRecipient Pattern

The HumanRecipient pattern captures fetch-and-delivery tasks where the robot retrieves a required object and delivers it back to the human.

The standard behavior, in this case, features two operational states, i.e., the human *waiting* for the robot to retrieve the object and *interacting* with the robot to collect the item. The latter action is performed upon the robot's instruction and is thus extended through the Disobey/Obey add-on.

To capture the possibility that the human might move while waiting for the robot, the SHA features an additional operational state capturing the human walking. The switch from the idle operational state, which is entirely up to the human, occurs through the Free Will add-on.

Since the pick-up action is supposedly almost instantaneous, the HumanRecipient pattern is not eligible for the Timer Expired add-on.

Finally, the Critical Status captures the possibility of unexpected accidents, while the Safety Violation add-on the possibility that the human might ignore a safety measure. However, the limited duration of the interactive phase leads to a reduced impact of this error on the HumanRecipient pattern.

8.2.5 HumanCompetitor Pattern

The HumanCompetitor pattern captures the human and the robot racing towards a certain location and is, thus, the only non-cooperative pattern.

The standard operational states are the human *moving* to the requested location, then either *waiting* for the robot to return to its original position (if the robot wins the competition) or *return* to their initial position themselves (if the human wins the competition).

Starting to walk and stopping are actions initiated by the human, both extended through the Free Will add-on, capturing the possibility that the human may get distracted or waste time while trying to reach the item's location.

Erroneous add-ons applied to the HumanCompetitor pattern do not *directly* impact the outcome of the mission. Still, they result in a higher chance of the robot winning the competition. Therefore, errors that occur amidst a HumanCompetitor service increase the impact on the overall mission outcome of errors that may occur amidst services provided by the robot if it wins the competition.

8.2.6 HumanRescuer Pattern

The HumanRescuer pattern captures the mirrored situation compared to the HumanApplicant, i.e., the robot requiring the human's support in performing a task (such as opening a door or placing an item on the robot's tray).

The standard behavior features three phases, modeled by as many $\langle \mathsf{op} \rangle_\mathrm{pub}_h$ instances: the human in *idle* state, the human *walking* towards the robot after noticing the signal requesting support, and the human *performing* the task requested by the robot.

Given the similar structure, this SHA is extended through the same add-ons as the HumanApplicant pattern described in Section 8.2.1.

Deciding to help the robot and move to its location is an action initiated by the human, extended through the Free Will add-on. It is the robot, instead, that instructs the human to begin the task when they are sufficiently close: the *walking* operational state is, thus, eligible for the Disobey/Obey add-on.

It is possible that the human is distracted by concurrent tasks before they are able to assist the robot; thus, the Timer Expired add-on imposes an upper bound on the time they take to reach the robot from their initial location.

As in the Applicant pattern, a critical situation may occur that requires

safety measures to be enforced while the human is performing the required task, which is captured by an additional $\langle \mathsf{op} \rangle_\mathrm{pub}_h$ instance, subject to the Safety Violation add-on.

Finally, as in previous patterns, the Critical Status add-on captures the possibility of unexpected accidents while the human supports the robot.

CHAPTER 9

Experimental Analysis of Human Errors' Impact

This chapter reports the experimental validation performed to assess the added value of the erroneous behavior add-ons to the model-driven development framework.

Three scenarios, inspired by the healthcare setting and featuring three different robotic missions (i.e., sequences of services), have been developed through the model-driven framework presented in Chapter 4.

We perform design-time analysis with the formal model devoid of erroneous behavior add-ons and, subsequently, with the extended SHA modeling human behavior presented in Section 8.2. The comparative analysis allows us to observe how different human errors impact different robotic missions and how introducing this aspect into the mission's design process can guide the practitioner toward forward-looking management choices.

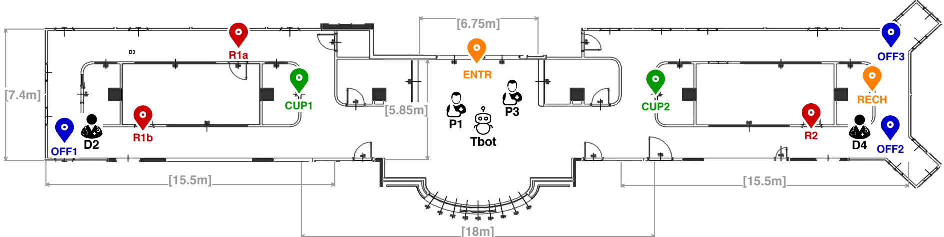


Figure 9.1: *Floor layout used for scenarios* DP1 *and* DP2*. Agents (*P1, P3, D1, D4, *and* Tbot *are represented in their starting positions). Location symbols mark the position of Points of Interest (POIs): entrance and robot's recharge station are shown in orange, waiting room and emergency room doors are in red, cupboards in green, and doctors' offices are in blue. Wall lengths (in meters) are also reported.*

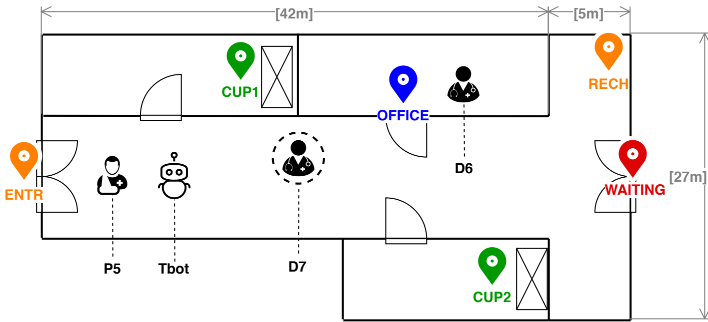


Figure 9.2: *Floor layout for scenario* DP3*, color-coded as Fig. 9.1. Since* D7*'s starting position is randomized, the displayed location is purely representative.*

9.1 Experimental Setting

The developed experimental scenarios capture a service robot assisting Doctor/Patient pairs (one or multiple) and are hereinafter referred to as DP1, DP2, and DP3. Scenarios are designed to capture realistic robotic missions featuring the complete set of services, highlight the flexibility of the overall framework, and test the impact of erroneous behavior models in a wide range of situations.

Scenarios DP1 and DP2 are set in the floor layout in Fig. 9.1. Fig. 9.1 shows the planimetry of the third floor of Building 22 of Politecnico di Milano, whose areas are featured in the two scenarios as three doctors' offices, a waiting room, and an emergency room. Fig. 9.2 depicts the layout for DP3, as presented in [130], featuring a T-shaped corridor with a waiting room, a doctor's office, and two rooms with cupboards containing medical kits. Table 9.1 summarizes the missions captured by each scenario.

Table 9.1: *Scenarios used for the validation phase (abbreviation, detailed description, and sequence of services constituting the mission).*

SCENARIO	DESCRIPTION	MISSION
DP1	The robot (Tbot) serves a patient-doctor pair (P1/D2, respectively). The robot meets the patient by the entrance (ENTR) and *leads* them to the waiting room (R1b) to wait for the doctor to visit them. The robot *follows* the doctor to CUP1 where they fetch required tools, and *follows* them back (carrying the tools) to the examination room (R2) where the patient will receive the treatment. Finally, the robot returns to R1b and *escorts* the patient to R2, where the doctor is waiting.	P1 Follower, D2 Leader, D2 Leader, P1 Follower
DP2	The robot (Tbot) serves two patient-doctor pairs (P1/D2 and P3/D4). The robot meets P1 by the entrance (ENTR) and *leads* them to the waiting room (R1a), then it performs the same task for P3 *leading* them from the entrance to R1b. The robot fetches the first required medical kit from CUP1 and *delivers* it to D2 at OFF1. The robot then serves D4 by *following* them to CUP2 and back to their office (OFF3) while carrying the kit. Finally, the robot *leads* P1 to OFF1 and P3 to OFF3 as both doctors are ready to visit them.	P1 Follower, P3 Follower, D2 Recipient, D4 Leader, D4 Leader, P1 Follower, P3 Follower
DP3	The robot (Tbot) serves a doctor patient pair (P5/D6) while a second doctor (D7) is active on the same floor. The robot *escorts* P5 to the waiting room. Then it *competes* with D6 for a resource in CUP1. If the robot wins the competition (referred to as **PLAN a**), it requires D6's *help* in opening the office door and then *delivers* them the fetched item in OFFICE. If D7 wins (referred to as **PLAN b**), D6 *leads* the robot to CUP2 to fetch the required item and has the robot carry it back to the office. Irrespective of the competition outcome, when D6 is ready to treat the patient, the robot *escorts* P5 from the waiting room to the office and then *assists* D6 in administering the medication.	P5 Follower, D7 Competitor, **PLAN a:** D6 Rescuer D6 Recipient **PLAN b:** D6 Leader D6 Leader P5 Follower D6 Applicant

Although our framework supports multi-robot teams [130], the three scenarios feature only one robot, indicated as Tbot, since this chapter focuses

Table 9.2: *Developed erroneous behavior profiles. Each profile has an identifier and the associated likelihood of occurring for each add-on.*

	Dis./ Ob.	Fr.W.	T.Exp.	S.Viol.	Cr.St.
Normal	O	O	O	O	O
Inattentive	CR	O	O	O	O
Focused	NS	O	O	O	O
Busy	O	CR	CR	O	O
Available	O	NS	NS	O	O
Inexperienced	O	O	O	CR	O
Experienced	O	O	O	NS	O
Critical	O	O	O	O	CR
Stable	O	O	O	O	NS

*O = Ordinary, *NS = Not Significant, *CR = Critical

on human behavior modeling. In all three scenarios, the employed service robot is a TurtleBot 3 WafflePi,[1] with an initial charge of 90% to ensure that it is sufficiently charged to complete each mission. Patients are identified as P1, P3 in DP1 and DP2, and P5 in DP3, and exhibit critical fatigue profiles, specifically Young/Sick for P1 and P5 and Elderly/Sick for P3. Doctors are identified as D2, D4 in DP1 and DP2, and D6, D7 in DP3, and all exhibit less critical fatigue profiles than the patients, specifically Elderly/Healthy for D2, and Young/Healthy for D4, D6, and D7. We recall that fatigue profiles impact the rate at which humans fatigue and recover, i.e., the values of parameters λ and ρ in Eq.6.5.

As per Fig. 4.1, scenarios are configured through the custom DSL [224], which is then automatically translated into the SHA network through the tool available at [124]. The generated formal model and set of queries are subject to SMC, performed through Uppaal v.4.1.26 on a machine with 4 cores and 16GB of memory. Performance data are reported and discussed at the end of this section.

9.2 Experimental Results

The experimental validation aims to illustrate—through the three example scenarios in Table 9.1—how the framework's design-time analysis is enriched by the introduction of erroneous behaviors. Each add-on features one or multiple parameters that can be tuned to calibrate the likelihood of

[1] Technical specification available at https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/.

the corresponding erroneous behavior. Such parameters are recapped in the following:

(a) probability weights disobey and obey for the Disobey/Obey add-on;

(b) thresholds $\mathsf{FW}_{\mathsf{th}}$ and $\mathsf{FW}_{\mathsf{max}}$ for the Free Will add-on (T equals constant $\mathsf{T}_{\mathsf{poll}}$ in our modeling approach);

(c) allowance factor δ for the Timer Expired add-on, which determines the value of upper bound TE;

(d) exponential rate λ_{safe} for the Safety Violation add-on;

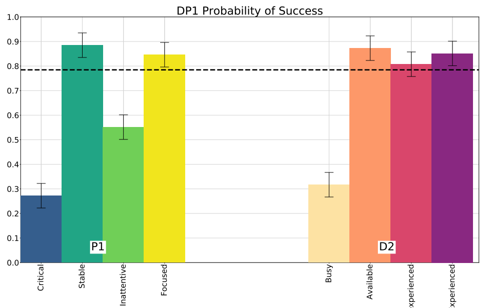
(e) rate hs and constant FS for the Critical Status add-on.

To exhaustively investigate the impact of each error on a specific scenario, it would be necessary to compute the probability of success for all possible values of such parameters within their respective domain. However, to keep the duration of a design-time analysis round within an acceptable range, we group possible values for each add-on into three macro-categories, similarly to control modes classification in the CREAM technique [115]. The identified levels are: ordinary (O), critical (CR), and not significant (NS) error likelihood.

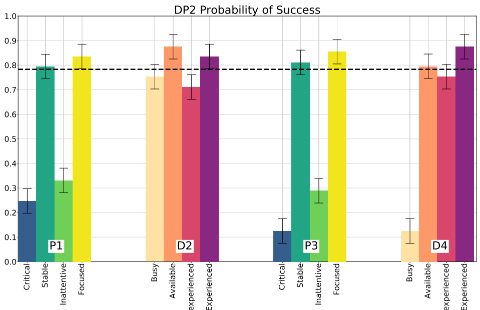
As with fatigue profiles, we have identified 9 profiles of erroneous behavior, each elevating (or dampening) the likelihood of one or multiple specific errors. Erroneous behavior profiles are summarized in Table 9.2. We remark that, given the acknowledged lack of empirical data for techniques analyzing human behavior such as Human Reliability Analysis (HRA) [51], the specific parameter values are not extracted from real datasets but arbitrarily chosen for this batch of experiments. However, the purpose of this validation is not to assess the accuracy of SMC results against real empirical data but to illustrate how the introduction of erroneous behaviors impacts design-time results and supports the mission design process. Therefore, the lack of real data has limited consequences on the results' significance.

For the impact analysis of the three scenarios, we calculate the probability of success of the mission with different erroneous behavior profiles applied to each human subject. The estimated success probabilities are reported in Fig. 9.3.

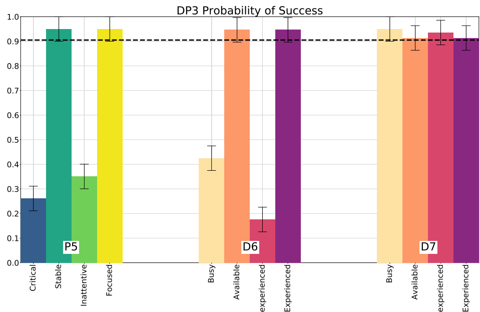
For each scenario, as explained in Section 3.1.2, we calculate the probability of success within a time bound τ through expression $\mathbb{P}_{\mathrm{M}}(\diamond_{\leq\tau}\ \mathsf{scs})$. Verification is iterated by changing the erroneous behavior profile for one human subject at a time while the value of τ remains unchanged. For the first iteration, all humans are assigned the Normal profile (see Table 9.2),



(a) *Estimated probability of success for* DP1.



(b) *Estimated probability of success for* DP2.



(c) *Estimated probability of success for* DP3.

Figure 9.3: *Bar plots reporting the estimated probability of success ($[0-1]$) for the three scenarios. Each bar represents the estimation with a different erroneous behavior profile grouped by the human subject. Dashed lines represent the success probability estimated with all human subjects' profiles set to* Normal *(see Table 9.2).*

representing the "standard" probability of success (the dashed horizontal lines in Fig. 9.3), also referred to as the "*baseline*". We recall that, as explained in Section 3.1.2, SMC results are of the form $[\mathrm{p}-\epsilon, \mathrm{p}+\epsilon]$, representing the confidence interval to which the real success probability be-

longs.

All experiments have been performed with Uppaal's default statistical parameters, specifically, the width of the estimated confidence interval is set to $\epsilon = 0.05$. In Fig. 9.3, the height of each bar equals the value of p obtained for the corresponding experiment, while black lines represent the 2ϵ-wide confidence interval. Note that none of the baseline success probability estimations is exactly 100%. The first reason behind this result is that the Normal behavioral profile features an *average* likelihood for all errors (while these are made more or less prominent by the other profiles); therefore, errors have a non-null impact on the success probability also when calculating the baseline.

Secondly, time bounds (parameter τ in each scenario) are chosen so that, should the success probability be calculated with an error-free model (i.e., without any add-on), it would equal the maximum value allowed by Uppaal with this set of parameters, which is $[0.95, 1]$. Even in this case, the SMC experiment does not yield *exactly* 100% for the probability of success because the result must be a confidence interval in any case (thus, it yields the feasible half of the confidence interval with $\mathrm{p} = 1$).

Since human subjects have different roles (i.e., either professionals or patients), not all profiles realistically apply to every subject. Specifically, verification is performed with patients (subjects P1, P3, and P5) cycling between Critical, Stable, Inattentive and Focused profiles. This set of profiles represents the fact that patients are more susceptible than professionals to accidents (captured by the Critical Status add-on) and prone to ignore the robot's instructions either due to lack of familiarity with the technology or to inattention due to their condition and surrounding environment (captured by the Disobey/Obey add-on).

On the other hand, professionals (subjects D2, D4, D6, and D7) rotate between Busy, Available, Inexperienced, and Experienced profiles. In this case, healthcare professionals are more likely to act in a hectic environment, effectively pushing them to either rush through a task (i.e., one of the phenotypes captured by the Free Will add-on) or start working on different tasks than the one involving interaction with the robot, thus exceeding the maximum allowed time-bound (captured by the Timer Expired add-on).

Moreover, professionals with little experience in working alongside a robot are more likely to erroneously step out of a safe operational state when a critical situation is still in place (captured by the Safety Violation add-on). Note that the described pairings between subjects and behavioral profiles are only conceived for the purposes of this experimental validation and do not reflect actual limitations of the approach (all profiles, including

combinations of them, are applicable to any human agent, irrespective of their role).

Fig. 9.3a displays the results for scenario DP1. With $\tau = 700\mathrm{s}$, in 5 cases out of 8, the probability of success is essentially unchanged (if not higher) compared to the one calculated in standard conditions, which is approximately 80%. Concerning human P1, these results are due to the nature of the profiles themselves: both Stable and Focused are "*positive*" profiles, as they feature lower likelihood of erroneous behaviors than the Normal profile. The same conclusion can be drawn about the Available and Experienced profiles for D2.

On the other hand, the probability of success is also unaffected by the Inexperienced profile. As a matter of fact, D2 *leads* the robot in DP1 and, since their walking speed is higher than the robot's (a healthy human walks at about $1.4\mathrm{m/s}$, whereas the robot moves at a maximum speed of $0.26\mathrm{m/s}$) it is unlikely for the human to walk too closely to the robot and trigger the enforcement (and subsequent erroneous violation) of the safety measure.

As shown in Fig. 9.3a, scenario DP1 is most affected by the Critical and Inattentive profiles for P1, and Busy for D2. As explained in Section 8.2.2, the HumanFollower interaction pattern, which P1 adheres to, is susceptible to both the Critical Status and Disobey/Obey add-ons (influenced by the Critical and Inattentive profiles), which cause the probability of success to drop to approximately 25% and 55%, respectively.

To address this issue, the practitioner designing the mission may decide to adopt additional monitoring measures regarding the patient's health status or have them walk a shorter distance to reduce the impact of unexpected accidents. Additionally, it is possible to make the robot's capability to communicate instructions more efficient to improve the patient's attention level. Concerning D2, the Busy profiles cause a 50% drop in the success probability: to address this issue, a possible design choice is to assign the mission to a different employee with a clearer schedule.

Similar conclusions can be drawn about P1 and D4 in scenario DP2, whose results are reported in Fig. 9.3b. The estimated success probability with $\tau = 1500\mathrm{s}$ in standard conditions is approximately 80%. In this case, the Critical Status add-on is more impactful for patient P3 compared to P1 due to the more critical fatigue profile, causing a steeper growth of probability weight p_{d}. The resulting success probability is slightly above 10% (compared to 25% for P1).

These results can guide the practitioner in modifying the plan of the mission to reduce the physical burden on the two patients, especially P3:

for example, by having the robot lead them, whenever possible, straight to the emergency room rather than to the waiting room first. On the other hand, since the Disobey/Obey add-on has no correlation with the evolution of fatigue, the Inattentive profile has a comparable impact on the mission's outcome when applied to P1 and P3. In this case, the same reconfiguration measures discussed for subject P1 in scenario DP1 may be applied.

In scenario DP2, although the same set of behavioral profiles are applied to D2 and D4, the different interaction patterns they participate in (i.e., HumanRecipient and HumanLeader) are differently influenced by error phenotypes.

More specifically, the Busy profile has a significantly larger impact when applied to D4 rather than D2. As described in Section 8.2.4, the HumanRecipient does not feature any instance of the Timer Expired add-on, whereas the Free Will add-on, which, like the Timer Expired add-on is made more prominent by the Busy profile, allows the subject to move *while* the robot is fetching the required object. Therefore, the human erroneously moving causes the robot to adjust the target of the delivery task to the new human's position, which does not necessarily result in a delay of the completion of the service nor lowers the success probability within time bound τ. Indeed, given the starting position of D2 (also shown in Fig. 9.1), it is more likely for them to move *closer* to CUP1 (the required object's location) than farther, leading to only a slight decline of the success probability (approximately 75% compared to 80% in standard conditions). For subject D4, instead, since they also participate in a HumanLeader pattern like subject D2 in scenario DP1, the Busy profile has a very significant impact leading to a 70% drop in the success probability compared to ordinary conditions.

The Inexperienced profile (which increases the likelihood of the Safety Violation add-on) has a comparably limited impact when applied both on D2 and D4. Concerning D4, the same conclusions drawn about the HumanLeader pattern for scenario DP1 also apply in this case. As for the HumanRecipient pattern, D2 can enter the critical *interacting* operating state at most for the amount of time required to pick up the item from the robot. Consequently, the likelihood of erroneously ignoring the safety measure leading to a collision during the interaction is also limited. In conclusion, the practitioner does not need to consider specific design choices concerning D2, while D4 is affected by the same guidelines discussed for scenario DP1.

Estimated success probabilities for scenario DP3 are shown in Fig. 9.3c, all calculated with $\tau = 600$s. In standard conditions, the mission ends suc-

cessfully with a 90% probability. Patient P5 exhibits a similar trend to that observed for P1 and P3 in scenarios DP1 and DP2, as the Critical and Inattentive profiles cause a drop of the success probability of approximately 65% and 55%, respectively.

On the other hand, given the same set of behavioral profiles, the trend is different for D6 and D7 compared to subjects covering the role of professionals in previous scenarios. We recall that, given the presence of a HumanCompetitor pattern, in this case, the robotic mission features two alternative plans depending on whether the human or the robot wins the competition [130], both summarized in Table 9.1. If the robot loses, D6 is involved in two HumanLeader interaction patterns, whose dependency on different behavioral profiles have already been discussed for subjects D2 in DP1 and D4 in DP2. Otherwise, D6 participates in a HumanRecipient, HumanRescuer sequence. The initial position of D7 is randomized to make the outcome of the competition unpredictable.

As observed in scenario DP2, the HumanRecipient pattern (involving subject D2) is only slightly affected by both "*negative*" profiles. In contrast, the outcome of the HumanRescuer pattern (see Section 8.2.6) is impacted by the Free Will, Timer Expired, and Safety Violation add-ons. The impact of the Busy profile, which makes the first two add-ons more prominent, on D6, is an average between the drop it causes on **PLAN a** (the impact of profile Busy is low for the HumanRecipient and high for the HumanRescuer patterns) and **PLAN b** (the impact of profile Busy is low for both instances of the HumanLeader pattern), resulting in an overall approximate 45% drop compared to the baseline.

Concerning the Inexperienced profile, both the HumanRescuer (featured by **PLAN a**) and HumanApplicant patterns (featured by both plans) are highly susceptible to the Safety Violation add-on, which can occur throughout the entire duration of the task they perform jointly and in close distance with the robot. Therefore, unlike in scenarios DP1 and DP2, the Inexperienced profile leads to a larger success probability drop (more than 70%) than the Busy profile. After examining these results, the practitioner designing the robotic mission may either assign the mission to a more experienced employee or invest in thorough training of the personnel in charge of performing tasks alongside the robot.

Finally, confirming the modeling choices discussed in Section 8.2.5, the HumanCompetitor pattern, in which subject D7 participates, is the least influenced by erroneous behaviors. This trait of the pattern is reflected by the results in Fig. 9.3c, showing that the success probability does not significantly change with respect to the baseline, irrespective of the erroneous

behavior profile assigned to D7. As a matter of fact, should D7 perform any erroneous action, this does not result in a failure of the service nor a delay of the overall mission, but rather it favors the "*victory*" of the robot in the competition. Therefore, the erroneous actions of D7 indirectly influence the outcome of the mission as they result in **PLAN a** being enacted more often than **PLAN b**, so erroneous behaviors with a more considerable impact on **PLAN a** also have a larger impact on the mission in its entirety.

As previously mentioned, we have selected a subset of behavioral profiles for each human subject to perform this impact analysis for the three scenarios, representing the most realistic contingencies. Consequently, we have performed four verification experiments (resulting in different success probability estimations) for each human subject plus the baseline, so 9 experiments for DP1, 17 for DP2, and 13 for DP3. With the described parameter set, verification ends in 66.72min for DP1, 133.95min for DP2, and 104.64min DP3 (these durations refer to the *cumulative* time required to complete *all* experiments—thus the complete analysis of errors' impact—for each scenario).

Given the user-friendliness and flexibility of the scenario configuration phase, the practitioner can easily modulate the number of experiments to be performed (thus, the duration of the design-time analysis round). Modulation is performed by selecting the combinations of behavioral profiles found to be more critical or more likely to be observed in their specific application.

Part II

Scenario Deployment

CHAPTER 10

Deployment Approach

This chapter focuses on the deployment phase of the framework.[a]

Firstly, the architecture of the deployment approach (summarized by Fig. 10.1) is presented in detail with the correspondence between deployment components and formal model entities.

The mechanism to convert formal model elements into executable code is then introduced, followed by a detailed presentation of the resulting deployable code patterns corresponding to recurring formal modeling patterns.

[a]The content presented in this chapter also appears in [128]. The author of this thesis declares to have also authored the reproduced text, figures, and data and to have the right to reproduce such content in a dissertation according to the license under which both articles are published.

10.1 Deployment Framework Architecture

Once the results at design-time are deemed adequate, the scenario can be either deployed in a real environment or simulated, as per Fig. 4.1. The main elements of the deployment framework are the executable units replicating agents' behavior, the executable orchestrator, the middleware layer, and the environment. The high-level role they play in the framework is explained

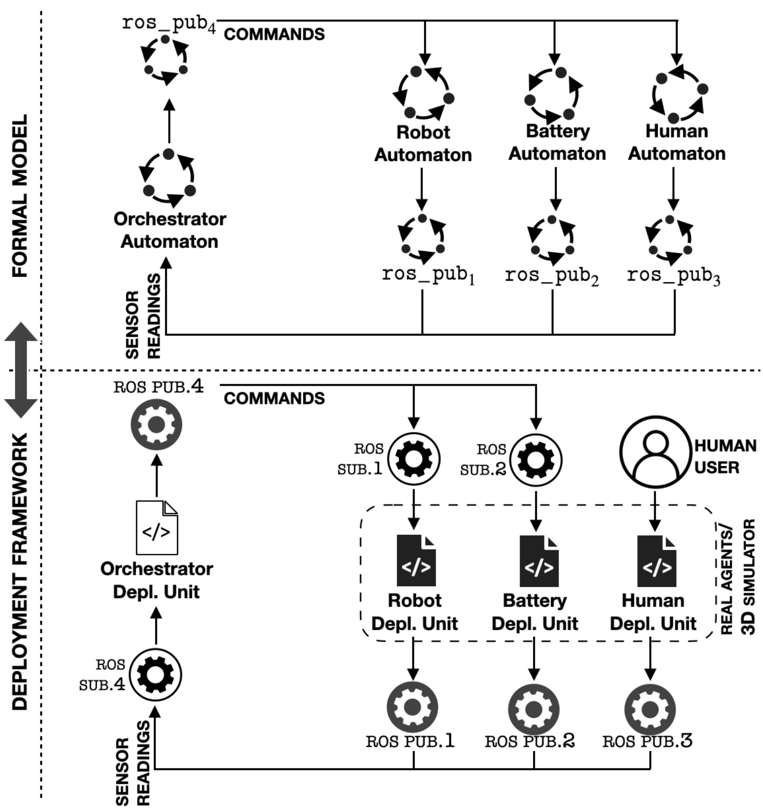


Figure 10.1: *Mapping between the formal model and the deployment infrastructure, as seen in [132]. The SHA for robots, batteries, and humans are mapped to deployment units. The orchestrator is mapped to a standalone script communicating with the agents through ROS nodes. Sensor readings are shared with the orchestrator by a publisher node for each agent. The orchestrator sends its commands through a fourth publisher to all agents but the human, which is directly controlled by the human user.*

in Chapter 4. In the following, we explain in detail how each formal model feature is mapped to an equivalent deployable form so that the deployed system and the formal model behave correspondingly.

The high-level mapping between the formal model and the deployment framework is depicted in Fig. 10.1. Each automaton of the SHA network, excluding instances of $\texttt{ros_pub}_{\langle \text{id} \rangle}$, corresponds to an atomic entity of the deployment model. All $\texttt{ros_pub}_{\langle \text{id} \rangle}$ instances correspond to a real ROS publisher node (and queue). Nodes internal to the simulator are implemented using the ROS interface provided by the CoppeliaSim API framework [185], while the nodes related to the orchestrator are implemented

using the `rospy` library.[1]

This architecture allows the analyst to choose between deploying the application in a real, virtual, or hybrid environment without changing the code. In the case of a virtual environment, robots, batteries, and humans are actuated through scripts internal to the simulator. If the application is deployed in a real environment, they correspond to the real entities endowed with suitable sensors. The orchestrator is implemented through a standalone script in both cases.

Although the framework focuses on assistive robotics applications, the model-to-code mapping principle is not exclusive to this domain. A cyber-physical system is eligible for this technique if it consists of distributed agents that periodically share sensor readings through a publish-subscribe middleware technology and a centralized controller that monitors the system's state and sends instructions accordingly.

10.2 Model-to-Code Mapping

We exploit the stochastic features of SHA presented in Chapter 6 to capture specific sources of uncertainty existing in reality, i.e., human behavior and unpredictable network delays. Other elements of the SHA network represent either electronic devices or software modules whose behavior is fully deterministic. Therefore, although all automata in the network are defined as SHA, only those modeling humans and the $\mathtt{ros_pub}_{\langle \mathsf{id} \rangle}$ instances show a stochastic behavior. For all other SHA in the network, the following properties hold:

1. there are no edges that may fire with unbounded delay, thus described by an exponential distribution;

2. for all states (l, ν), with $l \in L$ and $\nu \in \mathbb{R}^W$, there is only one delay $d \in \mathbb{R}_+$ such that $\mu(l, \nu)(d) = 1$ holds, whereas, for all other $d' \in \mathbb{R}_+$ such that $d' \neq d$, $\mu(l, \nu)(d')$ equals 0. This means that, for each state, there is always only one delay which allows an uncontrollable switch to fire: thus, it is necessarily assigned probability 1;

3. all edges have probability weight 1.

The aspects modeled by the stochastic features are naturally present in the deployment environment and do not require explicit modeling in the deployment infrastructure.

[1] `rospy` documentation available at: http://wiki.ros.org/rospy

Table 10.1: *Mapping relations between features of an SHA $\mathcal{A}$ and a deployment unit $\mathcal{D}$, as defined by function Δ.*

Automaton Feature ($\mathbf{X}$)	$\longrightarrow$	**Deployment Unit Feature** (*simulated/*real/*both) ($\mathbf{\Delta}(\mathbf{X})$)
Location ($l \in L$)	$\longrightarrow$	Agent State ($\sigma \in \Sigma$)
Real-valued Variable ($v \in W \setminus (X \cup V_{\mathrm{dc}} \cup K)$)	$\longrightarrow$	Physical Variable ($\upsilon \in \Upsilon \setminus (X \cup \Upsilon_{\mathrm{s}} \cup K)$)
Clock ($t_x \in X \subset W$)	$\longrightarrow$	Variable Uniform to Time ($\chi \in X \subset \Upsilon$)
Dense-Counter Variable ($v \in V_{\mathrm{dc}} \subset W$)	$\longrightarrow$	Sensor Reading ($\upsilon \in \Upsilon_{\mathrm{s}} \subset \Upsilon$)
Numerical Constant ($k \in K \subset W$)	$\longrightarrow$	Numerical Constant/Design Parameter ($\kappa \in K \subset \Upsilon$)
Flow-Condition ($f \in \mathcal{F}(l)$)	$\longrightarrow$	Physical Law Implementation ($\phi \in \Phi$)
Invariant ($i \in \mathcal{I}(l)$)	$\longrightarrow$	Conditional Expression ($s \in S(\Upsilon)$)
Robot/Battery-related Channel ($\mathtt{c} \in (C_{\mathrm{r}} \cup C_{\mathrm{b}}) \subset C$)	$\longrightarrow$	ROS Topic ($\mathtt{t} \in (T_{\mathrm{r}} \cup T_{\mathrm{b}}) \subset T$)
Human-related Channel ($\mathtt{c} \in C_{\mathrm{h}} \subset C$)	$\longrightarrow$	Keyboard Input/Human Behavior ($\mathtt{t} \in T_{\mathrm{h}} \subset T$)
Edge ($e \in \mathcal{E}$)	$\longrightarrow$	Conditional Statement ($\beta \in B$)

To guarantee that the deployed system is a sound transposition of the original network of automata, we define mapping function Δ that maps an SHA $\mathcal{A}$ to $\mathcal{D}$, a generic atomic entity of the deployment framework.

An atomic deployment unit $\mathcal{D} = (\Sigma, \Upsilon, \Phi, S, T, B, \sigma_{ini})$ consists of the following artefacts:

1. the set Σ of agent *states*, including the initial one σ_{ini};
2. the set Υ of *variables*, including sensor readings ($\Upsilon_s \subset \Upsilon$), constant parameters ($K \subset \Upsilon$), clocks ($X \subset \Upsilon$), and *physical* variables (real or simulated) ($\Upsilon \setminus (X \cup \Upsilon_s \cup K)$);
3. the set Φ, which contains *physical laws* in case of deployment in the real world, or the *implementation* of such laws in the case of simulation;
4. the set $S(\Upsilon)$ of *conditional expressions* on variables in Υ;
5. the set T of ROS topics over which *messages* (or *commands*) that trigger a change of state are published, with subsets $T_{\mathrm{r}}, T_{\mathrm{b}}$, and T_{h} related respectively to the robot, the battery, and the human;
6. the set $B \subset \Sigma \times T \times S(\Upsilon) \times A(\Upsilon) \times \Sigma$ of *conditional statements* governing the control flow, where $A(\Upsilon)$ is the set of assignment instructions executed on variables in Υ.

The definition of $\mathcal{D}$ is to be strictly followed while developing the orchestrator script (both in simulated and real environments) and the agents' scripts for the simulator. As for the agents' deployment units in a real environment, since this involves actual robotic systems and humans, this definition must be intended as a high-level guideline to identify the correspondence between the formal and real systems.

Two reasons underlie this discrepancy in the interpretation. Firstly, the robotic system's code might vary significantly depending on the specific manufacturer and model and might not be fully accessible to the public. Secondly, artifacts composing the human deployment unit should not be interpreted in a software engineering-specific sense but as abstract elements constituting the human decision-making process. An explanatory example is the set S that cannot contain classic Boolean expressions for the human but notionally corresponds to the set of questions that someone (consciously or not) ponders to make a decision [205].

Table 10.1 displays how function Δ maps each element of a SHA $\mathcal{A}$ to a deployment unit in $\mathcal{D}$ and is presented in detail in the following.

Each *non-committed* location in L corresponds to a state in Σ, that is to say, a block of code that defines the *behavior* of an agent under certain circumstances (e.g., the human walking or standing still).

Dense-counter variables in $V_{\text{dc}} \subset W$ have an equivalent variable in set $\Upsilon_{\text{s}} \subset \Upsilon$. In particular, variables representing sensor readings are periodically updated with frequency $1/\mathsf{T}_{\mathsf{poll}}$ like dense-counter variables in the formal model.

Constants in $K \subset W$ are mapped to constant parameters in $K \subset \Upsilon$, which match the design parameters of the physical equipment.

Each real-valued variable in $W \setminus (X \cup V_{\text{dc}} \cup K)$ matches a physical variable in $\Upsilon \setminus (X \cup \Upsilon_{\text{s}} \cup K)$, whereas flow conditions in F correspond to physical laws (or their implementations) in Φ.

Each clock in $X \subset W$ is mapped to a particular case of variable in $X \subset \Upsilon$ that evolves uniformly with time. As for the orchestrator, staying in a specific location until clock $t_x \in X \subset W$ reaches threshold $k \in K \subset W$ is implemented as a `sleep`(κ) instruction, where $\kappa \in K \subset \Upsilon$ is expressed in *seconds*. This binds the behavior of the orchestrator deployment unit to the *system* time, which can be considered an element of set $X \subset \Upsilon$.

Time in the simulated environment is discrete with a time-step Δt that has a minimum value of $10ms$. Given the system's time variables sizing (e.g., $\mathsf{T}_{\mathsf{poll}}$ equals $1s$, which is two orders of magnitude greater than Δt), the error caused by the discretization interval has a negligible impact on the system's behavior and, therefore, on the results of the analysis.

Let us consider, for example, the model of fatigue F while the human is walking in Eq.6.5. To prevent the human from reaching complete exhaustion, the orchestrator instructs them to stop when it detects that $F = 0.7$ holds. If the reaching of this threshold is detected at time t_{stop} with a continuous-time model, in simulation it is detected at time $t_{\text{stop}} + \Delta t$ at the latest. With a critical fatigue profile, e.g., with a mean $\lambda = 0.025$, the value of F when the reaching of the threshold is detected converges to $F(t_{\text{stop}} + \Delta t) \approx 0.700075$, which approximately corresponds to a $0.11‱$ error. Similar conclusions can be drawn about the other physical variables.

Considering that the nature of the system does not require a sharp real-time synchronization among the various components, we can reasonably conclude that the order of magnitude of the errors does not critically threaten the model-to-code transposition soundness.

As for commands issued by the orchestrator, it is necessary to make a distinction between the ones destined for the robot or the battery and the ones destined for humans.

In the first case, triggering an event through channel $\mathsf{c} \in (C_r \cup C_b) \subset C$

corresponds to publishing a message over a ROS topic $\mathtt{t} \in (T_r \cup T_b) \subset T$. The format of these messages is fixed and defined a priori.

Within the SHA network, the orchestrator shares its commands with the human through channels identically to what it does with the robot. The unpredictability of human behavior is embedded in the formal model in terms of stochastic features. At deployment time, these stochastic approximations are lifted by having an actual human either provide keyboard input (in simulation) or directly operate in the environment. In the latter case, human "inputs" are not explicit (e.g., when a human stops walking) and need to be inferred from sensors data. Therefore, human-related channels in $C_\mathrm{h} \subset C$ match user inputs $\mathtt{t} \in T_\mathrm{h} \subset T$.

Edges in $\mathcal{E}$ are translated to scripts' conditional statements and callback functions in B, representing the switch of an agent from state $\sigma \in \Sigma$ to a new state $\sigma' \in \Sigma$.

As discussed in Chapter 6, edges can model controllable and uncontrollable switches. Controllable switches are triggered by ROS messages, i.e., commands sent by the orchestrator.

Uncontrollable switches occur in correspondence with the intersection between values that satisfy $i \in \mathcal{I}(l)$ and values that satisfy condition $\gamma \in \Gamma(W)$ on the outgoing edge. An edge $e = (l, \epsilon, \gamma, \xi, l')$ modeling an uncontrollable switch is mapped to conditional statement (i.e., *if-then* construct) $\beta = (\sigma, \perp, s, a, \sigma')$, where states σ and σ' map locations l and l', respectively. Condition γ is mapped to the conditional expression $s \in S(\Upsilon)$ guarding β. Update instructions ξ are mapped to $a \in A(\Upsilon)$, representing the (set of) assignment instruction(s) performed as soon as β is executed and s evaluates to true.

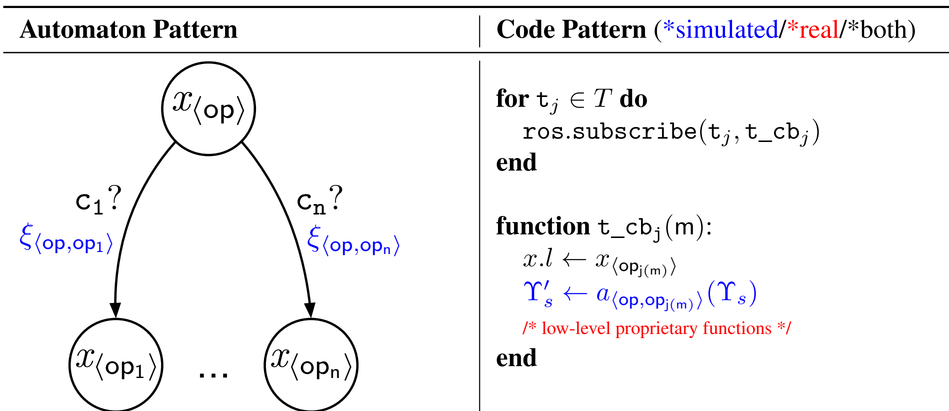
10.3 Deployable Code Patterns

Applying function Δ to recurrent modeling patterns in the SHA model leads to recurrent code patterns, presented in the following. Lines in Table 10.2, 10.3, 10.4, and 10.5 are color-coded to highlight differences between deployment in an actual and simulated environment.

Hereinafter, while presenting code patterns, we use the notation a $(\rightarrow b \in B)$ to indicate how a formal model element a is mapped to a deployment unit element $b \in B$: for example, the fact that guard condition γ_j is mapped to conditional expression s_j is expressed as γ_j $(\rightarrow s_j \in S(\Upsilon))$.

Automata modeling the robot, the battery, and the humans are generically labeled as x, whereas the location capturing a generic operating condition $\langle \mathsf{op} \rangle$ of agent x is labeled as $x_{\langle \mathsf{op} \rangle}$. The pattern featuring locations

Table 10.2: *Controllable Switch pattern model (on the left) to code (on the right) transformation, as seen in [128]. Color-coding for the automaton is the same as in Fig. 6.2, except for channel labels which, for visualization purposes, are black instead of red. The code pattern highlights mutually exclusive lines that are present if the application is simulated (in dark blue) or deployed in a real environment (in red).*



labeled as o is instead a subcomponent of the orchestrator automaton and is, thus, realized by a standalone script.

As for the patterns for simulation scripts (described in Section 10.3.1, Section 10.3.2, and Section 10.3.3), a further remark is necessary about their apparently non-cyclical nature. The scripts implement a standard interface provided by the simulator. All the code blocks shown in Table 10.2, 10.3, and 10.4 belong to a function of the interface that deals with agents' *actuation* in the scene and is, by default, re-run at each time step ($\Delta t = 10$ms) throughout the whole simulation.

10.3.1 Controllable Switch Pattern

The first pattern, shown in Table 10.2, is the controllable switch. The resulting code pattern consists of:

- a **for** loop so that, during initialization, agent x subscribes to ROS topics $\mathtt{t_j} \in T$;
- a **callback function** $\mathtt{t_cb_j}$ (one for each subscribed topic) which is invoked every time a new message $\mathtt{m}$ is received through topic $\mathtt{t_j}$: within the function, operating condition $x.l$ is updated, then assignment instructions $a_{\langle op, op_{j(m)} \rangle}$ are executed.

A controllable switch occurs when a change of operating condition from

$\langle \mathsf{op} \rangle$ to $\langle \mathsf{op_j} \rangle$ is appropriate according to the orchestrator policies. In the formal model, this causes the related channel $\mathtt{c_j}$ to fire; the orchestrator triggers an event ($\mathtt{c_j}!$) and an agent reacts to it ($\mathtt{c_j}?$). In the deployment model, commands are shared via ROS. Therefore, the orchestrator script publishes the command through a dedicated ROS publisher node. The script internal to the simulator corresponding to the destination agent, having subscribed to ROS topic $\mathtt{c_j}$ ($\rightarrow \mathtt{t_j} \in T$) at the beginning of the simulation/execution, receives such command over the same topic.

While in the corresponding status $\langle \mathsf{op} \rangle$, the agent is constantly listening on topic $\mathtt{c_j}$ ($\rightarrow \mathtt{t_j} \in T$). As soon as a new message is published, the script executes the callback function, generically called $\mathtt{t_cb}_j$.

In the real script, the whole ROS message m is passed as an input parameter to the callback function. The target operating condition is a function of message m and is, thus, with a slight abuse of notation, referred to in Table 10.2 as $\langle \mathsf{op_{j(m)}} \rangle$.

The content of function $\mathtt{t_cb}_j$ replicates the corresponding edge of the automaton: firstly the variable $x.l \in \Upsilon_\mathrm{s} \subset \Upsilon$ that keeps track of agent x's current location is updated to $x_{\langle \mathsf{op_{j(m)}} \rangle}$. Subsequently, the actions performed (or simulated) by agent x in reaction to the command are captured by update instructions $\xi_{\langle \mathsf{op}, \mathsf{op_{j(m)}} \rangle}$ ($\rightarrow a_{\langle \mathsf{op}, \mathsf{op_{j(m)}} \rangle} \in A(\Upsilon)$). As per Table 10.2, in case of deployment in a real setting, this would correspond to the execution of proprietary functions dealing with lower-level tasks (e.g., trajectory planning).

10.3.2 Uncontrollable Switch Pattern

The uncontrollable switch pattern, shown in Table 10.3, consists of:

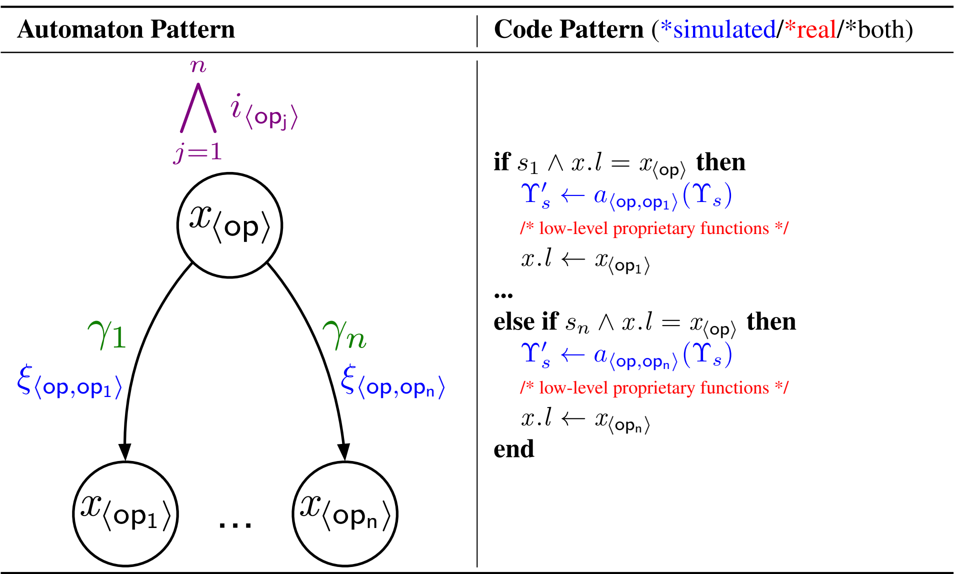
- n **if-then-else** statements $\beta_j \in B$ guarded by as many conditional expressions $s_j \wedge x.l = x_{\langle \mathsf{op} \rangle}$, where $s_j \in S(\Upsilon)$ holds for all $j \in [1, n]$: when one of conditions s_j evaluates to true, assignment instructions $a_{\langle \mathsf{op}, \mathsf{op_j} \rangle}$ are executed.

Combining the modeling patterns that formally model the scenario in our framework leads to uncontrollable switches which are guaranteed to be *well-formed*, that is, that have the following features:

1. given the invariant i and guard γ associated with the switch:

    a) $|N_i \cap N_\gamma| > 0$ holds for all uncontrollable switches, where $N_i \subset \mathbb{R}^W$ and $N_\gamma \subset \mathbb{R}^W$ are the sets of valuations that satisfy i and γ respectively;

Table 10.3: *Uncontrollable Switch pattern model-to-code transformation, as seen in [128]. Color-coding is the same as in Table 10.2 for both columns.*



Automaton Pattern	Code Pattern (*simulated/*real/*both)
	if $s_1 \wedge x.l = x_{\langle op \rangle}$ **then** $\Upsilon'_s \leftarrow a_{\langle op,op_1 \rangle}(\Upsilon_s)$ /* low-level proprietary functions */ $x.l \leftarrow x_{\langle op_1 \rangle}$ **...** **else if** $s_n \wedge x.l = x_{\langle op \rangle}$ **then** $\Upsilon'_s \leftarrow a_{\langle op,op_n \rangle}(\Upsilon_s)$ /* low-level proprietary functions */ $x.l \leftarrow x_{\langle op_n \rangle}$ **end**

b) there exists at least one real-valued variable or clock $v \in W \setminus V_{\text{dc}} \cup K$ such that $\nu_{\text{i,var}}(v) = \nu_{\text{j,var}}(v) = \overline{v}$ holds for some $\overline{v} \in \mathbb{R}$ for all $\nu_{\text{i,var}} \in N_i$ and all $\nu_{\text{j,var}} \in N_j$;

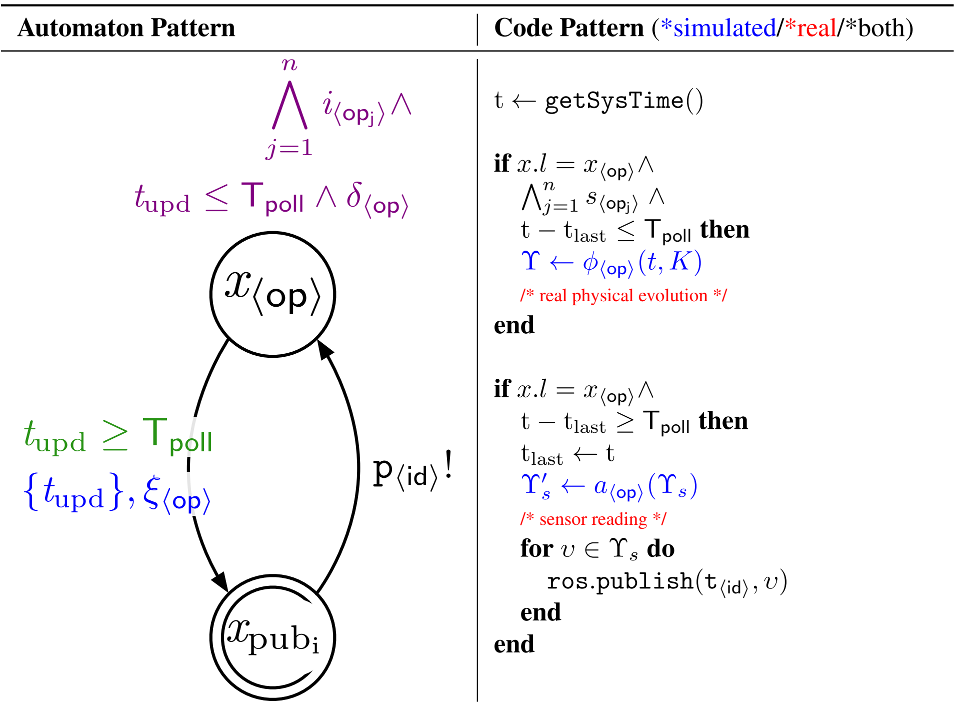
2. given n uncontrollable edges outgoing from the same location and guarded by as many γ_j conditions, such conditions must be *disjoint*: $\bigcap_{j=1}^{n} N_{\gamma_j} = \emptyset$ holds, where N_{γ_j} is the set of valuations satisfying γ_j.

In case of an uncontrollable switch, as in Table 10.3, the edge from location $x_{\langle op \rangle}$ to $x_{\langle op_j \rangle}$ is guarded by condition γ_j and $i_{\langle op_j \rangle}(\rightarrow s_j \in S(\Upsilon))$ is a member of the invariant of $x_{\langle op \rangle}$. The way in which invariants $i_{\langle op_j \rangle}(\rightarrow s_j \in S(\Upsilon))$ are enforced in scripts is explained in Section 10.3.3, here we focus only on how outgoing edges are translated into code.

As explained in Chapter 6, the automaton is forced to switch to $x_{\langle op_j \rangle}$ when variable and clock values simultaneously satisfy both $i_{\langle op_j \rangle}$ and γ_j. Note that, as previously mentioned, in our model guard conditions on uncontrollable edges are guaranteed to be disjoint. For this reason, in the corresponding code pattern reported in Table 10.3, it is correct that, when one of the $\gamma_j \wedge x.l = x_{\langle op \rangle}$ conditions is verified, no other *if* branch is visited. Constraint $x.l = x_{\langle op \rangle}$ is necessary because the location might also be updated by a callback function, as explained in Section 10.3.1.

Table 10.4: *Sensor Reading pattern ($\langle \mathsf{op} \rangle_\mathsf{pub}_{\langle \mathsf{id} \rangle}$) model-to-code transformation, as seen in [128]. Color-coding is the same as in Table 10.2 for both columns.*

Automaton Pattern	Code Pattern (*simulated/*real/*both)



As soon as a γ_j ($\rightarrow s_j \in S(\Upsilon)$) condition becomes true, update $\xi_{\langle \mathsf{op,op_j} \rangle}$ ($\rightarrow a_{\langle \mathsf{op,op_j} \rangle} \in A(\Upsilon)$) is executed (or proprietary functions are invoked). Finally, $x.l$ is updated to $x_{\langle \mathsf{op_j} \rangle}$.

10.3.3 Sensor Reading Pattern

The third pattern is the one presented as $\langle \mathsf{op} \rangle_\mathsf{pub}_{\langle \mathsf{id} \rangle}$ in Chapter 6, which consists of:

- an **update** of variable $\mathrm{t} \in \Upsilon$ through function `getSysTime()` that is specific to the employed simulator;
- an **if-then** statement $\beta_1 \in B$ which is executed if: agent x is in operating condition $x_{\langle \mathsf{op} \rangle}$, expression $s_{\langle \mathsf{op_j} \rangle} \in S(\Upsilon)$ is true for all $j \in [1, n]$, and $\mathrm{t} - \mathrm{t_{last}} \leq \mathsf{T_{poll}}$ holds. If all of these conditions hold, physical variables evolve according to laws $\phi_{\langle \mathsf{op} \rangle}$;
- an **if-then** statement $\beta_2 \in B$ which is executed if: agent x is in operating condition $x_{\langle \mathsf{op} \rangle}$, and $\mathrm{t} - \mathrm{t_{last}} \geq \mathsf{T_{poll}}$ holds. Assignment instruc-

tions $a_{\langle \mathsf{op} \rangle}$ are subsequently executed and each sensor reading $\upsilon \in \Upsilon_s$ is published on topic $\mathtt{t}_{\langle id \rangle} \in T$ through a **for** loop.

As per Table 10.4, every time this block of code is reached, variable $\mathrm{t} \in X \subset \Upsilon$ storing time is updated using proprietary functions provided by the system [185]. In the automaton, the value of clock t_{upd} is compared against $\mathsf{T}_{\mathsf{poll}}$ to check whether a new sensor reading is available. As for the code, variable $\mathrm{t}_{\mathrm{last}} \in \Upsilon_{\mathrm{s}} \subset \Upsilon$ keeps track of the time at which the previous sensor measurement was published. It follows that the expression $\mathrm{t} - \mathrm{t}_{\mathrm{last}}$ is uniform to clock t_{upd}.

If all invariants $i_{\langle \mathsf{op_j} \rangle}(\rightarrow s_{\langle \mathsf{op_j} \rangle} \in S(\Upsilon))$ hold, the simulated physical variables are then updated according to laws $\delta_{\langle \mathsf{op} \rangle}$ $(\rightarrow \phi_{\langle \mathsf{op} \rangle} \in \Phi)$. If the condition guarding the end of a sensor's refresh period (i.e., $\mathrm{t} - \mathrm{t}_{\mathrm{last}} \geq \mathsf{T}_{\mathsf{poll}}$) is satisfied, $\mathrm{t}_{\mathrm{last}}$ is updated and variables in $\Upsilon_{\mathrm{s}} \subset \Upsilon$ corresponding to sensor readings are updated as required by $\xi_{\langle \mathsf{op} \rangle}$ $(\rightarrow a_{\langle \mathsf{op} \rangle} \in A(\Upsilon))$.

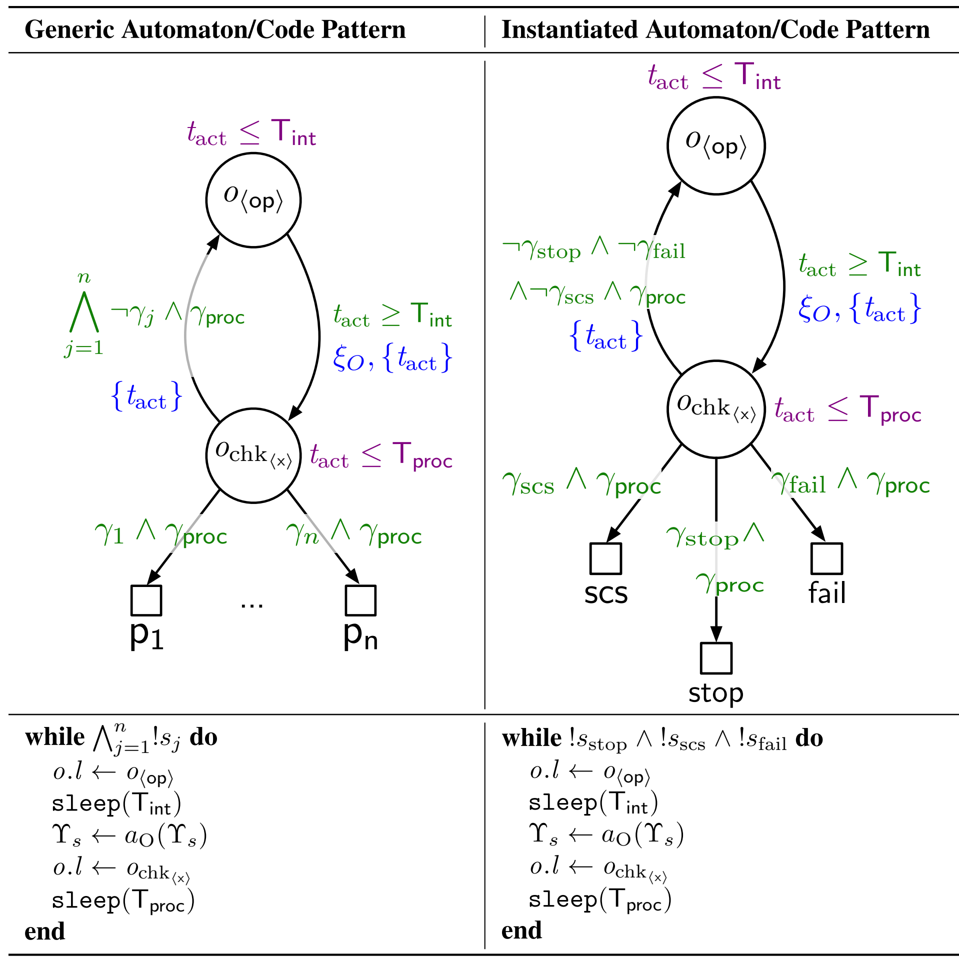
Since the committed location prescribes that no time elapses before the following instruction is executed (i.e., triggering channel $\mathtt{p}_{\langle \mathsf{id} \rangle}$), the script immediately instructs a dedicated ROS node to publish the updated sensor readings on topic $\langle \mathsf{id} \rangle$ through the ROS interface provided by the simulator. At this point, message publication occurs asynchronously with respect to the agent's script. As a matter of fact, ROS handles the publisher's queue independently of the script, which in the formal model is captured by a specific instance of template $\mathtt{ros_pub}_{\langle \mathsf{id} \rangle}$ presented in Chapter 6.

10.3.4 System Monitoring Pattern

The final pattern in Table 10.5 is the subcomponent $\langle \mathsf{op} \rangle$`_chk` [125], which is applied to all $\langle \mathsf{op} \rangle$ locations of the *orchestrator*. The purpose of this pattern is to periodically check the state of the system every $\mathsf{T}_{\mathsf{int}}$ time instants against a set of policies and, if necessary, send commands to the agents (e.g., stop the robot if it has reached the destination). Note that the pattern is also applicable to other systems with a likewise behavior (i.e., sampling-based system monitoring with custom policy enforcement). While Table 10.5 displays the generic pattern in the left column, in the following, to explain how the pattern works, we exploit the specific instance from our framework, which is shown on the right in Table 10.5.

We recall that the orchestrator controls the execution from a high abstraction level. Using as reference the abstraction levels proposed by Lutz et al. [144], the orchestrator in our framework operates at the *task* level, meaning that it manages when and how the robot does something, irrespective of the underlying implementation. All the lower-level details (e.g., the

Table 10.5: *System Monitoring pattern model-to-code transformation, as seen in [128]. The generic automaton and code patterns are shown on the left, whereas the right column features the specific instance of this pattern in our framework. Color-coding is the same as in Table 10.2 for both columns.*



Generic Automaton/Code Pattern	Instantiated Automaton/Code Pattern
while $\bigwedge_{j=1}^{n} !s_j$ **do** $o.l \leftarrow o_{\langle op \rangle}$ sleep(T_{int}) $\Upsilon_s \leftarrow a_O(\Upsilon_s)$ $o.l \leftarrow o_{chk\langle x \rangle}$ sleep(T_{proc}) **end**	**while** $!s_{stop} \wedge !s_{scs} \wedge !s_{fail}$ **do** $o.l \leftarrow o_{\langle op \rangle}$ sleep(T_{int}) $\Upsilon_s \leftarrow a_O(\Upsilon_s)$ $o.l \leftarrow o_{chk\langle x \rangle}$ sleep(T_{proc}) **end**

trajectory-planning algorithm) should be proprietary to the robot manufacturer and dependent on the specific robot model involved in the application. An implementation of these low-level algorithms has been provided to test the model-driven framework, though we do not claim it is the optimal one, since, for the reasons listed above, it is not the core of this research.

The resulting code pattern consists of:

- a **while** loop that runs as long as all expressions $s_j \in S(\Upsilon)$ that map the automaton's γ_j conditions, with $j \in [1, n]$, are false: in our specific

instance, these are s_{scs}, s_{fail}, and s_{stop}. Within the loop, the script: updates the state to $o_{\langle\mathsf{op}\rangle} \in \Sigma$; pauses for time $\mathsf{T}_{\mathsf{int}} \in K$ ($\mathtt{sleep}(\mathsf{T}_{\mathsf{int}})$); executes assignments $a_O \in A(\Upsilon)$; switches to $o_{chk_{\langle\mathsf{x}\rangle}} \in \Sigma$; and pauses for $\mathsf{T}_{\mathsf{proc}} \in K$ seconds ($\mathtt{sleep}(\mathsf{T}_{\mathsf{proc}})$).

Upon switching to checking location $o_{\mathrm{chk}_{\langle\mathsf{x}\rangle}}$, the automaton applies the orchestrator's set of policies, referred to as ξ_{O} ($\rightarrow a_{\mathrm{O}} \in A(\Upsilon)$), to the agents. After executing all the instructions in ξ_{O}, one of the following guard conditions might become true:

1. the condition that determines whether the mission has ended with success γ_{scs} ($\rightarrow s_{\mathsf{scs}} \in S(\Upsilon)$);
2. the condition that determines whether the mission has failed γ_{fail} ($\rightarrow s_{\mathsf{fail}} \in S(\Upsilon)$);
3. the condition that determines, for every controlled agent in the system, whether the current agent's action has to stop γ_{stop} ($\rightarrow s_{\mathsf{stop}} \in S(\Upsilon)$).

In the formal model, the time required by the orchestrator to make a decision based on the current system state is modeled by parameter $\mathsf{T}_{\mathsf{proc}} \in K \subset W$. When time $\mathsf{T}_{\mathsf{proc}}$ elapses ($\gamma_{\mathsf{proc}} = t_{\mathrm{act}} \geq \mathsf{T}_{\mathsf{proc}}$ holds) if one of the outgoing edges is enabled ($\gamma_{\mathsf{scs}} \vee \gamma_{\mathsf{stop}} \vee \gamma_{\mathsf{fail}}$ holds), the subcomponent $\langle\mathsf{op}\rangle$_chk is left, otherwise the orchestrator switches back to $o_{\langle\mathsf{op}\rangle}$.

The corresponding code block, shown in Table 10.5, captures the cyclical succession of $o_{\langle\mathsf{op}\rangle}$ and $o_{\mathrm{chk}_{\langle\mathsf{x}\rangle}}$. The orchestrator script is put on hold for $\mathsf{T}_{\mathsf{int}}$ seconds, while the system evolves, through the programmatic instruction sleep (the same happens afterward for $\mathsf{T}_{\mathsf{proc}}$ seconds in $o_{\mathrm{chk}_{\langle\mathsf{x}\rangle}}$). A set of policies equivalent to $\xi_{\mathrm{O}}(\rightarrow a_O \in A(\Upsilon))$ is enforced afterwards.

Variable $o.l \in \Upsilon_{\mathrm{s}} \subset \Upsilon$ that keeps track of the location is updated to $o_{\mathrm{chk}_{\langle\mathsf{x}\rangle}}$ and a second sleep instruction is issued to pause the execution for $\mathsf{T}_{\mathsf{proc}}$ seconds (thus, γ_{proc} is not part of the condition for the *while* cycle). Identically to the $\langle\mathsf{op}\rangle$_chk pattern, at this point the loop condition is re-evaluated and, if $\gamma_{\mathsf{scs}} \vee \gamma_{\mathsf{stop}} \vee \gamma_{\mathsf{fail}}$ holds, the loop ends along with the execution of this pattern.

As per Table 10.5, for the last pattern, the cycle is explicitly defined since it is part of an ordinary script external to the simulator or the real agent, whose execution flow requires explicit programming.

CHAPTER 11

Deployment Framework Validation

This chapter presents the results of an experimental campaign aimed at assessing the deployment framework presented in Chapter 10, specifically the model-to-code mapping's accuracy.

The validation is carried out on two sample simulated scenarios: one with a configuration leading to critical values of physical strain imposed on human subjects, while the second is configured to show higher criticality in terms of the robot's battery consumption.

A comparison of the simulation logs with runs of the formal model shows that errors are on average 5.35% for human fatigue and 0.13% for the robot's charge level.

11.1 Validation Goals

As discussed in Section 3.1.2, Statistical Model-Checking is a valuable tool to analyze cyber-physical systems' settings at an early design stage. However, it cannot provide all-inclusive results on its own [138].

The deployment module presented in Chapter 10 strengthens the model-driven framework by providing users with additional tools to test and val-

idate interactive scenarios. These features are enabled by the fact that the deployment framework elements are a rigorous transposition of the SHA network. Therefore, the approach ensures that the system will display corresponding behavior at design time and at deployment time.

In the following, we discuss the deployment module's validation process and present relevant case studies that serve the following two purposes:

P1: providing evidence that the formally modeled and the deployed agents behave correspondingly;

P2: showcasing the relevance of the deployment module to the development pipeline.

Through simulation, analysts can test applications in physically accurate environments. Moreover, they can focus their assessment on manually induced situations theoretically unlikely to occur but are critical for the mission's outcome.

To test the effectiveness of the approach in all its possible use cases, our experiments have focused on *charge-critical* or *fatigue-critical* configurations that are fitting sources of stress to the system.[1]

11.2 Experimental Setting

The chosen experimental setting is represented in Fig. 11.1 and replicates the layout used for validation purposes in Chapter 7. The scenario is set in a T-shaped hospital corridor with doors leading to offices and two cupboards with medical equipment. A mobile platform is deployed in this environment to assist patients and employees.

In this specific example, two humans are requesting the robot's assistance. The first person needs to fetch an item whose location is unknown to them but known to the robot. Therefore, the robot has to *lead* the human to their destination (**DEST 1** in Fig. 11.1), and the suitable interaction pattern is HumanFollower. The second human is a doctor who needs the robot to carry some tools. The doctor has to *lead* the robot to the tools' location (point **DEST 2** in Fig. 11.1), conforming to the HumanLeader interaction pattern.

Results obtained by applying our model-driven approach to this scenario significantly vary depending on parameter values. As already discussed, the two critical factors are the robot battery charge (C_0) and the fatigue profiles

[1]The content presented in this chapter also appears in [128]. The author of this thesis declares to have also authored the reproduced text, figures, and data and to have the right to reproduce such content in a dissertation according to the license under which both articles are published.

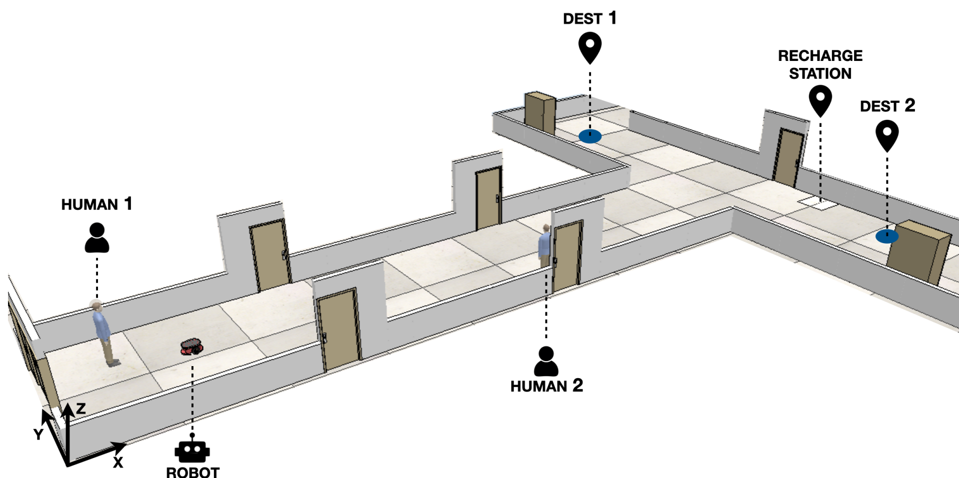


Figure 11.1: *Experimental setting used for both experiments, as seen in [128]. The layout from the simulator replicates the layout from the formal model. The two humans and the robot are represented in their starting positions. The picture also highlights the two destination points and the location of the recharge station.*

of the two humans ($\mathsf{p}_{\mathsf{f1}}, \mathsf{p}_{\mathsf{f2}}$), presented in Section 5.1. The MET is an indicator of how critical a fatigue profile is. The YoungHealthy fatigue profile corresponds—on average—to a MET of $5991.5s$ [221]. With an Elderly/Healthy profile, the MET equals $374.5s$, whereas with the Young/Sick profile the MET equals $299.6s$ (see Table 11.1).

We present two variations of the described scenario. The first experiment presents a *charge-critical* configuration and consists of two iterations of the approach (labeled as experiment $1\mathrm{a}$ and $1\mathrm{b}$). The second experiment starts with a *fatigue-critical* configuration. Table 11.1 summarizes the parameters for each experimental configuration.

As explained in Chapter 10, the listed design parameters and the low-level algorithms are specified for a generic mobile platform fit for testing purposes. However, they would need to be tuned (or come pre-packaged with the simulation model) based on the specific robot model to be deployed in a real environment.

11.3 Experimental Validation Process

The validation process we followed to assess if the deployment framework is an accurate translation of the formal model (i.e., goal **P1**) consists of the following steps (also conforming to the framework presented in Chapter 4):

Table 11.1: *Summary of experimental parameters set, as seen in [128]. Parameters that make the configuration critical are marked in red, whereas the ones the decrease the degree of criticality are in green.*

	Exp.1a	**Exp.1b**	**Exp.2**
$\mathsf{v}_{\mathsf{max}}$	$100cm/s$	$100cm/s$	$100cm/s$
$\mathsf{a}_{\mathsf{max}}$	$50cm/s^2$	$50cm/s^2$	$50cm/s^2$
C_0	30%	90%	50%
$\mathsf{T}_{\mathsf{poll}}$	$1s$	$1s$	$1s$
μ_λ	$0.5s$	$0.5s$	$0.5s$
σ_λ	$0.1s$	$0.1s$	$0.1s$
Pattern 1	Human-Follower	Human-Follower	Human-Follower
v_1	$100cm/s$	$100cm/s$	$80cm/s$
Ftg.Prof. 1	Young-Healthy	Young-Healthy	Elderly-Healthy
MET_1	$9210.34s$	$9210.34s$	$575.65s$
dest_1	$(2300.0, 500.0)$	$(2300.0, 500.0)$	$(2300.0, 500.0)$
Pattern 2	Human-Leader	Human-Leader	Human-Leader
v_2	$100cm/s$	$100cm/s$	$80cm/s$
Ftg.Prof. 2	Young-Healthy	Young-Healthy	Young-Sick
MET_2	$9210.34s$	$9210.34s$	$460.51s$
dest_2	$(2300.0, 100.0)$	$(2300.0, 100.0)$	$(2300.0, 100.0)$

1. automatically generate the formal model configured according to parameters in Table 11.1;
2. run the SMC experiment to estimate $\mathbb{P}_{\mathrm{M}}(\diamond_{\leq\tau}\ \mathsf{scs})$. The upper part of Table 11.2 displays the chosen time-bound values and performance data (duration of the experiment and number of states explored);
3. deploy the application in the simulated environment, with a real human user giving instructions to the human avatar in the scene;
4. collect the simulations' log files and compute metrics to compare the system's behavior at design time and at runtime and draw conclusions about the soundness of the deployment approach. The bottom part of Table 11.2 displays the resulting values of such metrics for each experiment, which will be analyzed in more detail later in this section.

Further remarks are necessary about the data shown in Table 11.2 and how we have calculated it. Firstly we empirically analyzed single traces for

Table 11.2: *Correspondence between the formal model (**FM**) and the simulated system behavior at deployment time (**SIM**), as seen in [128]. The upper part of the table reports performance data about the Statistical Model-Checking experiments and the number of runs for each experiment. In the bottom part, the above-mentioned metrics are listed. These metrics concern the average mission duration in simulation and the time-bound (τ) for the verification, the resulting probability of success within the specified time-bound, and the expected values for the most critical physical variables: the two humans' fatigue (F_{h1} and F_{h2}, respectively) and robot battery charge (C).*

	Experiment 1a		**Experiment 1b**		**Experiment 2**	
	FM	**SIM**	**FM**	**SIM**	**FM**	**SIM**
Runs	389	102	389	109	389	100
States	260961	–	520962	–	651867	–
Verification Time [min]	1.05	–	2.38	–	2.63	–
Avg. Completion Time [s]	–	65.32	–	66.10	–	70.95
Time-bound (τ) [s]	70	–	70	–	80	–
$\mathbb{P}_{\mathrm{M}}(\diamond_{\leq\tau}\ \mathsf{scs})$ [%]	≥ 90.186	96.1	≥ 90.186	94.5	≥ 90.186	100
$\mathrm{E}_{\leq\tau}[\max(F_{h1})]$ [%]	1.37 ± 0.3	1.36	1.39 ± 0.2	1.43	25.54 ± 4.9	25.57
$\mathrm{E}_{\leq\tau}[\max(F_{h2})]$ [%]	0.92 ± 0.3	0.96	0.93 ± 0.4	1.06	23.83 ± 3.7	21.43
$\mathrm{E}_{\leq\tau}[\min(C)]$ [%]	24.82	24.82	89.36	89.26	54.25	54.10

all experiments to estimate the average duration of the mission and choose the time-bound τ accordingly. Hence, we reasonably rule out the possibility of obtaining a low probability of success due to insufficient time-bound.

In the second place, it is necessary to explain why the number of runs differs between design time and runtime. Uppaal generates as many runs of the SHA as necessary to compute a confidence interval with the required confidence level. In this case, this amounts to 389 runs of the system for all experiments. For the deployment phase, as it does not involve the same statistical techniques, we produce a number of meaningful simulation runs for practical reasons. If about 40 people are served by the robot every day for 5 workdays a week and 2 humans are served in each run, simulating the mission about 100 times realistically corresponds to testing the deployment framework over the span of a week.

To fulfill goal **P2**, the following step is performed:

5. while simulating the application, force situations that can lead to failure but are not covered by the formal model. Some examples and possible countermeasures for each experiment are presented in the corresponding subsections.

The deployment framework used for the experiments is available at [132].

The formal model is created with Uppaal v.4.1.24 [46].[2] The virtual environment is created using CoppeliaSim v.3.6.2. The deployment units' scripts are implemented in Python v.3.6.9 and the LUA scripting language.[3] Finally, the middleware layer is built using the ROS Melodic distribution v.1.14.7.

11.4 Experiment 1: Charge-Critical Configuration

Experiment $1a$ involves two young, healthy humans and a robot with a low charge value ($\mathsf{C}_0 = 30\%$). With this configuration, when the mission starts, the robot has approximately $350s$ of battery life. If both humans start walking as soon as it is their turn to be served and perform flawlessly, $350s$ are sufficient to complete the mission.

As Table 11.2 shows, the average mission completion time observed in the simulations is $65.32s$. This result is compatible with the time-bound empirically chosen for the formal verification ($70s$), which leads to a success probability interval—estimated with SMC—of $[0.90186, 1]$. This success rate is confirmed while deploying the application since processing the collected log files reveals that in 96.1% of the simulations, the mission was indeed successfully completed within $70s$.

Note that, with a slight abuse of notation, both in Table 11.2 and below, the success rate of the deployed application is still indicated as $\mathbb{P}_{\mathrm{M}}(\diamond_{\leq\tau}\ \mathsf{scs})$, as in the formal model; however, it is not calculated through SMC, but as the ratio between how many runs feature the mission successfully ending within τ and the total number of runs, as per Eq.11.1.

$$\mathbb{P}_{\mathrm{M}}(\diamond_{\leq\tau}\ \mathsf{scs}) = \frac{\#\ \text{successful runs within}\ \tau}{\#\ \text{runs}} \tag{11.1}$$

Besides the success rate, it is crucial also to verify that physical variables are accurately simulated. To this end, we have calculated through Uppaal the expected value for the maximum value of fatigue reached by the two humans and the residual battery charge at the end of the mission (thus, the minimum value since, in these scenarios, the robot never recharges). The same three metrics are extracted from the simulation logs.

By comparing these indicators, we can conclude that both fatigue and charge evolutions in time are consistent with the results obtained with the formal model. In both phases, the two humans reach a negligible value of

[2] SMC experiments are performed on a Linux machine with 128 cores, 515GB of RAM, and Debian Linux version 10.

[3] Simulations are executed on a Ubuntu 18.04 virtual machine with two cores and 4GB of RAM.

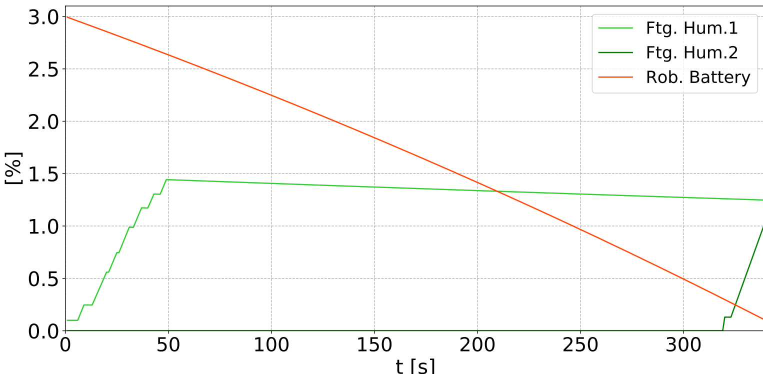


Figure 11.2: *Plot of the unsuccessful simulation run from Experiment 1a, as seen in [128]. Green lines correspond to human fatigue ($[0-100]$), while the orange line corresponds to the battery charge ($[0-10]$). Human 2 starts walking at $t \simeq 320s$, about $270s$ after the first human has been served, causing mission failure ($Q \simeq 0$).*

fatigue ($\sim 1\%$) as they both conform to the "best" fatigue profile and only walk for about $30s$ each. In contrast, the residual level of charge approximately equals 24%.

After an evaluation of the system's behavior in "regular" circumstances, the analyst can forcedly induce the situation in which the second human does not start as soon as it is their turn, and the whole execution of the mission is delayed (see Fig. 11.2). The cause of the delay in a real setting could be the doctor being unexpectedly held up or failing to react immediately, which is a common human mistake [10]. The consequently extended duration of the mission leads to the robot being fully discharged before its completion,[4] which is one of the two possible causes of *failure*.

The analyst is now able to assess the unsuccessful simulation run. They might decide that this manifestation of human free will is plausible in real life and critical enough to motivate a scenario refinement and an additional iteration of the design-time analysis. Since humans cannot be programmatically instructed to perform actions in a machine-like manner (neither in the formal model nor in real life), the most sensible refinement action would be selecting a higher value of C_0. If multiple mobile platforms are available, this can be realized by choosing a different robot from the fleet or recharging the robot before starting the mission.

[4]Although the robot can recharge itself both in the formal model and during simulation, it is effectively busy serving a *leader* human when it reaches the recharge threshold ($C = 20\%$). Under these circumstances, the robot cannot halt service delivery even if the human is delaying their action.

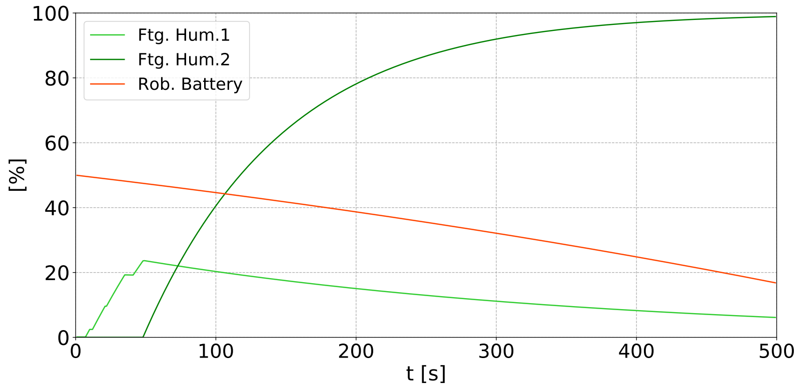


Figure 11.3: *Plot of the unsuccessful simulation run for Experiment 2, as seen in [128]. Green lines correspond to human fatigue ($[0-100]$), while the orange line corresponds to the battery charge ($[0-100]$). The plot shows that the first human is successfully served in about $50s$ as expected, whereas the second one keeps moving until they reach full exhaustion leading to mission failure ($F_{h2} \simeq 100\%$).*

The new configuration is shown in Table 11.1 as Experiment 1b, with the updated value of $\mathsf{C}_0 = 90\%$. The second iteration of SMC experiments, with $\tau = 70s$, yields success rate $[0.90186, 1]$ (see Table 11.2).

Re-running the batch of simulations leads to similar results as for Experiment 1a: the average completion time is approximately $66s$ and the success rate within $70s$ is 94.5%.

Note that metrics concerning fatigue are almost unchanged with respect to Experiment 1a and are still comparable to the ones estimated with the formal model. As expected, the residual battery charge is higher than for Experiment 1a (about 89%) coherently with the different value of C_0.

11.5 Experiment 2: Fatigue-Critical Configuration

The setup for the second experiment is more critical in terms of human endurance to physical strain. Specifically, criticality arises from the different fatigue profiles Elderly/Healthy and Young/Sick, as in Table 11.1. In practice, the second subject could be an employee with an undiagnosed condition affecting their respiratory capacity and physical endurance.

For this experiment, we have calculated the same metrics described in Section 11.4, whose resulting values are also shown in Table 11.2. In this case, the chosen value for τ is $80s$, which is slightly higher than for Experiment 1 since humans walk at a slower pace due to critical health conditions.

The SMC experiment yields a success probability range of $[0.90186, 1]$. Therefore, as in the first experiment, the analyst would have sufficient evidence to consider the mission fit for deployment.

Also, in this case, the results obtained at design time are corroborated by the application's deployment. The average mission completion time is $70.95s$, which, as expected, is slightly higher than for the first experiment and compliant with the chosen time-bound. More specifically, all the performed simulations were completed within $\tau = 80s$ ($\mathbb{P}_{\mathrm{M}}(\diamond_{\leq\tau}\ \mathsf{scs}) = 100\%$).

The expected values for the fatigue peaks are significantly higher than in the previous experiment ($\sim 20\% \gg \sim 1\%$) due to the different fatigue profiles. However, they still comply with the values calculated through Uppaal at design time. The same stands for the robot battery charge. Therefore, also in this second experiment, the deployed system's behavior shows evidence of accurate transposition of the formal model.

Nevertheless, suppose now that the second subject- with a more critical fatigue profile- follows an erratic trajectory, not the shortest one. This situation can be simulated in the virtual environment. Fig. 11.3 shows a specific simulation run in which the second human exhibits this behavior and keeps walking until complete exhaustion. The first human reaches their destination in approximately $50s$ (as in Fig. 11.3, they stop walking and start resting). The second human starts walking at around $t = 20s$ and keeps moving for about $450s$. As in Table 11.1, a subject with the Young/Sick fatigue profile can walk non-stop at most for approximately $7.5min$ ($\mathsf{MET}_2 = 460.51s$). Therefore, the mission fails after about $500s$ because the second human reaches the maximum endurable value of fatigue.

This manifestation of human autonomy is not covered by the formal model in its current development stage. SMC experiments do not account for this possibility and yield a very high value for the chances of success. Indeed, as explained in Chapter 10, the formal model accounts for human autonomy only to some extent. Precisely, it currently does not capture the possibility of the human freely straying from the planned trajectory, as in the simulation in Fig. 11.3.

11.6 Discussion

Firstly, the experiments in Section 11.4 and Section 11.5 provide evidence that the formally modeled and the virtually simulated agents behave correspondingly. Comparing the average completion times and success rates

allows us to conclude that the deployed robot controller issues the same commands as the corresponding automaton and with the same timing, leading to mission success in a comparable amount of time.

As for human fatigue and battery charge metrics, it is not surprising that fatigue is—apparently—less accurately simulated than charge: the average formal model-to-deployment error for fatigue is 5.35% while it is only 0.13% for the battery charge. This discrepancy is due to the higher degree of variability of fatigue ascribable to the unpredictability of human behavior. On the other hand, battery charge evolution in time is less subject to uncertainty (mostly unexpected trajectory variations due to obstacles), thus leading to a smaller error.

Furthermore, assessing these experimental results makes it possible to identify two types of model-to-reality discrepancies that can be detected through deployment.

In the first case, as in the first experiment, even if the deployment highlights a failure due to a gap in the formal model, it is still feasible to counteract it by tuning the parameters of the scenario, e.g., the initial battery charge $\mathsf{C}_{\mathsf{start}}$. In the second case, of which the second experiment is an example, the failure caused by the gap in the formal model can only be tackled at design time by refining the formal model itself.

The fundamental difference is that the first case can be directly handled by the analyst, whereas the second requires expertise in formal modeling that they are unlikely to possess. The following phase of the development of the approach, i.e., the data-driven refinement of the SHA network, makes it possible to counteract the second category of failures through multiple iterations of the design-time analysis.

Part III

Model Adjustment

CHAPTER 12

$\mathsf{L}^*_{\mathrm{SHA}}$ for Stochastic Hybrid Automata Learning

This chapter focuses on the automata learning phase of the framework.

Firstly, given the domain-agnostic nature of the procedure, part of the framework is generalized to a software development toolchain of complex Cyber-Physical Systems (CPSs) incorporating the automata learning task.

The $\mathsf{L}^*_{\mathrm{SHA}}$ algorithm developed within this research project and extending L^* to SHA is then introduced in detail.

12.1 System Under Learning Configuration

The application framework of $\mathsf{L}^*_{\mathrm{SHA}}$ targets CPSs whose physical component includes entities with predictable temporal behavior (e.g., programmable machines) and entities whose behavior is uncertain. The latter is the case of entities whose behavior is the combination of physical processes overly complex to be manually drafted and subject to uncertainties (e.g., human decision-making) [32]. Specifically, $\mathsf{L}^*_{\mathrm{SHA}}$, which yields SHA modeling the uncertain entities, is tailored to:

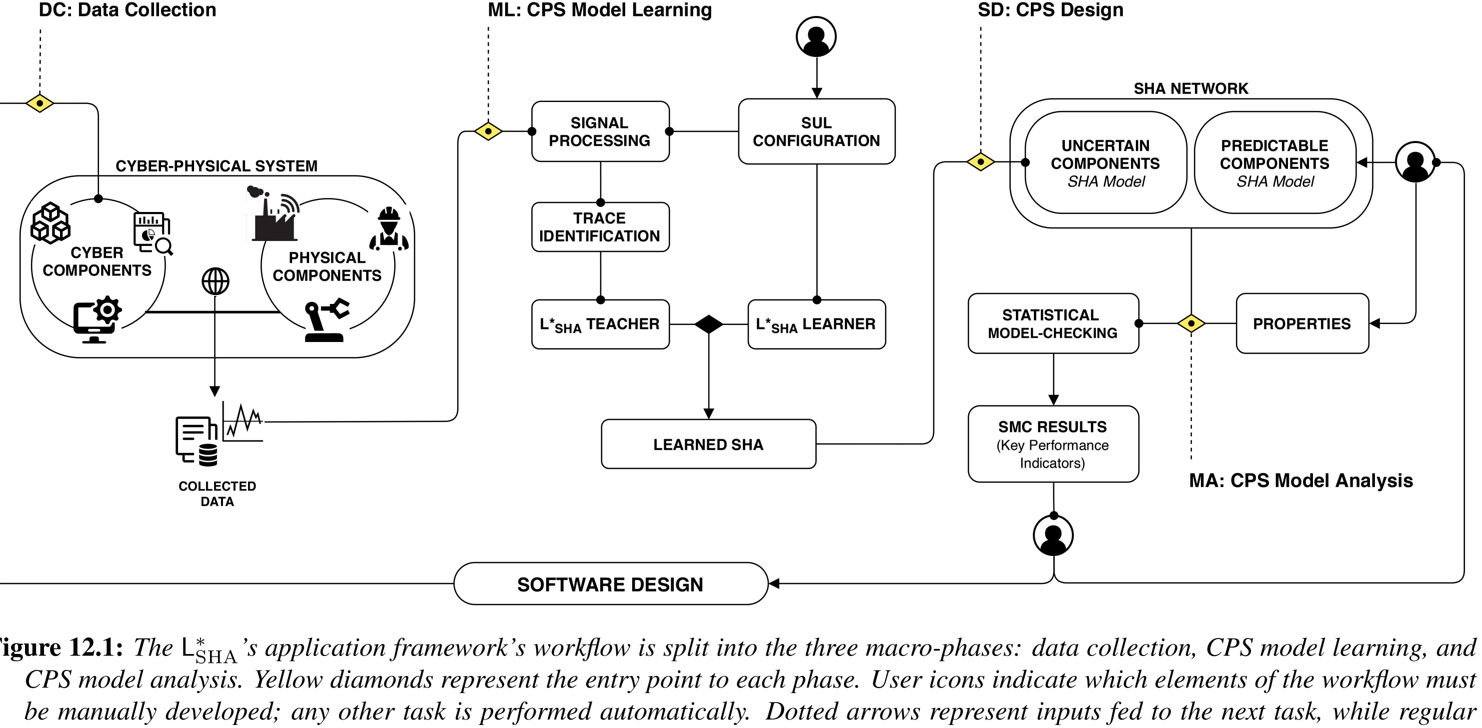


Figure 12.1: *The* $\mathsf{L}^*_{\mathrm{SHA}}$*'s application framework's workflow is split into the three macro-phases: data collection, CPS model learning, and CPS model analysis. Yellow diamonds represent the entry point to each phase. User icons indicate which elements of the workflow must be manually developed; any other task is performed automatically. Dotted arrows represent inputs fed to the next task, while regular arrows represent the generation of output.*

1. sources of uncertainty whose non-determinism is refined through stochastic processes;
2. entities with a finite number of feasible behaviors.

The model-driven software design framework is made up of four phases (labeled accordingly in Fig. 12.1):

DC: during the data collection phase, observations of the system's behavior are gathered in preparation to the learning;

ML: the CPS model learning phase exploits collected observations to learn a SHA of uncertain CPS components;

SD: the CPS design phase combines the learned formal model of uncertain entities with the model of deterministic components provided by the CPS designer;

MA: the CPS model analysis phase applies statistical techniques to the developed model to assess the CPS's performance: results support the developer during software design.

Collecting data is necessary to accumulate knowledge of the system's behavior in response to specific events. Physical entities, such as machinery or involved human subjects, need to be endowed with suitable sensors, while cyber entities log relevant information. $\mathsf{L}^*_{\mathrm{SHA}}$ infers from available data a model of the uncertain/predictable components capturing the evolution of physical processes. To this end, the learning procedure can only consider *measurable* physical variables and *observable* events.

Every physical variable whose dynamics are captured by the learned model must be measured through a suitable sensor: for instance, it is impossible to know how a room's temperature or human fatigue evolves in response to specific events if no sensor readings are available.

Similarly, for the learning to be feasible, the occurrence of a specific event (e.g., a window opening or a human starting to walk) must be deducible from available data: it is not possible to learn how the system responds to a specific event if the occurrence of such event is never recorded. Depending on the specific event, its occurrence might have to be inferred from data (e.g., a human starting to walk may be inferred from a change in their position) or explicitly logged (e.g., the robot's controller instructing the robot to stop moving may log this information directly). The concepts of measurability and observability are detailed in Section 12.2.1.

The outcome of **DC** is the set of collected data, which are likely heterogeneous (i.e., originating from different sources) and whose nature may vary significantly between different CPSs instances.

The cornerstone of the model learning phase (**ML**) is $\mathsf{L}^*_{\mathrm{SHA}}$. Similarly to L^*, in $\mathsf{L}^*_{\mathrm{SHA}}$, learning occurs through the interaction between a teacher and a learner. The teacher stores the accumulated knowledge about the CPS's uncertain component—referred to as the System Under Learning (SUL)—while the learner maintains the *hypothesis* SHA. The learner progressively refines the hypothesis SHA by querying the teacher about the system's behavior in response to specific event sequences.

While $\mathsf{L}^*_{\mathrm{SHA}}$ is agnostic with respect to the specific SUL, some of its features require manual configuration (indicated as "**SUL Configuration**" in Fig. 12.1). It is necessary to specify which events can occur in the system. Note that observable events that do not impact the behavior of the component under learning should not be specified. Each event is identified by a label and a condition (expressed as a set of logic formulae) that, if evaluated against data, determines whether the event occurred. Collected data then requires pre-processing to become accessible by the $\mathsf{L}^*_{\mathrm{SHA}}$ teacher (as per Fig. 12.1). Specifically, all sensor logs need to be parsed into sampled signals through signal processing techniques.

$\mathsf{L}^*_{\mathrm{SHA}}$ infers the SUL's behavior in response to specific sequences of events, referred to as "*traces*". Therefore, for each available batch of signals, it is necessary to *identify* the events (assumed to be observable) that led to the specific recorded behavior. This task is referred to as trace identification. For example, a variation in the room temperature's derivative (i.e., the *recorded* behavior) indicates that the heating system has been switched on (i.e., the *identified* event). How to parse signals and identify events must also be specified upon configuring the SUL. A set of *candidate* models underlying continuous-time physical processes (i.e., systems of ODEs) governing the SUL must also be specified beforehand. For each signal, $\mathsf{L}^*_{\mathrm{SHA}}$ identifies the equation that best describes it out of the specified candidates.

When $\mathsf{L}^*_{\mathrm{SHA}}$ terminates, it returns the learned SHA modeling the configured uncertain entity. Upon entering the CPS design (**SD**) phase, the learned SHA is plugged into a larger network of SHA representing the entire CPS, where the model of predictable components (either physical such as programmable robotic platforms, or cyber such as robot controllers) is known a priori and provided by the CPS designer rather than learned.

Finally, the purpose of CPS analysis (phase **MA**) is to formally verify whether the CPS model meets specific requirements. Requirements and associated performance indicators of the CPS are expressed as prop-

erties within the verification tool. Given the stochastic nature of the formal model, the chosen formal verification technique is SMC rather than exhaustive model-checking, which is not feasible. SMC results quantify either the probability that a certain property holds or the expected value of relevant indicators. These results provide developers with insights into the system's behavior, such as safety hazards and sources of inefficiency, to be factored in when developing (or re-configuring) cyber components that control and monitor the CPS.

12.2 $\mathsf{L}^*_{\mathrm{SHA}}$ Algorithm

This section introduces the $\mathsf{L}^*_{\mathrm{SHA}}$ algorithm for SHA learning. The core of $\mathsf{L}^*_{\mathrm{SHA}}$ is the interaction between the learner, which iteratively builds the hypothesis SHA, and the teacher, which possesses knowledge about the SUL and can thus answer the learner's queries. The algorithm relies on collected sample traces to gain knowledge about the SUL as, given the application domain (i.e., CPSs with a highly variable and unpredictable behavior), the existence of a teacher with *exact* knowledge is possible in theory but holds limited significance in practice. The learning procedure is active since the teacher can request new system traces if it finds that knowledge of a specific contingency (i.e., an event sequence) is insufficient: practical implications of this feature are discussed later in this section.

Fig. 12.2 shows a high-level overview of the algorithm's workflow: the fundamental infrastructure (i.e., the tasks and how they are split between learner and teacher) directly takes after L^*. With each round of learning, the learner progressively refines the SHA conjecture $\mathcal{A}_{\mathrm{hyp}}$ based on currently accumulated knowledge about the SUL. To check whether the current conjecture automaton is up-to-date with the available pool of traces, during each round, the learner asks the teacher for a counterexample, shortened as *c.ex.* in Fig. 12.2. A counterexample is a sequence of events that has already been observed, hence it is part of the teacher's knowledge (similarly to L^*) but not compliant with the current SHA conjecture. If no counterexample exists, the conjecture is up-to-date with the current knowledge, otherwise, a counterexample is identified.

At the start of each round, as per Fig. 12.2, the teacher performs a ref query to examine recorded sequences of events (and all their prefixes) that the hypothesis automaton must capture. If the pool of data for one (or multiple) event sequences is insufficiently informative to perform the learning, the teacher asks the SUL for new runs. For each $\mathcal{A}_{\mathrm{hyp}}$ location, the learner then modifies the $\mathcal{A}_{\mathrm{hyp}}$ graph to account for the latest counterexample, and

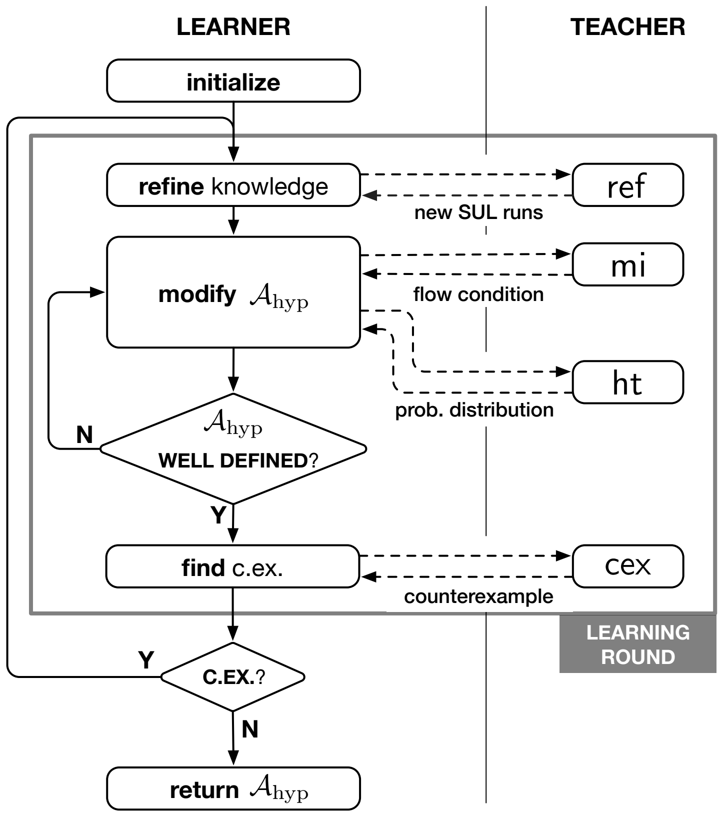


Figure 12.2: *High-level representation of $\mathsf{L}^*_{\mathrm{SHA}}$ workflow. The left lane contains the operations performed by the learner, the right lane shows the queries performed by the teacher. Rectangles represent instructions, whereas diamonds are decision points with two outgoing branches labeled as yes/no ($\mathbf{Y}/\mathbf{N}$). Solid arrows represent tasks executed sequentially. Dashed arrows represent function invocations, labeled to indicate what the invoked function returns.*

it asks the teacher to determine flow conditions and probability distributions (by performing mi and ht queries depicted in Fig. 12.2). At this point, the learner checks whether $\mathcal{A}_{\mathrm{hyp}}$ is well defined or not, that is, whether all locations are properly defined, and transitions are deterministic (the specific properties are formalized in Section 12.2.2). In the latter case, the learner modifies the conjecture graph and resubmits mi, and ht queries. When $\mathcal{A}_{\mathrm{hyp}}$ is well defined, the learner asks for a new counterexample through a cex query. If a counterexample exists, a new round of learning is necessary. If the teacher does not find any counterexample, $\mathsf{L}^*_{\mathrm{SHA}}$ terminates and returns $\mathcal{A}_{\mathrm{hyp}}$ as the automaton modeling the SUL given the collected set of traces.

The notation introduced in Chapter 3 and the pseudo-code presented in the following apply to systems with *one* physical variable whose flow conditions characterize each location. Nevertheless, the algorithm is general and can be extended to SULs with multiple physical variables whose behavior is not fully predictable and can thus be learned through $\mathsf{L}^*_{\mathrm{SHA}}$ based on collected traces.

12.2.1 Signals and Traces

Physical variables of the SUL captured by the SHA must be measurable through sensors and have their values recorded for each run. Each log of a variable's recording is hereinafter referred to as "*sampled signal*".

Definition 2. *A sampled signal is a finite sequence of pairs* $(\tau_0, v_0)(\tau_1, v_1) \ldots (\tau_{n-1}, v_{n-1})$*, such that the following properties hold:*

1. $\forall i \in [0, n-1] : \tau_i \in \mathbb{R}_+$;
2. $\forall i \in [0, n-2] : \tau_i < \tau_{i+1}$;
3. $\forall i \in [0, n-1] : v_i \in \mathbb{R}^{V_{\mathrm{dc}}}$.

The set of all sampled signals is indicated as Σ*. Value* v_i *of a sampled signal* $\sigma \in \Sigma$ *is also indicated as* σ_i*. The length of a sampled signal* $\sigma \in \Sigma$ *is the number of pairs in the sequence and is indicated as* $|\sigma|$*.*

Definition 3. *Given a sampled signal* $\sigma \in \Sigma$ *of length* n*, its change-point vector, denoted by* π_σ*, is a finite ordered sequence of timestamps* $\tau'_0 \ldots \tau'_{m-1}$ *of length* $m < n$ *such that the following properties hold:*

1. $\forall j \in [0, m-1], \exists i \in [1, n-1] : \ \tau'_j = \tau_i$;
2. $\forall j \in [0, m-1], i \in [1, n-1] : \tau_i = \tau'_j \Rightarrow v_i \neq v_{i-1}$.

Every element of π_σ *is called* change-point.

$\mathsf{L}^*_{\mathrm{SHA}}$ is applicable to systems where any change of behavior of the SUL—corresponding to a location switch in the SHA—is triggered by an observable event. An event with timestamp τ_e is observable if it is associated with a change point. The change-point vector is necessarily smaller than the sampled signal it derives from ($m < n$ holds) since the first timestamp τ_0 can never be a change-point. We assume that sampled signals feature measurable physical variables *and* their derivatives (e.g., position and speed). A change-point vector contains change points related to all variables, though not all of them correspond to significant events. For example,

in the case of human-robot interaction, human fatigue changes value at all times. Still, only change-points of its derivative point out an event (e.g., the human starting to walk).

In the SHA, an *event* is a pair (γ, c) of a guard condition $\gamma \in \Gamma(W)$ and a channel $\mathsf{c} \in C$ that causes an edge to fire. We indicate with A the set of possible events, where $A \subseteq \Gamma(\mathcal{W}) \times C$ is a fixed and finite alphabet. $\mathsf{L}^*_{\mathrm{SHA}}$ relies on a system-specific "*labeling function*" $\mathcal{L}_{\pi_\sigma} : \mathbb{R}_+ \rightarrow A \cup \{\bot\}$ (implementing the **trace identification** task in Fig. 12.1). Given a change-point vector π_σ, $\mathcal{L}_{\pi_\sigma}$ assigns a pair (γ, c) to a change-point τ_j in π_σ if τ_j corresponds to an event, $\bot$ otherwise. The function $\mathcal{L}_{\pi_\sigma}$ identifies at most one event for each timestamp since an event corresponds to firing an edge, and a single SHA (i.e., the one being learned) does not allow multiple edges to fire at the same time.

Definition 4. *Given a sampled signal $\sigma \in \Sigma$, its corresponding* timed trace *ttr_σ is a finite sequence of pairs $(\tau_0, (\gamma_0, \mathsf{c}_0)) \dots (\tau_{p-1}, (\gamma_{p-1}, \mathsf{c}_{p-1}))$ of length p, where $p = |\pi_\sigma|$ holds, such that the following properties hold:*

1. $\forall k \in [0, p-1], \exists \tau_j \in \pi_\sigma : \mathcal{L}_{\pi_\sigma}(\tau_j) = (\gamma_k, \mathsf{c}_k) \wedge \tau_k = \tau_j$;
2. $\forall \tau_j \in \pi_\sigma, \exists k \in [0, p-1] : (\gamma_k, \mathsf{c}_k) = \mathcal{L}_{\pi_\sigma}(\tau_j) \wedge \tau_k = \tau_j$;
3. $\forall k_1, k_2 \in [0, p-1] : \tau_{k_1} < \tau_{k_2} \iff k_1 < k_2$.

The set of all timed traces derived from signals in Σ is indicated as $\mathcal{TTR}$. A timed trace without timestamps constitutes a "*trace*" $tr \in \mathcal{TR}$ and is obtained from the original timed trace ttr through operation $\mathrm{untime}(ttr)$. We use the term "*string*" to indicate a trace or a trace prefix; notation $tr' \ll tr$ (resp., $tr' \gg tr$) is used to state that string tr' is a prefix (resp., a suffix) of tr. A string of which there exists a record in the collected pool of SUL runs is referred to as "*observed*". Since signals and traces are part of the teacher's knowledge, the set of observed sampled signals is hereinafter indicated as $\textsc{Teacher}.\Sigma_{\mathrm{obs}}$, the set of timed traces and untimed traces derived from signals in Σ_{obs} as $\textsc{Teacher}.\mathcal{TTR}_{\mathrm{obs}}$ and $\textsc{Teacher}.\mathcal{TR}_{\mathrm{obs}}$, respectively.

12.2.2 Observation Tables

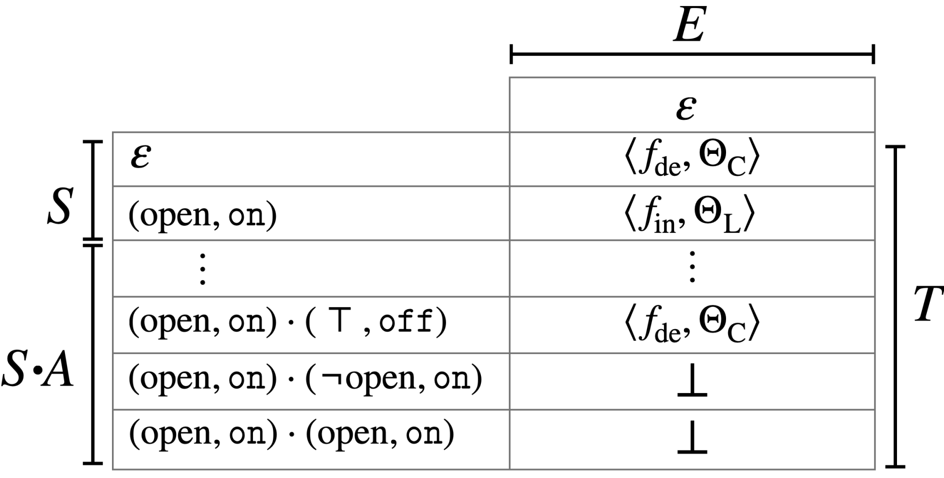
The $\mathsf{L}^*_{\mathrm{SHA}}$ learner interacts with a teacher capable of answering a set of queries and returns the learned SHA. During each learning round, the learner refines $\mathcal{A}_{\mathrm{hyp}}$, representing the most recent SHA hypothesis. The learner loop stops when, based on the currently accumulated knowledge about the SUL, $\mathcal{A}_{\mathrm{hyp}}$ can no longer be refined; thus, no further round of learning is

needed. The version of $\mathcal{A}_{\mathrm{hyp}}$ resulting from the last learning loop is the final learned automaton (returned by $\mathsf{L}^*_{\mathrm{SHA}}$).

The $\mathsf{L}^*_{\mathrm{SHA}}$ learner, as in the original L^* algorithm, stores information in an "*observation table*" [7]. Rows and columns of the table are labeled with traces (ϵ is the symbol for "no event"). Let $\cdot$ be the string concatenation operator. A table cell $T(r \cdot c)$ for a trace $tr = r \cdot c$, where r and c are the corresponding row and column labels, describes the behavior (i.e., flow condition and probability distribution pair) of $\mathcal{A}_{\mathrm{hyp}}$ after the occurrence of events in tr from the initial state. With a slight misuse of notation, we use $r \cdot c$ to indicate the location $l \in L$ where the automaton lands if the events in string $r \cdot c$ occur. This is possible because, as explained later in the section, all edges of the hypothesis SHA built by $\mathsf{L}^*_{\mathrm{SHA}}$ have a deterministic destination. A table cell is undefined (indicated as $\perp$) if it is yet to be analyzed or the pool of data corresponding to the associated trace is insufficiently informative.

The table can be seen as split into two sections (also highlighted in Fig. 12.3): the upper one contains rows labeled by strings in a set $S \subseteq \mathcal{TR}$, whereas the bottom one contains rows labeled by strings in S extended by all possible events in A, thus identified by set $S \cdot A$ (operator $\cdot$ also applies to sets of strings). The two sets of strings are referred to as *short traces* and *long traces*, respectively. As in L^*, each *unique* row identified by a short trace represents a candidate SHA location.

Definition 5. *An* observation table *is a tuple* $\langle S,\ E,\ \mathcal{Z},\ T\rangle$*, where:*



	ε
ε	$\langle f_{\mathrm{de}}, \Theta_{\mathrm{C}}\rangle$
(open, on)	$\langle f_{\mathrm{in}}, \Theta_{\mathrm{L}}\rangle$
⋮	⋮
(open, on) $\cdot$ ($\top$, off)	$\langle f_{\mathrm{de}}, \Theta_{\mathrm{C}}\rangle$
(open, on) $\cdot$ ($\neg$open, on)	$\perp$
(open, on) $\cdot$ (open, on)	$\perp$

Figure 12.3: *Portion of observation table of the SHA in Fig. 3.1a. Rows, columns, and cells are marked to visualize tuple $\langle S,\ E,\ \mathcal{Z},\ T\rangle$ and long traces in set $S \cdot A$.*

1. *$S \subseteq \mathcal{TR}$ is a set of strings such that, given $tr \in S$, all tr' such that $tr' \ll tr$ holds also belong to S (i.e., S is* "prefix-closed"*);*

2. *$E \subseteq \mathcal{TR}$ is a set of strings such that, given $tr \in E$, all tr' such that $tr' \gg tr$ holds also belong to E (i.e., E is* "suffix-closed"*);*

3. *$\mathcal{Z} \subseteq \wp(\mathbb{R})$ are the sample sets from which empirical distributions for the stochastic parameter derive;*

4. *T: $((S \cup S \cdot A) \cdot E) \to (M' \times \mathcal{Z}) \cup \{\bot\}$ is such that $T(tr) \neq \bot$ holds if, and only if, tr corresponds to an observed string of the system, where $M' \subseteq \{\mathbb{R}_+ \cup (\mathbb{R}_+ \times \mathbb{R}) \to \mathbb{R}^W\}$ is a given set of flow conditions.*

We define function $row : (S \cup S \cdot A) \to (E \to M' \times \mathcal{Z})$ such that $row(r)(c) = T(r \cdot c)$ holds.

With reference to Fig. 12.3, displaying part of an observation table, $row((\text{open}, \texttt{on}))(\epsilon)$ equals $T((\text{open}, \texttt{on}) \cdot \epsilon)$.

As mentioned above, both in L^* and $\mathsf{L}^*_{\text{SHA}}$, function T can be represented as a table. In L^*, function T yields a Boolean value indicating whether string $r \cdot c$ is *accepted* by the DFA under learning or not. In $\mathsf{L}^*_{\text{SHA}}$, the output of function T captures how physical variables evolve if the sequence of events identified by string $r \cdot c$ occurs from the initial state. In $\mathsf{L}^*_{\text{SHA}}$, $T(r \cdot c)$ yields a $\langle \mathcal{F}(r \cdot c), \mathcal{D}(r \cdot c) \rangle$ pair composed by a flow condition $\mathcal{F}(r \cdot c) \in M'$ and an empirical distribution $\mathcal{D}(r \cdot c) \in \mathcal{Z}$ ($\mathcal{F}$ and $\mathcal{D}$ are introduced in Definition 1). Since the observation table represents conjecture automaton $\mathcal{A}_{\text{hyp}}$, filling its cells is necessary to infer from available data the nature of learned locations. Filling observation table $\langle S, E, \mathcal{Z}, T \rangle$ amounts to the learner submitting queries to the teacher for each row r and column c such that the corresponding cell is undefined (i.e., $T(r \cdot c) = \bot$ holds). $\mathcal{F}(r \cdot c)$ and $\mathcal{D}(r \cdot c)$ are provided by the teacher as answers to mi and ht queries, respectively. Algorithm 1 shows the full function to fill an observation table $\langle S, E, \mathcal{Z}, T \rangle$. We remark that $T(r \cdot c) = \bot$ holds if *at least one* among $\mathcal{F}(r \cdot c)$ and $\mathcal{D}(r \cdot c)$ is undefined. Note that $T(r \cdot c)$ may still be undefined after submitting the queries since the teacher may not have enough observations of the requested trace to answer queries conclusively.

For instance, if $\mathsf{L}^*_{\text{SHA}}$ is fed with signals originating from the SUL modeled in Fig. 3.1a, it would produce the observation table partially shown in Fig. 12.3. The initial location (i.e., the row corresponding to the empty trace ϵ) is characterized by flow condition f_{de} (see Section 3.1.1) and sample set $\Theta_{\text{C}} \in \mathcal{Z}$, which should be drawn from $\mathcal{N}(\mu_{\text{C}}, \sigma^2_{\text{C}})$. In the case

Algorithm 1 Learner function to fill empty cells in $\langle S, E, \mathscr{Z}, T\rangle$ with TEACHER replies to mi and ht queries.

```
1: procedure fill(⟨S, E, 𝒵, T⟩)
2:     for r ∈ S ∪ S · A do
3:         for c ∈ E do
4:             tr ← r · c
5:             if T(tr)= ⊥ then
6:                 T(tr) ← ⟨TEACHER.mi(tr), TEACHER.ht(tr)⟩
```

that event $(\text{open}, \texttt{on})$ occurs (thus, an event fires through channel on with guard open enabled), the table identifies a different location with flow condition f_{in} and samples $\Theta_{\mathrm{L}} \in \mathscr{Z}$ (drawn from distribution $\mathcal{N}(\mu_{\mathrm{L}}, \sigma^2_{\mathrm{L}})$). In the portion shown in Fig. 12.3, traces ϵ and $(\text{open}, \texttt{on})$ are elements of set S (i.e., the short traces). Long traces (i.e., set $S \cdot A$) result from the concatenation of short traces with any event observable in the SUL: Fig. 12.3 shows the long traces derived from $(\text{open}, \texttt{on})$. If the thermostat has been switched on (hence, $(\text{open}, \texttt{on})$ fires) and then it is switched off (i.e., $(\top, \texttt{off})$ fires, whose guard condition is always enabled), the table captures the system switching to the initial location (i.e., $T(\epsilon \cdot \epsilon)$ equals $T((\text{open}, \texttt{on}) \cdot (\top, \texttt{off}) \cdot \epsilon))$). On the other hand, since in the SUL it is not feasible for the thermostat to switch on two times consecutively, no runs are found featuring long traces $(\text{open}, \texttt{on}) \cdot (\neg\text{open}, \texttt{on})$ and $(\text{open}, \texttt{on}) \cdot (\text{open}, \texttt{on})$. Therefore, the corresponding table cells are labeled with $\bot$.

The following presents the algorithms computing the answers to mi and ht queries.

Model Identification Query mi(tr)

Algorithm 3 reports the implementation in pseudo-code of a mi query for a generic string tr. This query aims to identify the flow condition that most accurately describes the system's evolution while in the location reached after the events in string tr have occurred. For example, in the room temperature control case study, if string tr terminates with an event implying that the thermostat is on, query mi(tr) identifies the equation for the exponentially increasing temperature. Currently, mi queries can identify the best-fitting model out of a *pre-determined* set of candidate functions $M \subseteq (\mathbb{R}_+ \to \mathbb{R}^W)$, provided as input to the algorithm as part of the **SUL Configuration** task in Fig. 12.1. Set M' contains the derivatives of functions in M (the flow conditions).

When performing a mi query, firstly, it is necessary to isolate the sam-

Algorithm 2 Teacher function that, given string tr, returns all segments of sampled signals in Σ_{obs} that follow tr.

```
1: function get_segments(tr)
2:     Σ̃ ← ∅
3:     for σ ∈ TEACHER.Σ_obs do
4:         ttr_σ ← {(τ_i, (γ_i, c_i)) : ∃τ_j ∈ π_σ s.t. τ_j = τ_i ∧ L_{π_σ}(τ_j) = (γ_i, c_i)}
5:         tr_σ ← untime(ttr_σ)
6:         if tr ≪ tr_σ then
7:             σ̃ ← {(τ_j, v_j) : (τ_j, v_j) ∈ σ ∧ τ_i < τ_j ≤ τ_k ∧ L(τ_i) = last(tr) ∧ tr_σ =
   tr · tr′ ∧ L_{π_σ}(τ_k) = first(tr′)}
8:             Σ̃ ← Σ̃ ∪ {σ̃}
9:     return Σ̃
```

Algorithm 3 Teacher function computing the answer to a mi query for a specific trace tr.

```
1: function mi(tr)
2:     Σ̃ ← get_segments(tr)
3:     if |Σ̃|=0 then
4:         return ⊥
5:     M̂ ← ∅
6:     for σ̃ ∈ Σ̃ do
7:         d̂ ← +∞; Δ̂ ← +∞; m̂ ← ⊥
8:         for m ∈ M do
9:             φ ← {m(τ_i) | i ∈ [0, |σ̃| − 1] ∧ (τ_i, σ̃_i) ∈ σ̃}
10:            if DDTW(σ̃, φ) < Δ̂ then
11:                Δ̂ ← DDTW(σ̃, φ)
12:                m̂ ← m
13:            else if DDTW(σ̃, φ) = Δ̂ ∧ DTW(σ̃, φ) < d̂ then
14:                d̂ ← DTW(σ̃, φ)
15:                m̂ ← m
16:        M̂ ← M̂ ∪ {m̂}
17:        if |M̂| > 1 then
18:            raise ERROR
19:    return m̂ ∈ M̂
```

pled signals in $\textsc{Teacher}.\Sigma_{\mathrm{obs}} \subseteq \Sigma$ that, intuitively, "follow" a specific event sequence tr (see Algorithm 2 for the pseudo-code of the function get_segments). Specifically, we identify set $\tilde{\Sigma} \subseteq \mathsf{Teacher}.\Sigma_{\mathrm{obs}}$ containing signals $\tilde{\sigma}$ such that:

1. there exists a sampled signal $\sigma \in \mathsf{Teacher}.\Sigma_{\mathrm{obs}}$ of which $\tilde{\sigma}$ is a "*factor*" (i.e., $\sigma = \sigma' \cdot \tilde{\sigma} \cdot \sigma''$ holds for some $\sigma', \sigma'' \in \Sigma$);
2. no events occur during $\tilde{\sigma}$ (i.e., $tr_{\tilde{\sigma}}$ equals ϵ);
3. trace tr under analysis is such that $tr = tr_{\sigma'}$ and $tr_\sigma = tr_{\sigma'} \cdot tr_{\sigma''}$ hold.

Fig. 12.4 plots a sampled room temperature signal σ for the SHA in Fig. 3.1a whose timed trace is $ttr_\sigma = (60, \mathsf{off}) \cdot (270, (\mathsf{open}, \mathsf{on})) \cdot (380, \mathsf{off}) \cdot (530, (\neg\mathsf{open}, \mathsf{on}))$ (each timestamp corresponds to a change-point in π_σ). Signal $\tilde{\sigma}$, also highlighted in Fig. 12.4, following trace $tr = \mathsf{off} \cdot (\mathsf{open}, \mathsf{on}) \cdot \mathsf{off}$ (note that $tr \ll tr_\sigma$ holds) contains all timestamps between 380s and 530s with $tr_{\sigma''} = (\neg\mathsf{open}, \mathsf{on})$.

Given trace tr, for each element in the corresponding set $\tilde{\Sigma}$ (line 6), Algorithm 3 identifies function $m \in M$ that best fits signal $\tilde{\sigma}$ or returns $\perp$ if $\tilde{\Sigma}$ is empty (no observations of tr are available). To this end, we adopt the Derivative Dynamic Time Warping (DDTW) approach [113], which, unlike ordinary Dynamic Time Warping (DTW), assesses the similarity between the time *derivatives* (i.e., the flow condition the query has to identify). In addition to DDTW, we rely on DTW in some (later explained) specific cases. Specifically, for each $\tilde{\sigma} \in \tilde{\Sigma}$ and each function in $m \in M$, Algorithm 3 calculates a new signal ϕ by applying m to the timestamps in $\tilde{\sigma}$ and considering as initial value $m(0) = \tilde{\sigma}_0$ (line 9). The index of similarity between signals $\tilde{\sigma}$ and ϕ is indicated in Algorithm 3 as $\mathrm{DDTW}(\tilde{\sigma}, \phi)$ (line 10, where $\tilde{\Delta}$ stores the current minimum value). The steps required to calculate such an index are presented in detail in [113].

Algorithm 3 identifies $\hat{m} \in M$ resulting in the minimum $\mathrm{DDTW}(\tilde{\sigma}, \phi)$ value and adds it to set $\hat{M}$ (line 16). If two models lead to the same

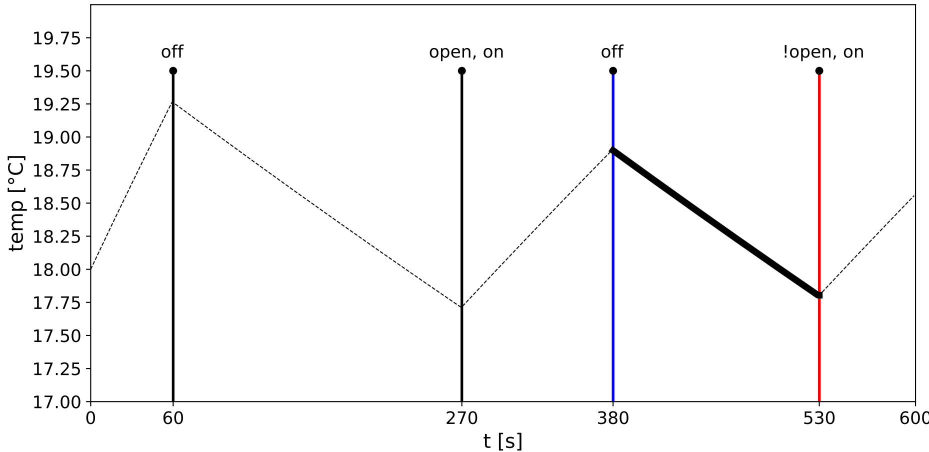


Figure 12.4: *Example of sampled signal factorization. The dashed line plots the room temperature sampled for* 600*s. Labeled vertical lines mark the events identified by the function $\mathcal{L}$. Timestamps marking the start and the end of $\tilde{\sigma}$ are in blue and red, respectively: the solid line is the corresponding $\tilde{\sigma}$.*

Algorithm 4 Function within the TEACHER returning the answer to a ht query.

```
1:  function ht(tr)
2:      Σ̃ ← get_segments(tr)
3:      if |Σ̃| ≤ n_min then
4:          return ⊥
5:      Θ ← {est_param(σ̃) | σ̃ ∈ Σ̃}
6:      D_min ← +∞; m ← |Σ̃|; Θ̂ ← ⊥
7:      for Θ′ ∈ Z do
8:          n ← |Θ′|
9:          D_{m,n}, p-value ← K-S_2sample_test(Θ, Θ′)
10:         if p-value > α ∧ D_{m,n} < D_min then
11:             D_min = D_{m,n};
12:             Θ̂ ← Θ′
13:     if Θ̂ = ⊥ then
14:         Z ← Z ∪ {Θ}
15:         return Θ
16:     else
17:         Θ̂ ← Θ̂ ∪ Θ
18:         return Θ̂
```

$\mathrm{DDTW}(\tilde{\sigma}, \phi)$ value, Algorithm 3 resorts to DTW to pick the one with the minimum distance from $\tilde{\sigma}$ (line 13). Note that, since the algorithm is limited to systems with deterministic edges, the SUL should display the same physical behavior every time the same sequence of events tr takes place. If this is not the case, after analyzing all segments in $\tilde{\Sigma}$, set $\hat{M}$ contains more than one element ($|\hat{M}| > 1$ holds at line 17). The algorithm, then, raises an error (line 18) indicating that the system is showing inconsistent physical behavior (likely due to sensor errors) and the learning procedure cannot continue.

Hypothesis Testing Query ht(tr)

While the mi query deals with the *hybrid* part of the automaton, ht queries characterize the *stochastic* component. Given string tr, query ht(tr), whose implementation is shown in Algorithm 4, performs a series of hypothesis tests to identify whether an empirical distribution describing a sample set in $\mathcal{Z}$ fits a set of (newly) collected samples under analysis. To this end, set $\tilde{\Sigma}$ of signals following tr is isolated through function get_segments identically to mi queries. If $\tilde{\Sigma}$ does not contain enough elements (less than a threshold $n_{\min}$), Algorithm 4 returns $\perp$ indicating that the teacher does not have sufficient observations of tr to answer the query conclusively (lines 3 and 4).

Algorithm 5 Learner function to make an observation table closed (multiple iterations might be necessary).

1: **function** AddRow($\langle S, E, \mathcal{Z}, T\rangle$)
2: **for** $r \in S, a \in A, c \in E$ **do**
3: **if** $T(r \cdot a \cdot c) \neq \bot$ **and**
4: $\forall r' \in S : row(r \cdot a) \not\approx row(r')$ **then**
5: $S \leftarrow S \cup \{r \cdot a\}$
6: **return** $\langle S, E, \mathcal{Z}, T\rangle$

This query only applies to signals that behave stochastically, that is, whose model includes a randomly distributed parameter. The calculation of the parameter value, given a specific signal, through function est_param varies depending on the specific physical variable under analysis, as different types of sampled signals may require different processing. For example, to extract the value of human fatigue rates in the human-robot interaction CPS exemplar, previous approaches rely on processing electromyography signals [139, 156].

After calculating the random parameter sample set Θ from signals in $\tilde{\Sigma}$ (line 5), we verify whether it is plausible that Θ and any $\Theta' \in \mathcal{Z}$ are samples of the same probability distribution. For each identified sample set Θ' (line 7), Algorithm 4 performs a two-sided two-sample Kolmogorov-Smirnov (K-S) test (line 9). The test returns a p-value and a statistic $D_{m,n}$ corresponding to the maximum distance between the two empirical distribution functions, where m is the size of the sample set Θ whose distribution needs identification and n is the size of the candidate set Θ'. If the p-value is greater than a significance level α (fed as an input parameter to $\mathsf{L}^*_{\mathrm{SHA}}$), the null hypothesis that Θ and Θ' are two samples of the same distribution cannot be rejected [90]. Algorithm 4 repeats the procedure for all sets in $\mathcal{Z}$ and stores the one leading to the smaller value of $D_{m,n}$ provided the corresponding test ended with a sufficient significance level (lines 10-12). If the null hypothesis is rejected for all $\Theta' \in \mathcal{Z}$ (hence, $\hat{\Theta} = \bot$ holds at line 13), set Θ represents a new empirical distribution and is added to $\mathcal{Z}$ (line 14). Otherwise, samples in Θ are added to $\hat{\Theta}$ (line 17), i.e., the population identified as the best fit.

Closedness and Consistency

As in L^*, an observation table is *well-defined* if it is *closed* and *consistent*. Well-defined tables can be examined by the teacher to identify a counterexample and employed by the learner for a learning round. Referring to the

Algorithm 6 Learner function to make an observation table consistent (multiple iterations might be necessary).

1: **function** AddCol($\langle S, E, \mathcal{Z}, T\rangle$)
2:     **for** $r_1, r_2 \in S, a \in A, c_1, c_2 \in E$ **do**
3:         **if** $T(r_1 \cdot c_1) \neq \bot$ **and** $T(r_2 \cdot c_2) \neq \bot$ **and**
4: $row(r_1) \simeq row(r_2)$ **and** $row(r_1 \cdot a) \not\simeq row(r_2 \cdot a)$ **then**
5:             $c_3 \leftarrow c \in E : \quad T(r_1 \cdot a \cdot c) \neq \bot \wedge T(r_2 \cdot a \cdot c) \neq \bot \wedge T(r_1 \cdot a \cdot c) \neq T(r_2 \cdot a \cdot c)$
6:             $E \leftarrow E \cup \{a \cdot c_3\}$
7:     **return** $\langle S, E, \mathcal{Z}, T\rangle$

table-like representation of tuple $\langle S, E, \mathcal{Z}, T\rangle$ exemplified in Fig. 12.3, an observation table is *closed* if all unique rows corresponding to long traces “equal" at least one row corresponding to a short trace; it is *consistent* if, for every pair of “equal" rows corresponding to short traces, the pair of rows corresponding to such traces concatenated to one more event are still “equal". An intuitive explanation of why these two properties are relevant is that L^* (thus, by inheritance, $\mathsf{L}^*_{\mathrm{SHA}}$) requires closedness to guarantee that all edges in the hypothesis SHA reach *existing locations* and consistency to build an SHA with *deterministic* edges. The notion of *rows equality* specific to $\mathsf{L}^*_{\mathrm{SHA}}$ is clarified in the following.

Given the stochastic nature of the SUL, some sequences of events may never be observed, although theoretically feasible. Therefore, $\mathsf{L}^*_{\mathrm{SHA}}$ uses the same notions of closedness and consistency as L^*, although based on a different criterion for *row equality* called “*weak equality*” and newly introduced for $\mathsf{L}^*_{\mathrm{SHA}}$. Since $\mathsf{L}^*_{\mathrm{SHA}}$ relies on sampled signals, two rows $row(r_1)$ and $row(r_2)$ are still considered (weakly) equal even if their content differs for some string $c \in E$ as long as at least one sequence of events among $r_1 \cdot c$ and $r_2 \cdot c$ has not been observed yet (i.e., $T(r_1 \cdot c) = \bot \vee T(r_2 \cdot c) = \bot$ holds). Weak equality among two rows is checked through an eq query returning a Boolean value that indicates whether weak equality holds or not.

Definition 6. *Given $\langle S, E, \mathcal{Z}, T\rangle$ and two strings r_1, r_2, $row(r_1)$* weakly equals *$row(r_2)$ ($row(r_1) \simeq row(r_2)$ holds) if, for all $c \in E$ such that $T(r_1 \cdot c) \neq \bot \wedge T(r_2 \cdot c) \neq \bot$ holds, $T(r_1 \cdot c) = T(r_2 \cdot c)$ also holds.*

Closedness and consistency are defined in the following.

Definition 7. *An observation table is* closed *if, for all $r_1 \in (S \cdot A)$, there exists $r_2 \in S$ such that $row(r_1) \simeq row(r_2)$.*

An observation table is closed if, for each long trace, the corresponding row weakly equals at least one row corresponding to a short trace. If this property is not verified, the observation table is modified through function AddRow($\langle S, E, \mathcal{Z}, T\rangle$) (see Algorithm 5) as follows: for each row $r \in S$ and event $a \in A$, if the row corresponding to long trace $r \cdot a$ is missing from the upper part of the table (there is no $r' \in S$ such that $row(r \cdot a) \simeq row(r')$ holds), string $r \cdot a$ is added to set S.

Definition 8. *An observation table is* consistent *if, for all $r_1, r_2 \in S$ such that $row(r_1) \simeq row(r_2)$ holds, then for all $a \in A$, $row(r_1 \cdot a) \simeq row(r_2 \cdot a)$ holds.*

An observation table is consistent if, for each pair of short traces with weakly equal rows, rows corresponding to long trace extensions (with the same symbol $a \in A$) are still weakly equal. If this is not the case, the observation table is modified through function AddCol($\langle S, E, \mathcal{Z}, T\rangle$) (see Algorithm 6) as described in the following. We first identify event sequence $c \in E$ such that, given rows $r_1, r_2 \in S$ and symbol $a \in A$, $row(r_1) \simeq row(r_2)$ holds but $T(r_1 \cdot a \cdot c)$ differs from $T(r_2 \cdot a \cdot c)$ (and both $T(r_1 \cdot a \cdot c)$ and $T(r_2 \cdot a \cdot c)$ differ from $\perp$). Intuitively, strings r_1 and r_2 lead to the same location because $row(r_1) \simeq row(r_2)$ holds but suffix $a \cdot c$ induces different system behaviors (i.e., leading to different locations) that require investigation. String $a \cdot c$ is, therefore, added to set E (a new column is added to the observation table) since it discriminates between prefixes r_1 and r_2 and thus locations reachable with $r_1 \cdot a \cdot c$ and $r_2 \cdot a \cdot c$. The conditions to identify string c and expand E are implemented in function AddCol($\langle S, E, \mathcal{Z}, T\rangle$).

Referring to the diagram in Fig. 12.2, the learner modifies the observation table by adding new rows and columns. The so-created new cells are filled with answers to mi, and ht queries provided by the teacher. The learner keeps modifying the table until it is closed and consistent.

12.2.3 Knowledge Refinement

Since $\mathsf{L}^*_{\mathrm{SHA}}$ relies only on samples to acquire knowledge about the SUL, some traces in $\mathcal{TR}_{\mathrm{obs}}$ may have been observed. Still, the number of observations is not sufficient for mi, and ht queries to produce significant results (hence, their corresponding rows are *weakly informative*). As in the sampling-based adaptation of L^* to Markov Decision Processes learning developed by Tappler et al. [207], in our algorithm, the learner carries out knowledge refinement by asking the teacher to *resample* specific sequences through the ref query. The full implementation of the ref query is shown in Algorithm 7.

Algorithm 7 Function within the TEACHER to run ref queries.

1: **procedure** ref($\langle S, E, \mathcal{Z}, T\rangle$)
2: $\quad \mathcal{ND} \leftarrow \emptyset$
3: $\quad$ **for** $s \in S \cup S \cdot A$ **do**
4: $\quad\quad \tilde{\Sigma} \leftarrow$ get_segments(s)
5: $\quad\quad$ **if** $|\tilde{\Sigma}| < n_{\min}$ **then**
6: $\quad\quad\quad \mathcal{ND} \leftarrow \mathcal{ND} \cup \{s\}$
7: $\quad\quad$ **else**
8: $\quad\quad\quad \mathcal{EQ} \leftarrow \{s' \mid s' \in S \cup S \cdot A \wedge s' \neq s \wedge row(s) \simeq row(s')\}$
9: $\quad\quad\quad$ **if** $\exists s_1, s_2 \in \mathcal{EQ} : row(s_1) \not\simeq row(s_2)$ **then**
10: $\quad\quad\quad\quad \mathcal{ND} \leftarrow \mathcal{ND} \cup \{s\}$
11: $\quad$ **for** $s \in \mathcal{ND}$ **do**
12: $\quad\quad$ TEACHER.$\Sigma_{\text{obs}} \leftarrow$ TEACHER.$\Sigma_{\text{obs}} \cup$ resample($s, n_{\min}$)

Only well-defined tables are eligible for knowledge refinement, which is carried out through resampling at the beginning of a learning round (see Fig. 12.2). Resampling is required if a row has *scarce* observations or if it is *ambiguous*. The first case occurs when a trace in $S \cup S \cdot A$ has been recorded a number of times smaller than a user-defined critical threshold; the second refers to a row that simultaneously weakly equals two rows that differ from each other, indicating that there is a source of ambiguity to be cleared out. This second case occurs because $\mathsf{L}^*_{\text{SHA}}$ is sampling-based, and some traces may be feasible but rare, and thus they may never be observed. The lack of observations associated with a trace is the reason why the notion of weak equality in Definition 6 is *not transitive*; hence it is not an equivalence relation. For example, there may exist $r_1, r_2, r_3 \in (S \cup S \cdot A)$ such that $row(r_1) = \langle\langle f_1, \Theta_1\rangle, \bot\rangle$, $row(r_2) = \langle\langle f_1, \Theta_1\rangle, \langle f_2, \Theta_2\rangle\rangle$, and $row(r_3) = \langle\langle f_1, \Theta_1\rangle, \langle f_3, \Theta_3\rangle\rangle$ hold. In this case, no traces are available to clear the $\bot$, hence, it is known that r_2 captures a different behavior than r_3 and that r_1 *might* be equivalent to r_2 and r_3, but it is not known whether r_1 equals r_2 or r_3 (to this end, the $\bot$ must be cleared). In fact, both $row(r_2)$ and $row(r_3)$ weakly equal $row(r_1)$ but $row(r_2) \not\simeq row(r_3)$ holds.

The value of parameter $n_{\min}$ (line 5)—provided as an input to $\mathsf{L}^*_{\text{SHA}}$—is the minimum acceptable number of observations to determine the behavior corresponding to the associated trace. To properly size $n_{\min}$, the $\mathsf{L}^*_{\text{SHA}}$ user has to balance the *effort* to obtain new traces from the real physical system and the minimum sample size required to run queries with significant results. Previous studies have shown how the Kolmogorov-Smirnov test provides good results with small sample sizes (e.g., $n = 10$) [73] and, for our experiments presented in Chapter 13, we set $n_{\min} = 20$.

To identify rows with scarce observations, the Teacher scans all row labels (line 3) and, for each string $s \in (S \cup S \cdot A)$, it isolates the collected corresponding signal segments (set $\tilde{\Sigma}$ in line 4). If there are less than $n_{\min}$ samples for string s, s is added to a set $\mathcal{ND}$ (line 6). Otherwise, there are enough observations ($|\tilde{\Sigma}| \geq n_{\min}$ holds), thus, for any string $s \in S \cup S \cdot A$, the teacher identifies all strings s', such that $s \neq s'$ holds and $row(s')$ weakly equals $row(s)$, and collects them in set $\mathcal{EQ}$ (line 8). According to Definition 6, it is possible that two distinct rows $row(s_1), row(s_2) \in \mathcal{EQ}$ are such that $row(s_1) \simeq row(s)$, $row(s_2) \simeq row(s)$, and $row(s_1) \not\simeq row(s_2)$ hold, thus giving rise to ambiguity. This situation occurs if there exists $e \in E$ such that $T(s \cdot e)$ is undefined while $T(s_1 \cdot e)$ and $T(s_2 \cdot e)$ are both defined and different from each other. Hence, the ambiguous string s is added to set $\mathcal{ND}$ (line 10).

As a matter of fact, when the learning terminates, the observation table is converted into a SHA, and each set of weakly equal rows (i.e., every set $\mathcal{EQ}$ identified in line 8) is mapped to a different location. Since $row(s_1) \not\simeq row(s_2)$ holds because of symbol e, $row(s_1)$ and $row(s_2)$ would be converted into distinct locations $l_1, l_2 \in L$. However, since $row(s) \simeq row(s_1)$ and $row(s) \simeq row(s_2)$ hold simultaneously, $row(s)$ would be associated with both l_1 and l_2, meaning that events in trace s would non-deterministically lead to either l_1 or l_2. Therefore, the resulting SHA would not be deterministic with respect to transition outputs, and resampling is required to prevent this situation.

All strings in set $\mathcal{ND}$ require resampling. For each sequence in $\mathcal{ND}$, set TEACHER.Σ_{obs} of collected sampled signals is enriched with the additional signals returned by a system-specific function resample. Specifically, function call resample$(s, n_{\min})$ requests the SUL for $n_{\min}$ new sampled signals such that, for each new signal, s is a prefix of the corresponding trace. The implementation of resample depends on how traces are collected in the specific CPS instance under learning (i.e., phase **DC** in Fig. 12.1).

If the SUL is such that events can be programmatically triggered, the function resample can force events in s to collect new observations. Otherwise, if—as in a typical realistic CPS deployment setting—the CPS operates independently of $\mathsf{L}^*_{\mathrm{SHA}}$ and traces are collected as a consequence, resample examines the available pool of traces to find new observations of s. The resampling strategies implemented for the specific case studies presented in the thesis are presented in Chapter 13 and Chapter 15.

Algorithm 8 Function within the TEACHER that returns a counterexample to $\langle S, E, \mathcal{Z}, T\rangle$ if it finds one, $\perp$ otherwise.

1: **function** cex($\langle S, E, \mathcal{Z}, T\rangle$)
2: $\quad \mathcal{TR}'_{\text{obs}} \leftarrow \{tr' \mid tr \in \text{TEACHER}.\mathcal{TR}_{\text{obs}} \wedge tr' \ll tr \wedge tr' \notin S \cup S \cdot A\}$
3: $\quad$ **for** $tr' \in \mathcal{TR}'_{\text{obs}}$ **do**
4: $\quad\quad$ **if** $\exists e \in E : T(tr' \cdot e) \neq \perp$ **then**
5: $\quad\quad\quad$ **if** $\nexists s \in S : \text{row}(s) \simeq \text{row}(tr')$ **then**
6: $\quad\quad\quad\quad$ **return** tr' ▷ non-closedness
7: $\quad\quad\quad$ **else if** $\exists s \in S, a \in A : \text{row}(s) \simeq \text{row}(tr') \wedge \text{row}(s \cdot a) \not\simeq \text{row}(tr' \cdot a)$ **then**
8: $\quad\quad\quad\quad$ **return** tr' ▷ non-consistency
9: $\quad$ **return** $\perp$ ▷ no counterexample

12.2.4 Counterexamples

Like L^*, $\mathsf{L}^*_{\text{SHA}}$ relies on the concept of *counterexample* to determine whether a new round of learning (lines 3 to 17 of Algorithm 9, described in detail in Section 12.2.5) is necessary. Nevertheless, the notion of counterexample in $\mathsf{L}^*_{\text{SHA}}$ is different than L^*. L^* assumes the existence of a teacher with exact knowledge of the language recognized by the DFA under learning (i.e., an "*omniscient*" teacher). An omniscient teacher always answers membership queries (i.e., whether a string is accepted or not by the DFA) correctly and identifies a counterexample with certainty, if at least one exists. In L^*, a counterexample is either: 1. a string accepted by the DFA conjecture but not by the real DFA, or 2. a string accepted by the real DFA but not by the DFA conjecture, and both are identifiable by the omniscient teacher. An omniscient teacher is not feasible in $\mathsf{L}^*_{\text{SHA}}$, which relies on samples of the real physical system. If a sequence of events can be observed in reality, then it is necessarily "accepted". On the other hand, the sampling-based teacher cannot identify the first type of counterexample, that is, event sequences captured by $\mathcal{A}_{\text{hyp}}$ but technically unfeasible.

A counterexample in $\mathsf{L}^*_{\text{SHA}}$ is an event sequence of which observations are available but is not compatible with the current version of $\mathcal{A}_{\text{hyp}}$. More specifically, a counterexample is a string that highlights a source of *non-closedness* or *non-consistency* which is not present in the observation table. In the first case, the counterexample highlights the absence of a location (we recall that each row of the observation table corresponds to a location modeling a specific behavioral state of the SUL) in the observation table. However, sufficient observations of it have been collected. Therefore, a row will be added to the table in the following round. In the second case, if the counterexample highlights non-consistency, it means that a location in

Algorithm 9 Main $\mathsf{L}^*_{\mathrm{SHA}}$ algorithm.

Input: A TEACHER capable of answering mi, ht, eq, ref, and cex queries.
Output: Final SHA conjecture $\mathcal{A}_{\mathrm{hyp}}$.

1: $S \leftarrow \{\epsilon\}$; $E \leftarrow \{\epsilon\}$
2: $T(\epsilon \cdot \epsilon) \leftarrow \langle$TEACHER.mi($\epsilon \cdot \epsilon$), TEACHER.ht($\epsilon \cdot \epsilon$)$\rangle$
3: **do**
4: TEACHER.ref($\langle S, E, \mathcal{Z}, T\rangle$)
5: fill($\langle S, E, \mathcal{Z}, T\rangle$)
6: **while** $\langle S, E, \mathcal{Z}, T\rangle$ is not closed or not consistent **do**
7: **if** $\langle S, E, \mathcal{Z}, T\rangle$ is not closed **then**
8: $\langle S, E, \mathcal{Z}, T\rangle \leftarrow$ AddRow($\langle S, E, \mathcal{Z}, T\rangle$)
9: fill($\langle S, E, \mathcal{Z}, T\rangle$)
10: **if** $\langle S, E, \mathcal{Z}, T\rangle$ is not consistent **then**
11: $\langle S, E, \mathcal{Z}, T\rangle \leftarrow$ AddCol($\langle S, E, \mathcal{Z}, T\rangle$)
12: fill($\langle S, E, \mathcal{Z}, T\rangle$)
13: $tr \leftarrow$ TEACHER.cex($\langle S, E, \mathcal{Z}, T\rangle$)
14: **if** $tr \neq \bot$ **then**
15: $T' \leftarrow \{tr' \mid tr' \ll tr\}$
16: $S \leftarrow S \cup \{tr\} \cup T'$
17: **while** $tr \neq \bot$
18: $\mathcal{A}_{\mathrm{hyp}} \leftarrow$ hyp($\langle S, E, \mathcal{Z}, T\rangle$)
19: **return** $\mathcal{A}_{\mathrm{hyp}}$

the observation table should actually be split into two separate ones (thus, a column will be added to the table in the following round) to make $\mathcal{A}_{\mathrm{hyp}}$ deterministic with respect to transitions.

To find a counterexample, the teacher performs a cex query, shown in Algorithm 8. The teacher scans all collected traces $tr \in$ TEACHER.$\mathcal{TR}_{\mathrm{obs}} \subseteq \mathcal{TR}$ and trace prefixes $tr' \ll tr$ (including tr) which are not already in the observation table (line 2). If there exists $e \in E$ such that $T(tr' \cdot e) \neq \bot$ holds (i.e., there are enough observations for string tr', as per line 4), the teacher first checks for non-closedness. If there is no string $s \in S$ such that $row(s)$ weakly equals $row(tr')$, then tr' is a counterexample (lines 5-6). Secondly, the teacher checks for non-consistency: if there is a string $s \in S$ such that $row(s)$ weakly equals $row(tr')$, but there is also an event $a \in A$ such that $row(s \cdot a)$ differs from $row(tr' \cdot a)$, tr' is a counterexample (lines 7-8). If none of the two conditions apply to any trace or trace prefix that has already been collected, the teacher cannot identify any counterexample and $\mathsf{L}^*_{\mathrm{SHA}}$ terminates.

12.2.5 Complete $\mathsf{L}^*_{\mathrm{SHA}}$ Algorithm

Algorithm 9 shows the main $\mathsf{L}^*_{\mathrm{SHA}}$ learner algorithm, which combines all the functions described in the previous sections. As previously explained, $\mathsf{L}^*_{\mathrm{SHA}}$ requires a teacher capable of answering a set of queries and returns SHA hypothesis $\mathcal{A}_{\mathrm{hyp}}$.

Firstly, the learner initializes the observation table with only one row and one column (sets S and E) labeled by the empty string ϵ (lines $1-2$). $\mathsf{L}^*_{\mathrm{SHA}}$ performs *at least one* learning loop, which starts with a ref query to refine the sample set (line 4), if necessary, and filling the observation table with answers to mi and ht queries as described in Section 12.2.2 through function fill (invoked on line 5 and shown in Algorithm 1). The so-obtained first version of the observation table is checked for closedness and consistency and progressively adjusted until it meets both requirements for well-formedness (lines $6-12$). Finally, the learner asks the teacher for a counterexample through query cex (line 13): if the teacher finds a counterexample, the latter and all its prefixes are added to set S (lines $14-16$). The learning loop stops when the observation table is well-formed, and the teacher cannot find new counterexamples. The algorithm returns automaton $\mathcal{A}_{\mathrm{hyp}}$ obtained from the observation table through function hyp. The implementation of function hyp is analogous to L^* and is not presented in detail for the sake of conciseness.

According to the criteria defined in the literature [191], $\mathsf{L}^*_{\mathrm{SHA}}$ can be classified as an *active* and *offline* algorithm for automata learning since: a) the Teacher can *actively* request new traces to proceed with the learning procedure; b) each collected trace is always available for processing—even *multiple* times—as long as the algorithm is running.

12.2.6 Correctness, Termination, and Complexity

In the following, we sketch an argument on $\mathsf{L}^*_{\mathrm{SHA}}$'s correctness and termination guarantees. Empirical results obtained by applying $\mathsf{L}^*_{\mathrm{SHA}}$ to the case studies are discussed in Chapter 13 and Chapter 15.

Given that the knowledge of the SUL's behavior is limited by samples, the learned SHA cannot fully capture the system's behavior if available traces only provide a partial insight. Whether $\mathsf{L}^*_{\mathrm{SHA}}$ *correctly* learns a SHA limitedly to the set of available traces depends on the teacher's accuracy in answering mi and ht queries (thus, determining the right value of $T(tr)$ for any $tr \in \textsc{Teacher}.\mathcal{TR}_{\mathrm{obs}}$). Therefore, the teacher fails to identify the correct $T(tr) \in (M' \times \mathcal{Z})$ element when the techniques underlying the two queries are at fault.

Keogh and Pazzani assess DDTW's accuracy in terms of *misalignment*, i.e., a measure of *error*. On benchmark experiments, DDTW has shown an adequate degree of accuracy with a mean misalignment ranging from 0.0034 to 0.0053 performing better than ordinary DTW, which results in a mean misalignment ranging from 0.0043 to 0.1278 on the same benchmarks. Since mi queries identify the closest model within finite pre-determined set M the biggest threats to their accuracy are:

1. Set M not being properly configured, which relies on the designer's knowledge of the SUL's physical behavior. Expressly, set M may contain candidate functions that are never identified by mi queries as the best fit (thus, they do not feature in the learned SHA), or it may not contain candidates that would be the best fit. The first case impacts the learning time because each mi query cycles through more candidates than necessary, but not on the learned SHA's correctness. The second case (i.e., missing candidates) compromises correctness since, if the best fitting function for specific signals is not listed as a candidate, the final SHA will feature a less accurate flow condition in the corresponding locations.

2. The SUL's physical behavior being naturally ambiguous; thus, there exists a possibility that the mi query assigns a batch of segments to the wrong cluster. However, if this is the case, set M features two (or multiple) highly similar elements. Therefore, even if mi provides the "wrong" answer, the final SHA error in describing the SUL's behavior is limited. Otherwise, as per Section 12.2.2, if the physical behavior is so fluctuating that different segments following the same trace are clustered into different groups (i.e., $|\hat{M}| > 1$ holds at the end of Algorithm 3), an error is raised since the learning procedure cannot be completed.

Concerning the ht query, although larger sample sizes are preferable, studies have shown that the K-S test is accurate with small samples [73]. In hypothesis testing, two types of errors are possible, given that, with the two-sample K-S test, the null hypothesis assumes that the two samples come from the same distribution:

1. Rejecting the null hypothesis when it is true (i.e., Type I error [17]): the $\mathsf{ht}(tr)$ query erroneously excludes that samples following tr in observed signals (i.e., set Θ in Algorithm 4) belong to a previously identified population in $\mathcal{Z}$. The impact of this error on the correctness of $\mathcal{A}_{\mathrm{hyp}}$ is limited since it results in the SHA having a redundant location

rather than missing information: its likelihood can still be minimized by minimizing the significance level α.

2. Accepting the null hypothesis when it is false (i.e., Type II error [17]): the $\mathsf{ht}(tr)$ query erroneously concludes that samples observed after tr belong to a previously identified population in $\mathcal{Z}$. This is the most critical error for the ht query since the final SHA lacks information about the SUL (specifically, the distribution it failed to identify). This error grows less likely with large sample sizes. Thus its impact can be minimized by increasing $n_{\min}$, if possible, or performing new iterations of $\mathsf{L}^*_{\mathrm{SHA}}$ whenever new traces are available.

Unlike in L^*, the $\mathsf{L}^*_{\mathrm{SHA}}$ teacher is not omniscient but relies on collected samples. Therefore, it cannot distinguish *impossible* traces (i.e., sequences of events whose probability of being observed in the real system is 0) from *rare* traces because $T(tr) = \bot$ holds in both cases. Assume, instead, that an omniscient teacher is available for $\mathsf{L}^*_{\mathrm{SHA}}$. In this case, having sufficient observations to answer mi and ht queries is no longer an issue since an omniscient teacher provides the correct value of $T(tr) \in (M' \times \mathcal{Z})$ for any *possible* trace tr—hence $T(tr) = \bot$ (capturing the situation in which not enough observations of tr have been collected to answer queries conclusively) never holds. If one introduces a different symbol $\otimes$ to capture the case in which tr is impossible (hence, $T(tr) = \otimes$ holds), $\mathsf{L}^*_{\mathrm{SHA}}$ behaves as L^*, whose correctness and termination are proven in [7].

Nevertheless, an omniscient teacher is not implementable when a real SUL is involved since the source of information on its behavior is a *finite* set of traces. The sampling-based version of $\mathsf{L}^*_{\mathrm{SHA}}$ (presented in this section) terminates when the teacher cannot find a new counterexample within the available traces. Were set of traces TEACHER.Σ_{obs} to remain unvaried throughout the execution of $\mathsf{L}^*_{\mathrm{SHA}}$, in the worst case in which all collected traces constitute a counterexample (thus, a round of learning is performed for each collected trace) $\mathsf{L}^*_{\mathrm{SHA}}$ terminates because the number of traces is finite. However, as per Section 12.2.3, the learner can request new traces at the beginning of each learning round through the ref query, meaning that the number of traces, although finite, grows during the execution of $\mathsf{L}^*_{\mathrm{SHA}}$. As discussed in Section 12.2.4, identifying a counterexample always leads to creating a new location of $\mathcal{A}_{\mathrm{hyp}}$, either because a completely new behavioral state has been identified or because an existing location should be split into two. Therefore, even if the set of traces grows at the beginning of each round, termination is still guaranteed *provided that* counterexamples are identified *correctly*, and the original SUL has finite behavioral states

(thus, only a finite number of counterexamples can be identified).

The temporal complexity of L^*, of which $\mathsf{L}^*_{\mathrm{SHA}}$ inherits the core infrastructure, is proven to be polynomial in the size of the observation table and maximum length of a counterexample [7]. $\mathsf{L}^*_{\mathrm{SHA}}$ differs from L^* because of mi, ht, ref, and cex queries, whose complexity is estimated in the following.

Each mi query applies DDTW to every signal portion in set $\tilde{\Sigma}$ and every candidate function in M. Therefore, the cost of performing a mi query is polynomial in the number of available traces (from each trace, a signal portion may be isolated), the number of candidate functions, and the cost of performing DDTW, which is shown to be polynomial in the number of points in the two signal portions under analysis [113].

Each ht query, for each element of set $\mathcal{Z}$, performs a Kolmogorov-Smirnov test, of which a possible implementation is proven to be linear in the number of sample points [84]. However, the size of $\mathcal{Z}$ may grow after the execution of a ht query if a new distribution is identified. Suppose a new distribution is identified for every ht query. In that case, its cost is polynomial in the number of cells of the observation table, thus the number of traces and the maximum length of a trace.

Each ref query searches for ambiguous rows and requests new observations. Therefore, its running time is polynomial in the number of available traces and the maximum length of a trace (there may be a row for each trace and trace prefix). The cost of one execution of function resample$(s, n_{\min})$, although its implementation is SUL-specific, is assumed to be constant and proportional to parameter $n_{\min}$. It is executed once for each ambiguous trace (thus, at most, as many times as the number of available traces).

Each cex query searches for a counterexample among the available traces: therefore, its running time is also polynomial in the number of available traces, the number of rows, and the number of columns (both depending on the number of traces and the maximum length of a trace).

In conclusion, the overall running time of $\mathsf{L}^*_{\mathrm{SHA}}$ remains polynomial in the metrics mentioned above concerning the volume of input data (number and length of traces) and the number of candidate flow conditions. Performance metrics of $\mathsf{L}^*_{\mathrm{SHA}}$ for the validation experiments are reported in Chapter 13 and Chapter 15.

CHAPTER 13

$\mathsf{L}^*_{\mathrm{SHA}}$ Empirical Validation

This chapter reports on the experiments carried out to empirically validate $\mathsf{L}^*_{\mathrm{SHA}}$.

The validation is carried out with two approaches. In the first one, the original SHA is exploited as a reference to assess the learned SHA's accuracy. In the second one, the learned SHA's accuracy is assessed in comparison to a validation dataset.

To this end, two case studies are exploited to showcase the general-purpose nature of $\mathsf{L}^*_{\mathrm{SHA}}$. The first one consists of a room heating system. The second one involves energy consumption models of machining centers.

13.1 Validation Process

Validation is carried out with two approaches that differ based on how traces are generated and how $\mathsf{L}^*_{\mathrm{SHA}}$ accuracy is measured:

A. **Model-Driven**: reference models of the CPS component subject to uncertainty (thus, whose model requires learning) are manually set up in a formal modeling and verification tool (specifically, Uppaal).

Traces capturing CPS behavior are generated by Uppaal through the simulate operator [46] and fed to $\mathsf{L}^*_{\mathrm{SHA}}$. $\mathsf{L}^*_{\mathrm{SHA}}$'s accuracy is assessed through manual inspection by comparing the learned SHA with the reference SHA, which, with this approach, is known. When knowledge refinement is required, the teacher also exploits Uppaal to request new traces. Experiments are reported in Section 13.2.

B. **Simulation-Driven**: real sensor readings are grouped into longer sequences of sampled signals to create realistic simulated field data. Simulation traces are stored in a pool and then fed to $\mathsf{L}^*_{\mathrm{SHA}}$ to learn a SHA modeling the CPS component subject to uncertainty. The accuracy of the learned SHA is assessed through system-specific quality metrics of the CPS. Such indicators are calculated through Uppaal for the learned SHA and extracted from simulation traces for the reference system. The teacher extracts new traces from the available pool when knowledge refinement is required. Simulation-driven experiments are reported in Section 13.3.

Validation exploits two case studies. The first CPS instance (indicated as CPS1) is a room subject to temperature control through a thermostat, also serving as a running example in Chapter 3. While the thermostat is a programmable component, the room's temperature dynamics are subject to uncertainty due to structural variations (e.g., a window can be opened or closed by people in the room). $\mathsf{L}^*_{\mathrm{SHA}}$ is validated on CPS1 through the model-driven approach.

The second CPS instance (indicated as CPS2) focuses on the smart manufacturing domain, specifically on modeling the energy consumption of machining centers. In such applications, the sequence of tasks performed on the workpiece is fixed, while the power request of the machine is a source of uncertainty given the broad spectrum of parameters at play. CPS2 is exploited to validate $\mathsf{L}^*_{\mathrm{SHA}}$ through the simulation-driven approach.

The implementation of $\mathsf{L}^*_{\mathrm{SHA}}$ used to run the experiments is found at [133]. Experimental results are reported in [133]. Learning experiments with $\mathsf{L}^*_{\mathrm{SHA}}$ have been performed on an OSX machine with 2 cores and $8\mathrm{GB}$ of RAM. Formal models are implemented in Uppaal v.4.1.24 using the extension for SMC [46, 123]. SMC experiments are performed on a Linux machine with 128 cores, $515\mathrm{GB}$ of RAM, and Debian Linux version 10.

In the following, we report on each validation phase in detail and present the obtained results.

13.2 CPS1: Room Temperature Control System

In the first case study, the CPS component subject to uncertainty is the room (equipped with the heating system), of which we learn the temporal behavior of the temperature in different operational states due to windows being open/closed and the heating system being on or off.

More specifically, differential equations governing the temperature are known, and distributions governing random parameters θ and R are known to be Normal. Still, the mean and variance of such distributions may vary based on the room's structural properties (i.e., whether any window is open). Set $\mathcal{Z}$ contains sample sets of parameter R associated with locations capturing the room's temperature dynamics when the thermostat is off, and samples θ otherwise (see Fig. 3.1a).

Each location of $\mathcal{A}_{\text{hyp}}$ is associated with an element in $M' \times \mathcal{Z}$. For this specific case study, M' is the set of flow conditions constraining the room's temperature in different situations. Therefore, values of real-valued variable T constitute the first sampled signal in $\text{TEACHER}.\Sigma_{\text{obs}}$. Featured flow conditions (constituting set M') are of two types: i) those reported in Fig. 3.1a, modeling exponential dynamics; ii) $T(t) = \theta t$ and $T(t) = -\frac{1}{\mathsf{R}}t$, modeling the temperature varying linearly with time (introduced to enrich the running example model).

Observable events (whose number for each experiment is reported in Table 13.1) that can occur in the system and that constitute the traces used for learning concern the heating system being switched on or off with guard conditions on how many windows are open, thus decreasing the room's capability to retain heat and impacting the distributions of R and θ. The actual opening or closing of a window does not constitute an event to reduce the complexity of the reference model. Specifically, events are of the form $\langle open = x, \mathtt{c}\rangle$ with $x \in [0, 2]$ and $\mathtt{c} \in \{\mathtt{on}, \mathtt{off}\}$. Discrete variable $open \in V_{\text{dc}}$ marks the number of open windows in the room, and its values constitute the second sampled signal.

As previously mentioned, CPS1 is subject to the model-driven validation approach. To test the efficacy of $\mathsf{L}^*_{\text{SHA}}$ against known models, we have manually created through Uppaal 10 increasingly complex SHA modeling the room, of which Fig. 3.1a shows an example. Each manually-created SHA constitutes the reference model for a standalone experiment and is the source of traces fed to $\mathsf{L}^*_{\text{SHA}}$. When performing the ref query, the teacher asks Uppaal for new runs to be added to $\text{TEACHER}.\Sigma_{\text{obs}}$ whenever a specific event sequence has less than $n_{\min} = 20$ observations. The thermostat is fully programmable, and thus its SHA (shown in Fig. 3.1b) remains

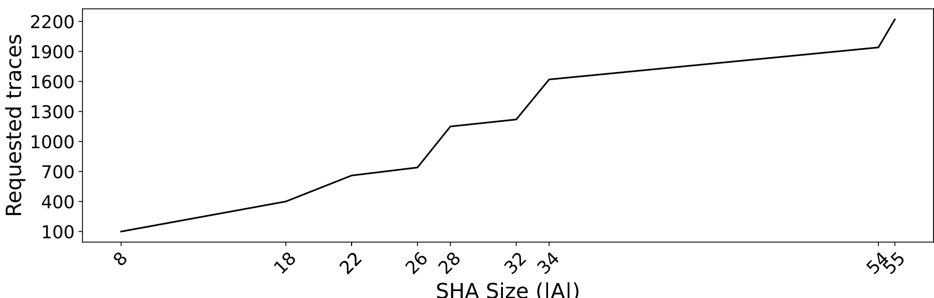
Table 13.1: *Metrics comparing the original SHA for CPS1 modeled in Uppaal (labeled as SUL) and the learned SHA (labeled as $\mathsf{L}^*_{\mathrm{SHA}}$) for the* 10 *experiments. The two SHA are compared based on the number of locations ($|L|$), edges ($|\mathcal{E}|$), flow conditions ($|M'|$) and distributions ($|\mathcal{Z}|$).*

Exp.		Events	$\lvert L\rvert$	$\lvert\mathcal{E}\rvert$	$\lvert M'\rvert$	$\lvert\mathcal{Z}\rvert$
1	SUL/$\mathsf{L}^*_{\mathrm{SHA}}$	2	2	2	2	2
2	SUL/$\mathsf{L}^*_{\mathrm{SHA}}$	4	4	8	2	4
3	SUL/$\mathsf{L}^*_{\mathrm{SHA}}$	3	6	18	2	6
4	SUL/$\mathsf{L}^*_{\mathrm{SHA}}$	4	5	10	2	5
5	SUL/$\mathsf{L}^*_{\mathrm{SHA}}$	4	5	10	2	5
6	SUL	4	5	11	2	5
	$\mathsf{L}^*_{\mathrm{SHA}}$		7	14	2	5
7	SUL	4	6	14	2	6
	$\mathsf{L}^*_{\mathrm{SHA}}$		7	14	2	6
8	SUL/$\mathsf{L}^*_{\mathrm{SHA}}$	6	6	18	4	6
9	SUL	6	7	21	3	8
	$\mathsf{L}^*_{\mathrm{SHA}}$		11	33	3	8
10	SUL	6	7	21	3	7
	$\mathsf{L}^*_{\mathrm{SHA}}$		11	33	3	7

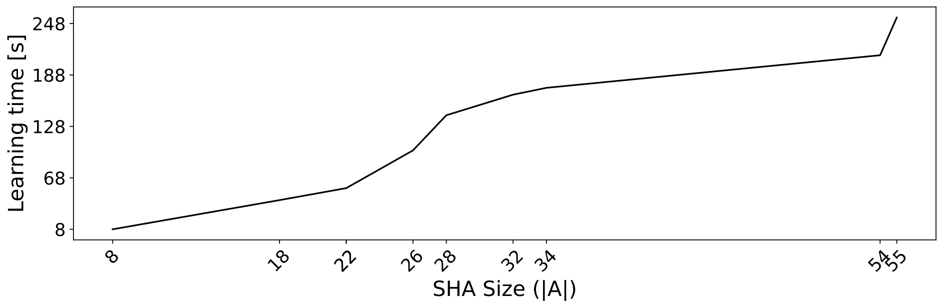
unchanged throughout all experiments. Updates of variable $open$ are randomly triggered through Uppaal. We remark that $\mathsf{L}^*_{\mathrm{SHA}}$ does not learn what causes a specific event to fire (which is random) but how the room behaves once it has fired.

Experimental results are reported in Table 13.1, which compares the reference manually-drafted SHA (labeled as SUL) with the SHA learned by $\mathsf{L}^*_{\mathrm{SHA}}$. The comparison takes into account the main SHA features involved in the learning process: locations (set L), edges (set $\mathcal{E}$), flow conditions (the outcome of the function $\mathcal{F}$) and probability distributions (the outcome of the function $\mathcal{D}$) assigned to each location. Learned SHA are fully viewable in [133] and Appendix A.

Manual inspection and data reported in Table 13.1 show that $\mathsf{L}^*_{\mathrm{SHA}}$ accurately learns the SHA in all experiments. Experiments 6, 7, 9, and 10 report *apparent* discrepancies between the SUL and the learned SHA, for which two distinct rows are shown, unlike for the other experiments. The origin of such discrepancies is explained in the following. In these experiments, reference SHA feature one (or multiple) edges whose guards include con-



(a) *Traces requested by* $\mathsf{L}^*_{\text{SHA}}$ *to complete the learning for all CPS1 experiments.*



(b) *Time (s) requested by* $\mathsf{L}^*_{\text{SHA}}$ *to complete the learning for all CPS1 experiments.*

Figure 13.1: *Plots reporting on* $\mathsf{L}^*_{\text{SHA}}$ *performance with increasing SUL complexity, measured through the learned SHA size (*$|\mathcal{A}|$*).*

straints on a discrete variable in V_{dc} that is *not observable*. For example, consider a location $l \in L$ with two outgoing edges with the same channel label and guards "$open \wedge w = 0$" and "$open \wedge w = 1$", where $w \in V_{\text{dc}}$ is not observable. As per Chapter 12, the $\mathsf{L}^*_{\text{SHA}}$ labeling function cannot account for non-observable features. In this case, not considering variable w makes the two edges identical since they have the same channel label and the same guard (condition "$open$"), causing $\mathcal{A}_{\text{hyp}}$ to be nondeterministic (i.e., *not consistent* according to Definition 8). $\mathsf{L}^*_{\text{SHA}}$ resolves the inconsistency through locations $l_1, l_2 \in L$ (each with one outgoing edge) representing l with $w = 0$ and l with $w = 1$, respectively. Manual inspection shows learned SHA in experiments 6, 9, and 10 behave correspondingly with the original SHA albeit featuring more locations and edges.

Data on $\mathsf{L}^*_{\text{SHA}}$ performance for CPS1 experiments are reported in Fig. 13.1 as a function of the complexity of the SHA under learning. The latter is measured in terms of SHA **size**, which is indicated as $|\mathcal{A}|$ and calculated as the sum of $|L|$, $|\mathcal{E}|$, $|M'|$, and $|\mathcal{Z}|$. As per Section 12.2.4, $\mathsf{L}^*_{\text{SHA}}$ termi-

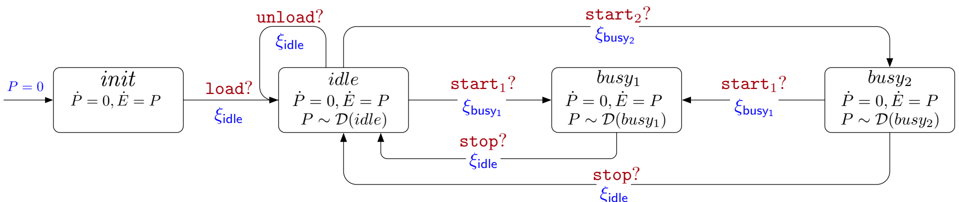


Figure 13.2: *Example SHA modeling a machine's power demand and energy consumption.*

nates when the teacher does not identify a counterexample to $\langle S, E, \mathcal{Z}, T\rangle$ in TEACHER.$\mathcal{TR}_{obs}$. We report on the number of **traces** requested by the teacher to terminate (see Fig. 13.1a) and the duration of each experiment (see Fig. 13.1b). As previously discussed, all learned SHA are correct. As per Fig. 13.1a, the most resource-intensive experiment (i.e., experiment 9, with $|\mathcal{A}| = 55$) has required 2220 traces to terminate. In terms of computation time, all experiments for CPS1 have been completed in a time ranging from 8s (i.e., experiment 1, with $|\mathcal{A}| = 8$) to 4min15s for experiment 9.

13.3 CPS2: Energy Consumption of Machining Centers

The application domain of CPS2 involves the prediction of machine energy consumption in industrial applications. Specifically, we focus on the spindle energy consumption. The learned SHA captures the correlation between the machining tasks, executed accordingly to a set of designed operations, and the spindle power request.

The approach is designed to work even under partial knowledge of the machining task. Therefore, it mandatorily requires only three signals: the spindle speed, which is one of the main cutting parameters, the clamping pressure, which indicates the presence of a workpiece in the machine; and the spindle energy consumption. Spindle speed and clamping pressure are also referred to as *event mining* signals as they highlight contingencies in the SUL significantly affecting the physical property under learning (i.e., the energy consumption).

The learned SHA, of which Fig. 13.2 shows an example, enables the prediction of the spindle power request of a certain machining task given the event mining signals.

For this CPS exemplar, set W of the automaton in Fig. 13.2 contains variables P and E modeling the power request and energy consumption of a machine, respectively. In Fig. 13.2, P is understood as a function $P(t, k)$ and E as $E(t, k)$, where t is the variable representing time and k is

an independent randomly distributed term.

For every location $l \in L$, $\mathcal{D}(l)$ characterizes E while the machine behavior is modeled by location l. Therefore, the domain of function $\dot{P}$ and $\dot{E}$ is $\mathbb{R}_+ \times \mathbb{R}$ and P and E are interpreted as stochastic processes. The two locations in Fig. 13.2 are such that $\mathcal{F}(l) = \langle \dot{P} = 0, \dot{E} = k \rangle$ holds for all locations $l \in \{idle, busy\}$, meaning that power P is constant over time and equal to k thus E is linear over time with slope k.

Event Mining

Collected data is processed to extract the timed traces. The event mining signals hereby considered are the spindle speed and the clamping pressure. Let $\mathrm{p}(t_i) \in \mathbb{R}_+$ be the clamping pressure value measured at time t_i and $\mathrm{s}(t_i) \in \mathbb{R}_+$ be the spindle speed value measured at time t_i.

A necessary condition for the machining process to start is the presence of the workpiece loaded into the machine. Events `load` and `unload` are respectively representing the beginning and the end of part-program execution and need to be identified. Signal $\mathrm{p}(t_i)$ is used to identify `load` and `unload` events as for sudden variations of the clamping pressure, where $\mathsf{p}_{\mathsf{min}}$ is the minimum pressure required to block the workpiece.

Variations in spindle speed might result in a different power request. Thus, speed variations are events to be extracted from signal $s(t_i)$. The spindle speed signal is discretized into a sequence of step-like signals to bypass small variations.

The spindle speed range is technologically limited in $[\mathsf{s}_{\mathsf{min}}, \mathsf{s}_{\mathsf{max}}]$ where $\mathsf{s}_{\mathsf{max}}$ is the nominal maximum speed of machine's spindle and $\mathsf{s}_{\mathsf{min}}$ is the minimum speed to consider the spindle active. The range is divided into m bins of equal width $\delta_{\mathrm{s}} = \mathsf{s}_{\mathsf{max}} - \mathsf{s}_{\mathsf{min}}/m$. Specifically, for the three scenarios, $\mathsf{s}_{\mathsf{min}} = 100$ rpm, $\mathsf{s}_{\mathsf{max}} = 10'000$ rpm, and $m = 50$ according to expert knowledge. Events are identified whenever the spindle speed signal switches between two bins. We label as $\mathtt{start}_j$ with $j = 1 \ldots m$ the event of the spindle reaching a speed within the j-th range defined as $(\mathsf{s}_{\mathsf{min}} + (j - 1)\delta_{\mathrm{s}}, \mathsf{s}_{\mathsf{min}} + j\delta_{\mathrm{s}}]$. When the spindle speed drops below $\mathsf{s}_{\mathsf{min}}$, the spindle stops (event `stop`).

Function $\mathcal{L}$ is fully given in Eq.13.1:

$$\mathcal{L}(t_i) = \begin{cases} \texttt{load}, & \text{if } i > 0 \wedge \mathrm{p}(t_i) \geq \mathsf{p}_{\mathsf{min}} \wedge \mathrm{p}(t_{i-1}) < \mathsf{p}_{\mathsf{min}} \\ \texttt{unload}, & \text{if } i > 0 \wedge \mathrm{p}(t_i) < \mathsf{p}_{\mathsf{min}} \wedge \mathrm{p}(t_{i-1}) \geq \mathsf{p}_{\mathsf{min}} \\ \texttt{stop}, & \text{if } i > 0 \wedge \mathrm{s}(t_i) < \mathsf{s}_{\mathsf{min}} \wedge \mathrm{s}(t_{i-1}) \geq \mathsf{s}_{\mathsf{min}} \\ \texttt{start}_j, & \text{if } i > 0 \wedge \mathrm{s}(t_i) \in (\mathsf{s}_{\mathsf{min}} + (j-1)\delta_{\mathrm{s}}, \mathsf{s}_{\mathsf{min}} + j\delta_{\mathrm{s}}] \\ & \wedge\, \mathrm{s}(t_{i-1}) \notin (\mathsf{s}_{\mathsf{min}} + (j-1)\delta_{\mathrm{s}}, \mathsf{s}_{\mathsf{min}} + j\delta_{\mathrm{s}}] \\ \bot, & \text{otherwise} \end{cases} \tag{13.1}$$

Learned SHA Post-Processing

Data is split into training and validation datasets. Training data is fed to $\mathsf{L}^*_{\mathrm{SHA}}$. Post-processing is then necessary to make the learned SHA amenable to trace-based simulation [123]. Specifically, edges of the learned SHA are extended with updates while a distribution estimate is assigned to each location.

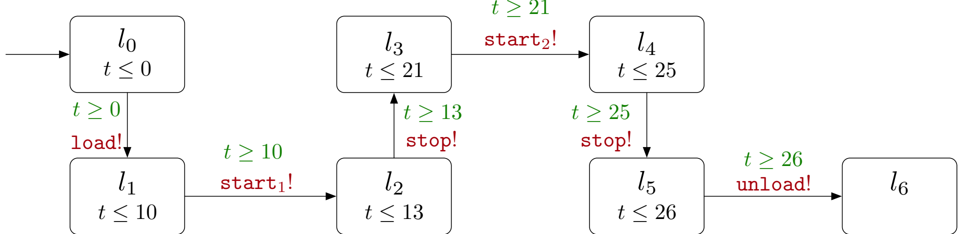
The SHA learned through $\mathsf{L}^*_{\mathrm{SHA}}$ features, for every location $l \in L$, an empirical distribution identified by ht queries. $\mathsf{L}^*_{\mathrm{SHA}}$ has no constraints on the specific shape of distribution functions; therefore, empirical functions identified by ht queries may converge to any arbitrary distribution.

Let $X(l) \subseteq \mathbb{R}_+$ be the sample set underlying the empirical distribution assigned to $l \in L$. The Gaussian Kernel Density Estimation (KDE) method [169, 186] is applied to $X(l), \forall l \in L$ to estimate $\mathcal{D}(l)$. KDE is a well-established non-parametric method to estimate the probability density function of a random variable based on kernels as weights that does not imply any prior distribution. Gaussian KDE specifically uses the normal kernel function. KDE requires the proper selection of a kernel smoothing parameter $h > 0$ called *bandwidth*, which is calculated through the Silverman's approximation method [198].

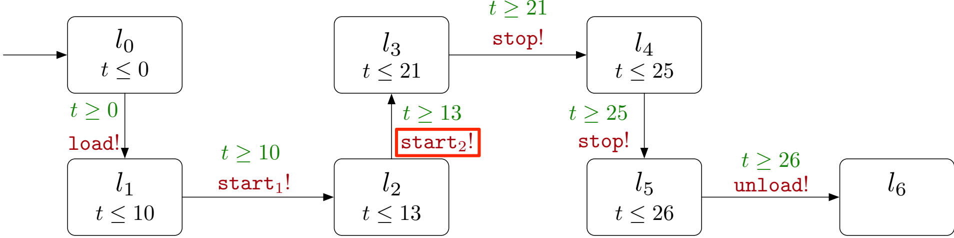
For all locations $l \in L$ of the learned SHA, the empirical distribution is replaced with the corresponding kernel density estimator of $\mathcal{D}(l)$.

Learned SHA Validation

The validation dataset is used to validate the SHA resulting from post-processing. The compatibility check filters the validation dataset to identify traces eligible for trace-driven simulation. The estimated energy consumption is compared against field data to assess the learned model's accuracy. To improve model accuracy, new training traces should be collected (loop in Fig. 4.1).



(a) *Controller reproducing a trace* compatible *with the machine SHA in Fig. 3.1.*



(b) *Controller reproducing a trace* not *compatible with the machine SHA in Fig. 3.1 (the highlighted event causes the incompatibility).*

Figure 13.3: *Examples of controller SHA. Each location is represented with its label and invariant.*

Verifying whether a trace is compatible with the machine SHA entails the generation of a *controller* SHA to compose a SHA network. The controller SHA mimics the *timed* sequence of events of the trace under analysis. The trace determines the structure of the controller SHA, which, thus, does not require learning through $\mathsf{L}^*_{\mathrm{SHA}}$. In all controller SHA, $t \in W$ is a real-valued variable that grows uniformly with time (i.e., $\dot{t} = 1$ holds in all locations).

An edge labeled with $\mathtt{c}!$ with $\mathtt{c} \in C$ cannot fire (thus, time cannot flow) unless another SHA in the network has an edge enabled with label $\mathtt{c}?$. If all edges of the controller fire successfully, the machine SHA captures the trace under analysis: hence, the trace is *compatible* with the learned SHA and eligible for trace-based simulation.

For instance, let us assume that the validation dataset contains two traces, namely:

$$\langle \mathtt{load}, \mathtt{start}_1, \mathtt{stop}, \mathtt{start}_2, \mathtt{stop}, \mathtt{unload} \rangle$$

$$\langle \mathtt{load}, \mathtt{start}_1, \mathtt{start}_2, \mathtt{stop}, \mathtt{stop}, \mathtt{unload} \rangle$$

with events at timestamps $\langle 10, 13, 21, 25, 26 \rangle$ in both cases. Traces compatibility with the machine SHA in Fig. 3.1 needs to be verified, and Fig. 13.3 shows the two controller SHA corresponding to the first and second trace,

respectively. The controller shown in Fig. 13.3a mimics a trace compatible with the machine SHA in Fig. 3.1, while the trace mimicked by the SHA in Fig. 13.3b is not compatible because the edge from l_2 to l_3 cannot fire.

Trace-driven simulation is executed for compatible traces. To this end, the machine SHA is paired with the controller SHA mimicking the trace under analysis. The resulting SHA network is subject to trace-based simulation through the Uppaal tool for formal modeling, and verification [46]. The tool generates $\mathsf{N}_{\mathsf{val}} \in \mathbb{N}$ runs of the system simulating the power P in response to the events of the trace under analysis (fired by the controller SHA).

Whenever an edge incoming location $l \in L$ fires, update ξ_l draws a sample of P from estimated $\mathcal{D}(l)$ through an acceptance/rejection algorithm [35]. However, trace-based simulation requires the estimation of probability density function $\mathcal{D}(l)$ to generate random samples through updates ξ_l [46]. From each of the $\mathsf{N}_{\mathsf{val}}$ generated simulations, Uppaal estimates the expected energy consumed to execute the trace $\mathbb{E}[E]$ by integrating the power P over the simulated time.

Design of Experiments

Three scenarios of increasing complexity are devised:

1. Scenario A represents a mass production application where high volumes of the same type of product are machined.

2. Scenario B represents the production of a family of products (i.e., three product types).

3. Scenario C represents the production of highly customized products. Machines work on unique solutions executing part-programs tailored to client requests, where the production of individual workpieces can be entirely different.

All experiments are performed on a machine running Ubuntu 22.04 with 64GB of memory and 4 cores. Trace-based simulation is performed using Uppaal v.4.1.24.

Numerical results are obtained by performing 10 experiments, summed up in Table 13.2 and characterized as follows:

- Six instances of scenario A are created starting from literature data, specifically the 27 machining tasks presented in [134]. Data has been combined to create the sequence of machining tasks for six part types.

Table 13.2: *Design of experiments (Scenarios A, B, and C).*

Exp.	Scenario	Part mix	Part type	Trace length
$1-6$	A	$i-vi$	Single	$p=6$ / $p=12$
$7-9$	B	i,ii,iii / iv,v,vi / i,iv,vi	Multiple	$p=6$ / $p=12$
10	C	Unknown	Multiple	$p \in [6,21]$

Types i,ii,iii consist of $p=6$ tasks, while part types iv,v,vi consist of $p=12$ tasks. Experiments 1 to 6 refer to part types $i-vi$, respectively. Training and validation datasets have been generated by concatenating the signals available and adding white noise.

- Three instances of scenario B are created starting from literature data [134]. Each instance includes three part types, respectively: experiment 7 features types i,ii,iii, experiment 8 features iv,v,vi, and experiment 9 features i,iv,vi. Training and validation datasets have been generated by adding white noise to the source signals.

- Experiment 10 refers to scenario C. The part mix is unknown. As for the field acquisition, part types include up to 21 tasks (with a minimum of 6) executed sequentially to produce a single workpiece. Training and validation datasets are extracted from field data.

Experimental Results

The approach is applied to problem instances by increasing the number of N_{tr} training traces. For scenarios A and B, we consider $\mathsf{N}_{\mathsf{tr}} = 3, 10, 20$ traces for each produced part type to train $\mathsf{L}^*_{\mathrm{SHA}}$, whilst for scenario C, we consider $\mathsf{N}_{\mathsf{tr}} = 5, 7, 9$ traces.

Table 13.3, Table 13.4, and Table 13.5 report on $\mathsf{L}^*_{\mathrm{SHA}}$ performance for scenarios A, B, and C, respectively.

The training dataset volume is proportional to N_{tr} (i.e., the number of traces) and execution time of the part-program, which is the acquisition time to obtain a single trace. The acquisition time is reported in Table 13.3 for each part type $i-vi$. For instances of scenario A, the average part-program execution time to obtain a single item is 1331 s (22.19 minutes). Thus, the training dataset for a single instance of scenario A includes on average 1.1h, 3.7h, and 7.4h of acquisition for respectively $\mathsf{N}_{\mathsf{tr}} = 3, 10, 20$. Similarly an instance of scenarios B includes the production of three items and requires on average 3.7h, 12.2h, and 24.4h of acquisition for respectively $\mathsf{N}_{\mathsf{tr}} = 9, 30, 60$. Regarding the instance of scenario C, a single trace

Table 13.3: $\mathsf{L}^*_{\mathrm{SHA}}$ *running time ([s]) for scenario A broken down by experiment, produced part mix, number of locations and edges of the learned SHA, and number of training traces* N_{tr} *with related acquisition time ([h]).*

Exp.	**Part Mix**	$\lvert L \rvert$	$\lvert \mathcal{E} \rvert$	N_{tr}	**Acquisition Time** [h]	**Learning Time** [s]
1	i	8	11	3	1.00	8.84
				10	3.33	16.63
				20	6.65	18.40
2	ii	12	12	3	0.46	4.67
				10	1.53	8.35
				20	3.06	10.27
3	iii	11	12	3	0.93	7.97
				10	3.09	15.58
				20	6.18	19.94
4	iv	12	16	3	1.77	46.37
				10	5.91	52.44
				20	11.82	67.66
5	v	17	19	3	0.95	32.29
				10	3.16	48.87
				20	6.31	58.24
6	vi	23	24	3	1.55	58.03
				10	5.18	81.93
				20	10.35	99.78

Table 13.4: $\mathsf{L}^*_{\mathrm{SHA}}$ *running time ([s]) for scenario B broken down by experiment, produced part mix, number of locations and edges of the learned SHA, and number of training traces* N_{tr} *with related acquisition time ([h]).*

Exp.	**Part Mix**	$\lvert L$	$\lvert \mathcal{E} \rvert$	N_{tr}	**Acquisition Time** [h]	**Learning Time** [s]
7	i, ii, iii	20	25	3×3	4.32	49.28
				10×3	14.41	76.59
				20×3	28.82	146.21
8	iv, v, vi	48	54	3×3	2.38	247.34
				10×3	7.95	354.32
				20×3	15.89	659.59
9	i, iv, vi	30	36	3×3	4.27	146.08
				10×3	14.24	204.43
				20×3	28.48	385.04

requires on the average 1.44h; thus it includes a total of 5.5h, 8h, and 13h of acquisition, respectively for $\mathsf{N}_{\mathsf{tr}} = 5, 7, 9$ training traces.

Concerning the SHA complexity, flow conditions and real-valued variables do not vary across experiments. Since each location is endowed with at most one probability distribution, the number of locations and edges of

Table 13.5: $\mathsf{L}^*_{\mathrm{SHA}}$ *running time* ([s]) *for the experiment of scenario C broken down by the number of locations and edges of the learned SHA, and number of training traces* N_{tr} *with related acquisition time [h].*

Exp.	$\|L\|$	$\mathcal{E}$	N_{tr}	Acquisition time [h]	Learning Time [s]
	15	34	5	5.5	568.05
10	30	60	7	8	1216.49
	42	83	9	13	4119.48

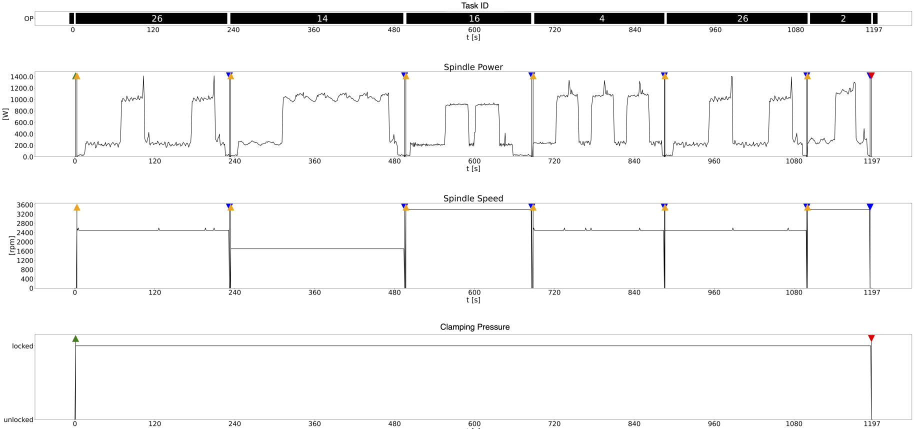
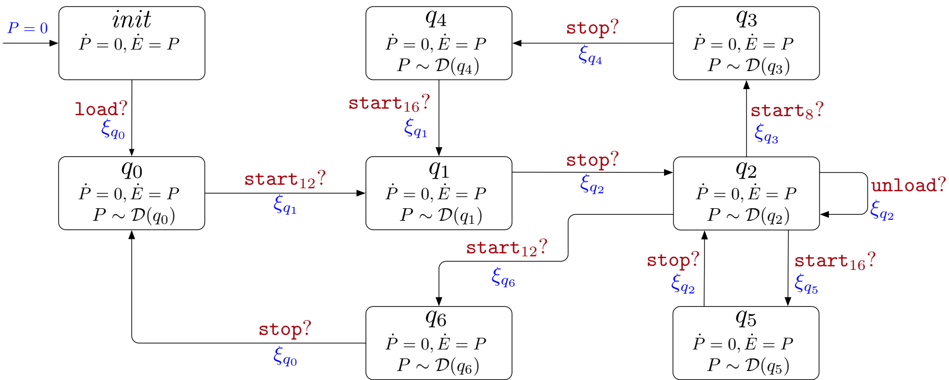


Figure 13.4: *Example of field data for part type* i. *Events are marked with triangles: green for load, yellow for start, blue for stop, and red for unload. The top plot highlights the time window during each task and the corresponding task numerical identifier.*

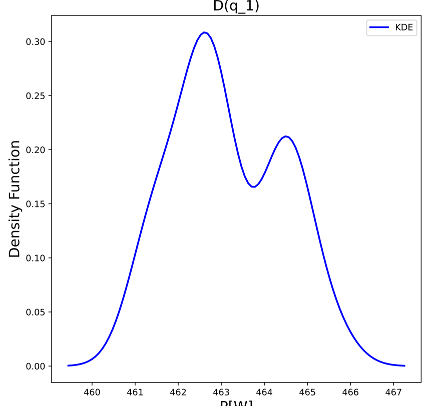
the learned SHA (reported in Table 13.3, Table 13.4, and Table 13.5) indicate how the SHA complexity varies between different experiments.

In all experiments, $\mathsf{L}^*_{\mathrm{SHA}}$ running time (i.e., the computation time required by the algorithm to perform the learning) is increasing in the volume of the training dataset and the complexity of the learned SHA.

We illustrate an example of learned SHA, specifically for part type i as obtained with $\mathsf{N}_{\mathsf{tr}} = 10$. Machining tasks constituting the part-program of part type i feature the spindle moving at three different nominal speeds (Fig. 13.4). Accordingly, the data processing of the speed signal captures three events $\texttt{start}_8$, $\texttt{start}_{12}$, and $\texttt{start}_{16}$ in addition to `stop`, `load` and `unload` events which are trivially captured for any trace. The learned SHA has seven locations in addition to the default *init* and accordingly to the estimated $\mathcal{D}(l)$ we can differentiate among locations q_1, q_3, q_5, and q_6 where the cutting process is in execution and the spindle requires not negligible



(a) *Learned SHA, color-coded as in Fig. 3.1.*



(b) *Example of P distribution estimated through KDE for location q_1.*

Figure 13.5: *Output of* $\mathsf{L}^*_{\mathrm{SHA}}$ *for Scenario A, Part type* i, $\mathsf{N}_{\mathsf{tr}} = 10$ *training traces.*

power, and locations q_0, q_2, and q_4 where the spindle stops and does not require any power. Locations q_0, q_2, and q_4 are reached after either `load`, `unload`, or `stop` events, as expected. Fig. 13.5b shows an example of KDE, specifically $\mathcal{D}(q_1)$.

Although there are 6 machining s, the learned SHA has 4 locations modeling active operational states. Indeed, operation 26 is repeated twice, and location q_1 captures both repetitions. Also, operation 16 is not considered statistically different from operation 26, thus $\mathsf{L}^*_{\mathrm{SHA}}$ groups them into a single location.

Furthermore, locations modeling operational states (q_1, q_3, q_5, and q_6) only have outgoing edges labeled with the `stop` event. Similarly, locations

modeling idle states (q_0, q_2, and q_4) only have outgoing edges with event labels $\texttt{start}_i$ with $i \in \{8, 12, 16\}$. As a consequence, upon simulating the behavior of the SHA in Fig. 13.5a, it is impossible to observe two consecutive stop events or two consecutive $\texttt{start}_i$ events as it never occurs in the input field data.

Table 13.3 and Table 13.4 report the number of locations and edges of the learned SHA in scenarios A and B, respectively. For each instance of scenarios A and B, varying the value of N_{tr} does not impact the number of locations and edges of the learned SHA. As a matter of fact, $\mathsf{L}^*_{\mathrm{SHA}}$ experiences the same sequence of events in scenario A as the produced workpieces belong to the same part type. Although scenario B includes three part types, at least one trace for each part type is included in the training dataset, and each training trace represents a known part type. Therefore, all learned SHA equally capture such traces irrespective of the value of N_{tr}.

Differently, the sequence of events changes for each training trace of scenario C capturing a possibly new part type being produced. As a consequence, the structure of the learned $\mathsf{L}^*_{\mathrm{SHA}}$ might vary significantly as the training dataset enlarges. Table 13.5 reports the number of locations and edges of the learned SHA for scenario C showing an increasing complexity.

The validation phase is performed with $\mathsf{N}_{\mathsf{val}} = 30$ validation traces in scenario A (i.e., on average $11.1\mathrm{h}$ of acquisition per instance) and with $\mathsf{N}_{\mathsf{val}} = 90$ validation traces (30 traces for each involved part type) in scenario B (i.e., on average $36.6\mathrm{h}$ of acquisition per instance). SHA learned in scenario C are validated against $\mathsf{N}_{\mathsf{val}} = 103$ traces collected over the course of three weeks of acquisition.

Each instance of scenario A includes a single part type, so any trace represents the same sequence of machining tasks. Similarly, scenario B includes three part types, but the part mix is assumed to be known. Therefore, the learned models in scenarios A and B are compatible with all traces in the validation dataset.

As a difference, never-before-seen part types are included in the validation dataset of scenario C. Therefore, each trace may capture a different part-program with a new sequence of tasks and/or never-before-seen tasks. As the training dataset increases in the number of traces, the probability of facing never-before-seen part types or tasks decreases. When a trace is not compatible with the learned automata, partial compatibility is checked by generating a controller that only mimics a *prefix* of the trace (in this case, containing no less than 5 events).

Numerical results referring to scenario C (instance ID=10) include $\mathsf{N}_{\mathsf{val}} =$

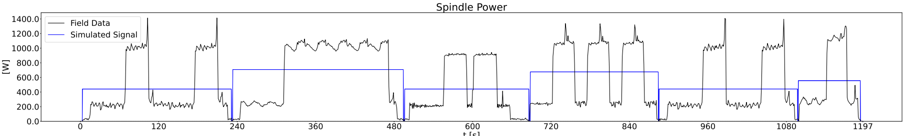


Figure 13.6: *Example of a sampled signal of the spindle power for part type i (also shown in the top plot in Fig. 13.4) and power signal generated through Uppaal by simulating the behavior of the learned SHA in Fig. 13.5a.*

Table 13.6: *Estimate errors broken down by instance ID and the number of traces N_{tr} used for $\mathsf{L}^*_{\mathrm{SHA}}$ learning.*

ID	N_{tr}	$\mathrm{MRE}_{\mathrm{est}}$	$\mathrm{MRE}_{\mathrm{BL}}$	ID	N_{tr}	$\mathrm{MRE}_{\mathrm{est}}$	$\mathrm{MRE}_{\mathrm{BL}}$
	3	0.918%	0.410%		3	1.240%	0.507%
1	10	0.592%	0.361%	6	10	1.198%	0.487%
	20	0.558%	0.349%		20	1.029%	0.167%
	3	0.720%	0.747%		3×3	4.482%	13.408%
2	10	0.691%	0.482%	7	10×3	4.255%	11.747%
	20	0.100%	0.109%		20×3	3.764%	9.370%
	3	0.566%	0.274%		3×3	3.292%	14.011%
3	10	0.475%	0.264%	8	10×3	2.839%	12.363%
	20	0.402%	0.246%		20×3	2.332%	12.060%
	3	1.393%	0.425%		3×3	3.787%	9.422%
4	10	1.041%	0.390%	9	10×3	3.304%	8.745%
	20	1.034%	0.388%		20×3	2.354%	8.478%
	3	1.319%	0.319%		5	37.67%	96.84%
5	10	1.208%	0.195%	10	7	26.75%	89.84%
	20	1.165%	0.120%		9	22.52%	84.05%

103 traces all different from each other and never-before-seen during training. Among them, 12, 27, and 32 are (at least) partially compatible with the automata learned with $\mathsf{N}_{\mathsf{val}} = 5, 7, 9$, respectively. The low compatibility ratio observed is motivated by the degree of customization of the involved part types, which yields a very diverse part mix. Nevertheless, the compatibility ratio is intrinsically increasing with the value of N_{tr}. Moreover, should a desired trace be found to be incompatible, it is possible to add the trace to the training dataset and to update the SHA by repeating the training.

Each learned SHA is paired with a controller SHA mimicking a compatible trace. Each SHA pair is imported into Uppaal to generate $\mathsf{N}_{\mathsf{val}}$ runs through trace-based simulation. As an example, Fig. 13.6 shows the original and the estimated power signals obtained by performing trace-driven simulation on the SHA learned for part i with $\mathsf{N}_{\mathsf{tr}} = 10$ (see Fig. 13.5a).

We exploit as a measure of accuracy for the learned SHA the mean relative error $\mathrm{MRE}_{\mathrm{est}}$ over all validation traces in estimating the energy consumed by trace $i = 1 \dots \mathsf{N}_{\mathsf{val}}$ under analysis, i.e., $E_{i,\mathrm{est}}$. Let us denote $E_{i,\mathrm{ref}}$ as the reference energy consumption of trace i computed from the spindle power signal. We define the relative percentage error $\mathrm{RE}_{i,\mathrm{est}}$ for validation trace $i = 1 \dots \mathsf{N}_{\mathsf{val}}$ as follows:

$$\mathrm{RE}_{i,\mathrm{est}} = \frac{|E_{i,\mathrm{ref}} - E_{i,\mathrm{est}}|}{E_{i,\mathrm{ref}}} \cdot 100.$$

Then, the mean relative percentage error $\mathrm{MRE}_{\mathrm{est}}$ is computed over the $\mathsf{N}_{\mathsf{val}}$ traces (Table 13.6).

The proposed methodology is compared with a standard and simple *baseline* estimation of the energy. To this end, the machine's average power is estimated from the training traces (irrespective of speed variations). The baseline energy estimate $E_{i,\mathrm{BL}}$ is calculated by multiplying the duration of each validation trace and the estimated machine average power. Similarly for estimate $E_{i,\mathrm{est}}$, the relative percentage error $\mathrm{RE}_{i,\mathrm{BL}}$ and its mean $\mathrm{MRE}_{\mathrm{BL}}$ is computed (Table 13.6).

Obtained results in Table 13.6 show that the proposed method achieves good results in estimating machine energy consumption. Validation of scenario A and scenario B yields respectively up to $\mathrm{MRE}_{\mathrm{est}} = 1.393\%$ and $\mathrm{MRE}_{\mathrm{est}} = 4.482\%$ in the worst case. In scenario C, all training traces are different, and the $\mathsf{L}^*_{\mathrm{SHA}}$ is fed with a wider variety of part types. Therefore, $\mathrm{MRE}_{\mathrm{est}}$ is high, suggesting more extended training needs.

Obtained results in Table 13.6 show that errors decrease as the number of training traces increases, as expected. The improvement is more evident for ID=10 (from 37.67% to 22.52% with four additional traces), where each additional training trace might carry important information about new part programs.

As for the comparison against the baseline, the proposed method achieves significantly better results in scenarios B and C, while in scenario A, the results are similar. In scenario A, the learned SHA is less accurate than the benchmark (i.e., $\mathrm{MRE}_{\mathrm{est}} > \mathrm{MRE}_{\mathrm{BL}}$ holds) in 16 cases out of 18 due to fitting errors which do not come into play when calculating the benchmark. Indeed, since a single part type is produced, the potential of $\mathsf{L}^*_{\mathrm{SHA}}$in capturing machine flexibility is not exploited.

As for scenario B, the learned SHA can capture the differences among the three produced part types using the spindle speed signals. On the contrary, the baseline estimate considers an average part type unable to distinguish between tasks. The learned SHA is more accurate in all instances (up

to 30% of relative gap among $\mathrm{MRE}_{\mathrm{est}}$ and $\mathrm{MRE}_{\mathrm{BL}}$). In scenario C, the advantage of using the learned SHA compared to the baseline is even more evident.

CHAPTER 14

Human Behavior Model Adjustment

Human-robot interactive applications are a paradigmatic example of the learning technique described in Chapter 12, constituting the model adjustment phase of the development framework (see Section 4.4).

Field observations cover physiological aspects, specifically human fatigue, and physical elements from which events related to human behavior are inferred (e.g., the human's location within the room). Collected data are fed to $\mathsf{L}^*_{\text{SHA}}$ *to learn a SHA modeling human behavior, which is thus always up-to-date with the available observations. The learned SHA modeling human behavior is then plugged into the SHA network replacing the initial draft model to perform the SMC experiments. Application developers then proceed with the robotic mission planning or reconfiguration process in light of the learning and SMC results.*

This chapter introduces how raw EMG signals are processed before being fed to $\mathsf{L}^*_{\text{SHA}}$ *and how the learned SHA capturing human behavior is extended to be compliant with the SHA network presented in Chapter 6.*

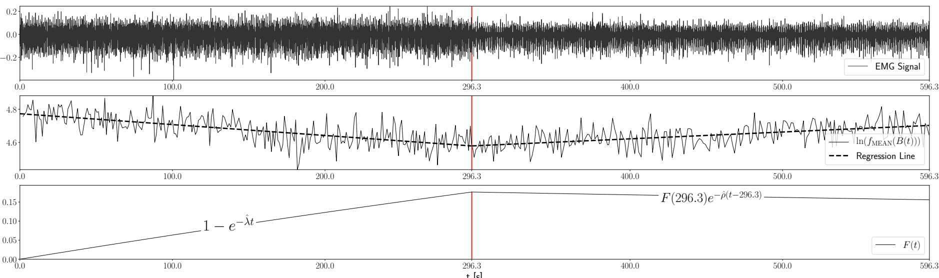


Figure 14.1: *Plots representing the three processing stages of a sample EMG signal from [110]. The upper plot contains an example of the raw EMG signal of a subject walking up to* $t = 296.3s$ *and then standing for* $5\mathrm{min}$. *The signal was collected through electrodes attached to the subject's left tibialis anterior. The middle plot displays the corresponding* $\ln(f_{\mathrm{MEAN}}(t))$ *(solid line) and regression line (dashed line) with slope* m. *The bottom plot shows the estimated fatigue curve with* $\hat{\lambda} = -m$ *up to time* 296.3 *and* $\hat{\rho} = m$ *afterwards. Note that the plotted fatigue curves are* portions *of exponential curves with slow dynamics (thus, the apparent linearity).*

14.1 Data-Driven Fatigue Estimation

As explained in Section 12.2, flow conditions characterizing each location of the learned SHA feature a stochastic parameter (as is the case in models of human behavior described in Section 6.3). Such parameter is calculated differently depending on the nature of the physical variable through function `est_param` while performing the `ht` queries introduced in Section 12.2.2.

In the specific application to human-robot interaction, human fatigue is the involved physical variable, and fatigue (resp. recovery) rate λ (resp. ρ) from Eq.6.5 are the random parameters whose calculation, starting from the raw sampled signal, is required and illustrated in the following. This signal processing procedure is exploited when running `ht` queries to process raw signals and estimate the parameter values whose distribution has to be identified.

Fatigue—i.e., a muscle's performance decay—is not *directly* measured. Previous studies document how it can be estimated starting from variations in muscle fibers' electrophysiological properties [139, 156]. The input to the estimation procedure is the electromyography (EMG) signal (an example of which is shown in Fig. 14.1), recorded through non-invasive electrodes placed on the skin.

Spectral variations of the EMG signal are frequently proposed as fatigue indexes, most notably in the work by Lindström et al. [139]. The authors

discuss how the mean spectral frequency (MNF) is used to estimate the fatigue and recovery rates (λ and ρ in Eq.6.5). To calculate the MNF[1], first, it is necessary to identify all activation windows of the EMG signal.

Previous studies show how muscle activation is periodic, and this *intermittency* manifests itself through **bursts** of the signals [40]. Given a time instant t, the corresponding burst j—starting at time $t_{s,j}$ and ending at time $t_{e,j}$ and such that $t_{s,j} \leq t \leq t_{e,j}$ holds—is determined by function $B : \mathbb{R}_+ \rightarrow (\mathbb{R}_+ \times \mathbb{R}_+)$ using the Teager-Kaiser energy operator [199]. The MNF $f_{\text{MEAN}}(B(t))$ for burst $B(t) = \langle t_{s,j}, t_{e,j} \rangle$ is defined in Eq.14.1: f_i and P_i are the frequency value and EMG signal power spectrum at frequency bin i, and $n = \mathsf{sr} \cdot (t_{e,j} - t_{s,j})$ is the number of frequency bins, which depends on burst $B(t)$'s length and sampling rate sr.

$$f_{\text{MEAN}}(B(t)) = \frac{\sum_{i=1}^{n} f_i P_i}{\sum_{i=1}^{n} P_i} \tag{14.1}$$

To estimate the rates, it is necessary to calculate $\ln(f_{\text{MEAN}}(B(t)))$ for every point of the EMG signal and calculate a regression line. The middle plot in Fig. 14.1 shows the logarithm of the MNF (solid line) obtained from the EMG signal in the upper plot and its corresponding regression line (the dashed line). The module of the regression line's slope m is an estimate for rates λ and ρ [156], as per Eq.14.2. The MNF decreases when the muscle is strained and increases during recovery, as in Fig. 14.1 (middle plot).

$$\begin{aligned} m < 0 &\longrightarrow \hat{\lambda} = -m \\ m \geq 0 &\longrightarrow \hat{\rho} = m \end{aligned} \tag{14.2}$$

The bottom plot in Fig. 14.1 shows the estimated fatigue curve computed by replacing in Eq.6.5 parameters λ, ρ with estimates $\hat{\lambda}, \hat{\rho}$: fatigue increases while the subject is walking and starts decreasing when the subject stops (time $296.3s$).

The hereby presented procedure to calculate $\hat{\lambda}$ and $\hat{\rho}$ constitutes the implementation of the function est_param invoked by Algorithm 4 (line 3) specific to our use case. Function est_param takes as input a segment of a signal collected from the SUL and returns either the $\hat{\lambda}$ or the $\hat{\rho}$ estimate calculated as previously explained. The $\mathsf{L}^*_{\text{SHA}}$ teacher, then, performs hypothesis testing on a collection of fatigue rates (all $\hat{\lambda}$ or $\hat{\rho}$ estimates from all EMG signals segments following a specific sequence of events) to identify their probability distribution.

[1] To perform EMG signal processing (including burst identification), we rely on library **biosignalsnotebooks** (https://biosignalsplux.com).

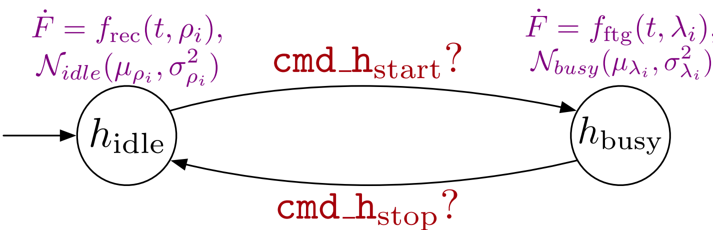


Figure 14.2: *Example of SHA modeling human behavior resulting from* $\mathsf{L}^*_{\text{SHA}}$.

After multiple runs of the robotic application, by iterating this procedure for each run, it is possible to collect a set of estimated values for parameters λ and ρ. Fatigue and recovery rates are mutable to the extent that they can display slight variations across subjects and different trials of the same subject [141]. It is also acknowledged that fatigue and recovery rates might be considered constant for a single subject over a limited time frame (in the range of minutes) but can vary significantly among different subjects depending on their physical conditions [141].

The model adjustment phase aims to reduce the model-to-reality gap and increase the accuracy of formal verification results. Therefore, formal verification experiments must consider the fatigue phenomenon's variability. To this end, as explained in Section 6.3, we do not model fatigue and recovery rates as scalars (e.g., the mean of all estimated values $\hat{\lambda}$ and $\hat{\rho}$) but as samples of a probability distribution.

The fatigue profile of a category of subjects is characterized by two normal distributions describing parameters λ and ρ for that specific category (e.g., *young* and *sick* or *elderly* and *healthy*) and for each operating condition (e.g., *running* and *sitting*). Each run of the robotic application (real or simulated) adds to the traces available to the $\mathsf{L}^*_{\text{SHA}}$ teacher and is processed as described in this section to obtain the corresponding set of $\hat{\lambda}$ and $\hat{\rho}$ estimates. Suppose the collected set of estimates is not compatible with the Normal distributions that have already been identified (e.g., because a different class of subjects was involved in the application). In that case, a new one will be added through an ht query as per Algorithm 4 (line 12).

14.2 Integrating Learned SHA in the SHA Network

In this section, we explain how SHA modeling human behavior learned by $\mathsf{L}^*_{\text{SHA}}$ fit into our model-driven framework and, in particular, the formal model defined through a network of SHA. The automata network includes SHA modeling the robot, the orchestrator, the robot's battery, and

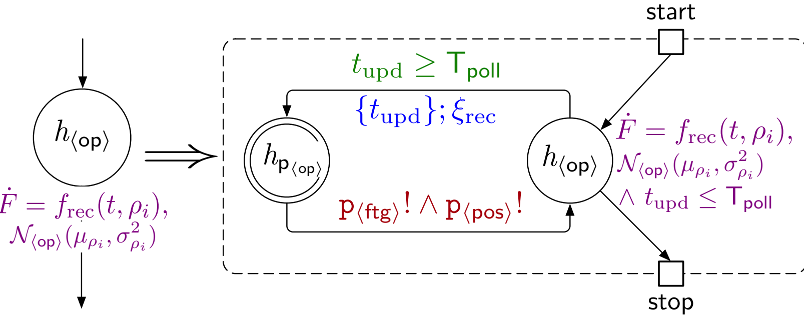


Figure 14.3: *Transformation pattern for a* non-critical operating condition. *Invariants are in purple and channels in red, as in Fig. 14.2, whereas guard conditions and update instructions are in green and blue, respectively. Separate instances of this pattern differ from each other because of the location label* $h_{\langle\text{op}\rangle}$.

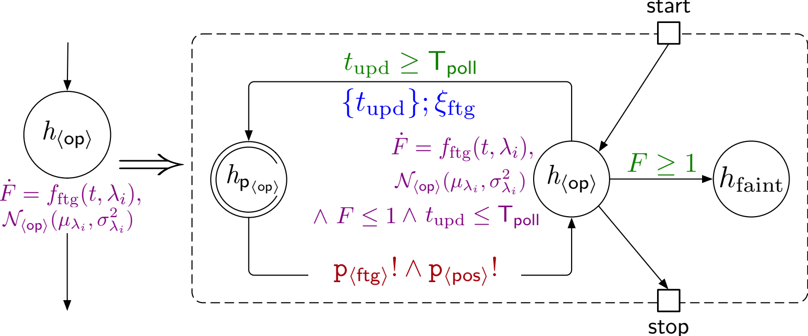


Figure 14.4: *Transformation pattern for a* critical operating condition. *Color-coding and features distinguishing two instances are the same as for Fig. 14.3. This pattern features an additional outgoing edge (and invariant) from* $h_{\langle\text{op}\rangle}$ *to deadlock location* h_{faint} *due to fatigue* $F \in W$ *reaching the critical threshold* 1.

the humans: the latter are progressively replaced by the automata learned by $\mathsf{L}^*_{\text{SHA}}$ based on field observations. To integrate learned SHA in the network, they need to be modified to make them homogeneous with the rest of the network. This adaptation depends on the modeling approach used in the model-driven framework. In the following, we introduce the modification rules required by our modeling approach.

The formal model has features capturing phenomena from the real world, which are required to make the model more realistic and, thus, obtain a more reliable outcome prediction. Specifically, these features are:

1. the $\langle\text{op}\rangle$_pub pattern, which captures the periodic publication of sensor data performed by robots and humans towards the orchestrator,

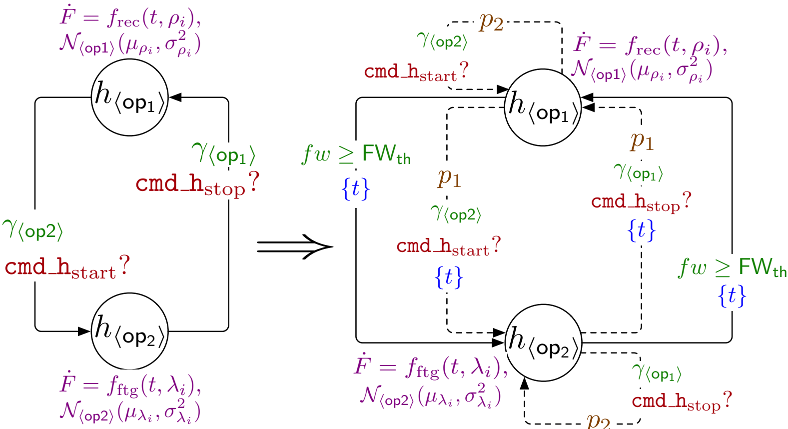


Figure 14.5: *Transformation pattern for an edge between two operating conditions $h_{\langle \text{op1} \rangle}$ and $h_{\langle \text{op2} \rangle}$. Weights p_1 and p_2 influence the automaton to take the edge or not when an event fires through channels* cmd_h$_{\text{start}}$ *(or* cmd_h$_{\text{stop}}$*) and guard $\gamma_{\langle \text{op2} \rangle}$ (or $\gamma_{\langle \text{op1} \rangle}$) is verified. Edges might also fire autonomously due to variable fw.*

which is necessary for the latter to have up-to-date information about the system (e.g., the current human fatigue level);

2. the possibility for the human to make decisions independently of the orchestrator's instructions capturing the unpredictability of human behavior, i.e., the Disobey/Obey and Free Will SHA add-ons introduced in detail in Chapter 8.

The parts of the formal model corresponding to these features are fixed for all SHA modeling humans and, therefore, do not need to be learned. Nevertheless, any SHA learned by L^*_{SHA} must be expanded with these features, using a fixed set of rules, to be fully compatible with the SHA network.

The rules distinguish locations based on the characteristics of their flow conditions, and in particular, based on whether the corresponding flow condition (an element of set M) causes physical variables in the system to reach a critical threshold. In our model-driven framework, the core physical variable is human fatigue, whose critical threshold (i.e., 1) is reachable when the human is moving, and fatigue is increasing (thus, evolving according to Eq.6.5(fatigue)).

In the following, we present the rules used to expand L^*_{SHA} outcomes, using as a running example the SHA in Fig. 14.2. In this case, there are only two locations capturing the human standing and walking (h_{idle} and h_{busy}, respectively). Each location is labeled with the identified flow con-

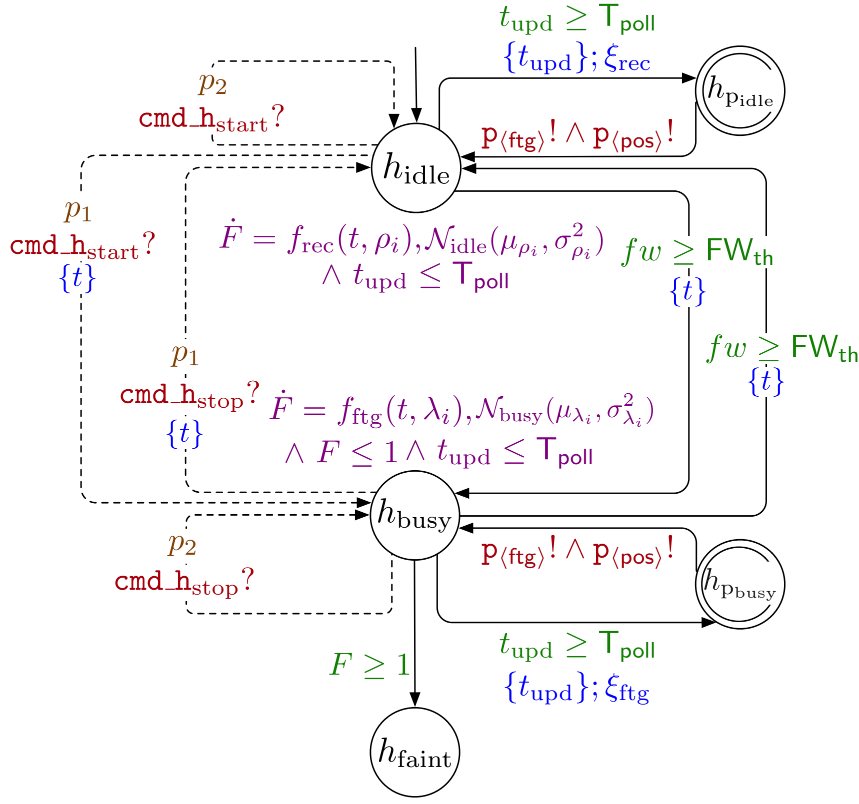


Figure 14.6: *SHA resulting from the application of the three patterns to the SHA in Fig. 14.2, specifically: h_{idle} is an operating condition, h_{busy} is a critical operating condition, and the edges connecting them are two free will edges. Color-coding is the same as in Fig. 14.3.*

dition, $f_{\text{rec}}(t, \rho_i)$ and $f_{\text{ftg}}(t, \lambda_i)$ (see Eq.6.8 and Eq.6.7) where ρ_i and λ_i are distributed according to $\mathcal{N}(\mu_{\rho_{idle}}, \sigma^2_{\rho_{idle}})$ and $\mathcal{N}(\mu_{\lambda_{busy}}, \sigma^2_{\lambda_{busy}})$, respectively.

The transition between the two locations is triggered by firing an event through channels cmd_h$_{\text{start}}$, cmd_h$_{\text{stop}}$$\in C$, instructing the human to start and stop walking, respectively. Note that the labeling function $\mathcal{L}_{\pi_\sigma}$ defined by $\mathsf{L}^*_{\text{SHA}}$does not distinguish between labels $(\gamma, \mathsf{c}!)$ and $(\gamma, \mathsf{c}?)$ (i.e., whether the event is sent or received through channel c): in our network since humans receive instructions from the orchestrator, learned edge labels are always finalized as $(\gamma, \mathsf{c}?)$.

In a learned SHA, we identify the following elements, which are then transformed according to the patterns in Fig. 14.3, Fig. 14.4, and Fig. 14.5:

a) non-critical operating conditions are locations with a flow condition that prevents the physical variable from reaching a critical threshold

(e.g., h_{idle} in Fig. 14.2);

b) critical operating conditions are locations in which the associated flow condition may cause the physical variable to reach a critical threshold (e.g., h_{busy} in Fig. 14.2);

c) edges between operating conditions (critical or not, e.g., both edges in Fig. 14.2).

The publication of sensor data is a feature that is introduced by suitably expanding the SHA around operating conditions (critical or not). The act of sharing sensor data, indeed, occurs *while* the agent is performing an action corresponding, in the formal model, to the automaton staying in a specific location (i.e., the operating condition).

The free will of humans, instead, is introduced by modifying the edges connecting operating conditions. Free will manifests itself when the human receives an orchestrator's instruction to switch from one behavior to another, which is modeled as an edge between operating conditions but can decide otherwise.

Finally, the model is expanded by allowing for the possibility that critical operating conditions can lead to a deadlock/error due to the physical variable exceeding a critical threshold. We remark that, although we defined the transformation rules to introduce features typical of applications involving human-robot interactions, they apply to any system with the same characteristics (i.e., periodic sensor data sharing, the existence of flow conditions that may lead to a critical deadlock situation, and the possibility of haphazard behavior). Nevertheless, when presenting the transformation patterns in detail in the following, some features (e.g., flow conditions $f_{\mathrm{rec}}(t, \rho_i)$ and $f_{\mathrm{ftg}}(t, \lambda_i)$) will be specific to the human model use case which is the core of this work.

Every operating condition $h_{\langle \mathsf{op} \rangle}$ is expanded according to the pattern made of the following elements, formalizing the publication of sensor data (fatigue and position) regarding the human (see Fig. 14.3):

1. invariant $t_{\mathrm{upd}} \leq \mathsf{T}_{\mathsf{poll}}$, which is added to location $h_{\langle \mathsf{op} \rangle}$ (e.g., h_{idle} in Fig. 14.2): clock $t_{\mathrm{upd}} \in X$ measures the time elapsed between two sensor readings and a new measurement is available every $\mathsf{T}_{\mathsf{poll}} \in K$ instants;

2. committed location $h_{\mathrm{p}_{\langle \mathsf{op} \rangle}}$: as per Uppaal convention, time cannot elapse in a committed location [123], thus the automaton is forced to take an outgoing edge immediately upon entering such locations;

3. the edge from $h_{\langle\mathsf{op}\rangle}$ to $h_{\mathrm{p}_{\langle\mathsf{op}\rangle}}$ that, due to the combination of invariant $t_{\mathrm{upd}} \leq \mathsf{T}_{\mathsf{poll}}$ and guard $t_{\mathrm{upd}} \geq \mathsf{T}_{\mathsf{poll}}$ fires as soon as $t_{\mathrm{upd}} = \mathsf{T}_{\mathsf{poll}}$ holds, resetting clock t_{upd} and executing instruction ξ_{rec} to calculate the new sensor readings;

4. the edge from $h_{\mathrm{p}_{\langle\mathsf{op}\rangle}}$ to $h_{\langle\mathsf{op}\rangle}$ that, due to the nature of committed locations, immediately fires an event over channels $\mathtt{p}_{\langle\mathsf{ftg}\rangle}$ and $\mathtt{p}_{\langle\mathsf{pos}\rangle}$ to publish the new sensor readings for fatigue and position, respectively.

As in Fig. 14.3, we depict transitions that connect a pair of locations, one inside the pattern and one outside it, as connected to *ports* start and stop, depending on their direction (incoming or outgoing, respectively); indeed, ports are not part of the SHA formalism but an expedient to visualize how the pattern is connected to the rest of the automaton.

The transformation pattern for critical operating conditions, shown in Fig. 14.4, includes elements formalizing that the automaton must enter a deadlock location when a real-valued variable reaches a critical threshold. In the case of SHA modeling a human, the physical variable constrained by flow conditions is human fatigue (modeled by real-valued variable $F \in W$) whose maximum value is 1. According to the fatigue model in Eq.6.5, fatigue can reach the critical threshold if it is increasing, thus, when it evolves according to flow condition $f_{\mathrm{ftg}}(t, \lambda_i)$. This pattern contains the same elements as the one for non-critical operating conditions (capturing the sending of human-related data), plus the following ones:

5. invariant $F \leq 1$ on $h_{\langle\mathsf{op}\rangle}$ to ensure that the automaton leaves the location after the critical threshold is reached;

6. deadlock location h_{faint}. Note that h_{faint} is an ordinary location and not an *operating* condition since, once the human reaches full exhaustion, the robotic mission immediately fails. It is no longer necessary to share sensor data. The pattern is not, therefore, applied recursively;

7. the edge from $h_{\langle\mathsf{op}\rangle}$ to h_{faint} with guard condition $F \geq 1$.

The third and final pattern, shown in Fig. 14.5, is applied to transitions connecting two operating conditions $h_{\langle\mathsf{op}_1\rangle}$ and $h_{\langle\mathsf{op}_2\rangle}$ (irrespective of whether they are critical or not) and captures the agent's *free will*. For illustrative purposes, in Fig. 14.5 $h_{\langle\mathsf{op}_1\rangle}$ is non-critical, and $h_{\langle\mathsf{op}_2\rangle}$ is critical, but all remarks in the following also hold in the other cases. The pattern includes the following elements:

1. a probabilistic transition from $h_{\langle\mathsf{op}_1\rangle}$ to $h_{\langle\mathsf{op}_2\rangle}$ guarded by condition $\gamma_{\langle\mathsf{op}_2\rangle}$, triggered by an event through channel $\mathtt{cmd_h}_{\mathrm{start}}$ and labeled

with probability weight p_1: when this edge fires, it captures the case in which the human decides to follow the orchestrator's instruction;

2. a probabilistic self-loop on $h_{\langle \mathsf{op}_1 \rangle}$ guarded by the same condition $\gamma_{\langle \mathsf{op}_2 \rangle}$, triggered by an event through the same channel $\texttt{cmd_h}_{\text{start}}$ but labeled with a different probability weight p_2: when this edge fires, it captures the case in which the human decides to ignore the orchestrator's instruction and remain in operating condition $h_{\langle \mathsf{op}_1 \rangle}$;

3. a transition from $h_{\langle \mathsf{op}_1 \rangle}$ to $h_{\langle \mathsf{op}_2 \rangle}$ that may autonomously fire when guard $fw \geq \mathsf{FW}_{\mathsf{max}}$ holds: variable $fw \in V_{\text{dc}}$ is periodically updated with a value uniformly drawn from a range $[0, \mathsf{FW}_{\mathsf{max}}]$ (where $\mathsf{FW}_{\mathsf{max}} \in K$). The autonomous decision can then be enacted if and only if the new value is such that $fw \geq \mathsf{FW}_{\mathsf{th}}$ holds, where $\mathsf{FW}_{\mathsf{th}} \in K$ is a constant parameter.

After any of the edges to $h_{\langle \mathsf{op}_2 \rangle}$ fires, clock t_{phase} (see Eq.6.8 and Eq.6.7) is reset and a new value of λ_i is generated from $\mathcal{N}(\mu_{\lambda_{\langle \mathsf{op}_2 \rangle}}, \sigma^2_{\lambda_{\langle \mathsf{op}_2 \rangle}})$. The edge from $h_{\langle \mathsf{op}_2 \rangle}$ to $h_{\langle \mathsf{op}_1 \rangle}$ (also in Fig. 14.5) encapsulates the same features. However, it is enabled by condition $\gamma_{\langle \mathsf{op}_1 \rangle}$, triggered by events on channel $\texttt{cmd_h}_{\text{stop}}$ and, when it fires, a new value of ρ_i is drawn from $\mathcal{N}(\mu_{\rho_{\langle \mathsf{op}_1 \rangle}}, \sigma^2_{\rho_{\langle \mathsf{op}_1 \rangle}})$.

The patterns described above are added to learned SHA via the following rules:

1. locations with flow condition f_{rec}, capturing recovery, are expanded with the pattern for non-critical operating conditions (when fatigue is decreasing, it cannot reach the critical threshold);

2. locations with flow condition f_{ftg}, capturing increasing fatigue, are expanded with the pattern for critical operating conditions (when fatigue increases, it may eventually reach the critical threshold);

3. edges connecting two operating conditions (critical or not) are expanded with the pattern capturing free will.

Fig. 14.6 shows the SHA resulting from the application of these three rules to the example of Fig. 14.2. The resulting automaton includes, in addition to locations h_{idle} and h_{busy}, location h_{faint} connected to h_{busy}. Locations h_{idle} and h_{busy} are connected by edges capturing the agent's free will triggered via channels $\texttt{cmd_h}_{\texttt{start}}$ and $\texttt{cmd_h}_{\texttt{stop}}$, respectively. Note that, like the original edges in Fig. 14.2, the edges capturing free will in Fig. 14.6 do not include a guard condition, meaning they are always enabled. The

SHA in Fig. 14.6 maintains the same physical behavior as the automaton learned from real observations and has the additional features necessary to be plugged into the formal model. The resulting SHA network with the updated model of human behavior can thus undergo formal verification to predict the scenario's outcome as explained in Chapter 4.

CHAPTER 15

Human Behavior Learning Validation

$\mathsf{L}^*_{\mathrm{SHA}}$ is validated on the human-robot interaction CPS exemplar first through the model-driven approach (i.e., with manually drafted reference models) and, secondly, through the simulation-driven approach with simulation data (see Chapter 13 for the description of the two approaches).

With the model-driven validation, $\mathsf{L}^*_{\mathrm{SHA}}$'s accuracy is assessed through manual inspection of the learned SHA (which is compared with the reference model of human behavior) and by estimating through two distinct Uppaal experiments the probability of success of the benchmark scenario with the learned SHA and the reference model.

With the simulation-driven approach, the success probability with the learned SHA is estimated through Uppaal and compared against the success rate observed at runtime.

15.1 Model-Driven Experiments

In this CPS exemplar, the entities subject to uncertainty are the **humans** interacting with the robot, of which we learn the temporal behavior of physical fatigue. The latter is impacted by human actions that involve moving

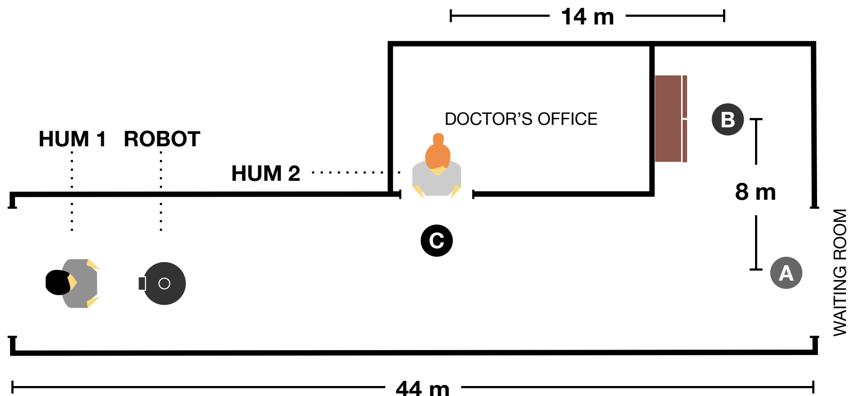


Figure 15.1: *Experimental setup of the benchmark robotic mission used to evaluate the SHA learned through $\mathsf{L}^*_{\text{SHA}}$. The humans and the robot are displayed in their starting positions. The figure also highlights the distance between points A, B, and C, i.e., the destination for the 3 services.*

(causing an increase in fatigue) and resting (causing a decrease in fatigue).

For the purpose of validation, we devise a benchmark human-robot interaction scenario from the application domain described in Chapter 4. Fig.

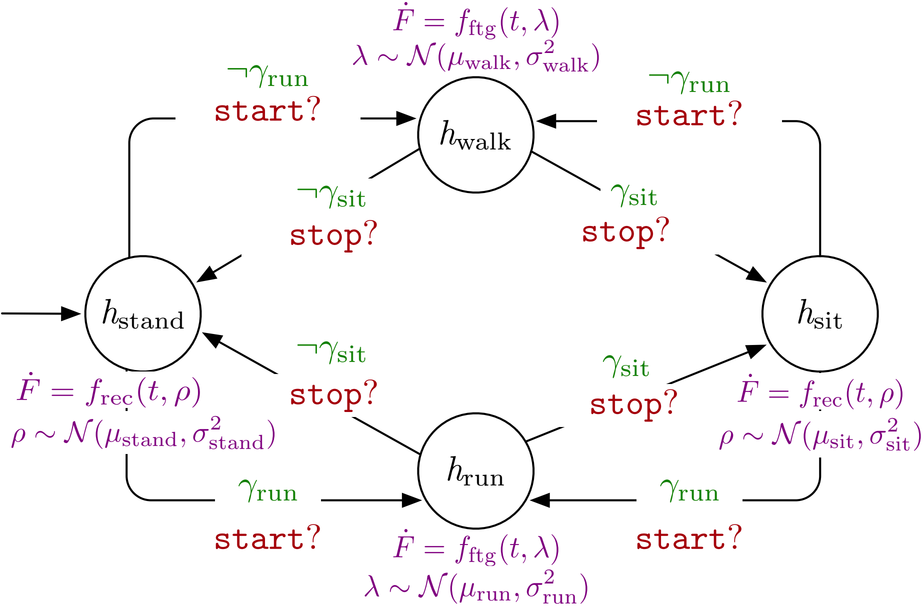


Figure 15.2: *Learned SHA for Exp.3. Color-coding is the same as in Fig. 3.1.*

15.1 represents the floor layout and the agents involved in the scenario: two humans—a patient in need of medication and a doctor—and a mobile robot. The patient (labeled as **HUM 1**) follows the robot to the waiting room (point A), where they can sit. The doctor (**HUM 2**) leads the robot to a cupboard (point B) to fetch the required medication. Finally, the robot returns to the waiting room to lead the patient to the doctor's office.

Each location of $\mathcal{A}_{\text{hyp}}$ features an element in set $M' \times \mathcal{Z}$ representing the dynamics of fatigue (i.e., either increasing or decreasing) and the distribution of the respective rate in a specific **behavioral state**. Subjects undergo alternate fatigue and recovery cycles, depending on whether they are actively performing a task (such as walking) or resting. In the corresponding SHA, fatigue is modeled through real-valued variable $F \in W$. The possible flow conditions constraining fatigue are known a priori and referred to as $f_{\text{ftg}}(t, \lambda)$ for fatigue cycles and $f_{\text{rec}}(t, \rho)$ for recovery cycles [118]. Parameters λ and ρ represent the rate at which fatigue increases or decreases, respectively, which are known to be normally distributed [140]. The mean and variance of such distributions vary based on the specific operational state and are subject to learning. Set $\mathcal{Z}$ contains samples of parameter ρ when the human is resting and parameter λ when the human is moving.

As explained in Section 12.2, $\mathsf{L}^*_{\text{SHA}}$ handles observable events. The fact that a subject starts or stops walking (without sitting) can be inferred by the position's sampled signal. If the human starts running, their position will vary at a higher speed, which can be recorded through an Indoor Positioning System (IPS) sensor. The building layout is known a priori as well as the location of seating areas, thus we can reasonably assume that, if a human stops close to a chair's location (which is also recorded by the IPS), they have sat down.

In this validation phase, the events (also in Table 15.1) that impact the subject's fatigue or recovery rate are "*walk*", "*run*", "*stand*", and "*sit*", each corresponding to a different behavioral state: walking, running, resting (while standing still), and sitting. When a subject runs, fatigue rate λ is higher than during a normal walk, and—similarly—sitting increases recovery rate ρ compared to standing. All events are observable and identifiable by labeling function $\mathcal{L}_{\pi_\sigma}$ (see Section 12.2.1).

We have defined 5 different SHA modeling human behavior to be learned by $\mathsf{L}^*_{\text{SHA}}$. These SHA are manually drafted in Uppaal and produce the traces fed to $\mathsf{L}^*_{\text{SHA}}$ when the Teacher requests it. All learned SHA are reported in [133] and Appendix A, while Fig. 15.2 shows the learned SHA for Exp. 3 featuring 4 locations—h_{stand}, h_{sit}, h_{walk}, and h_{run}—corresponding to as many behavioral states.

Table 15.1: *Comparison between the original SHA modeled in Uppaal (labeled as SUL) and the learned SHA (labeled as $\mathsf{L}^*_{\mathrm{SHA}}$) for the 5 experiments. Employed metrics are analogous to Table 13.1. The two SHA are further compared based on the probability of success of the benchmark mission ($\mathbb{P}_{\mathrm{M}}(\diamond_{\leq\tau}\mathsf{scs})$).*

Exp.		Events	$\lvert L \rvert$	$\lvert \mathcal{E} \rvert$	$\lvert M' \rvert$	$\lvert \mathcal{Z} \rvert$	$\mathbb{P}_{\mathrm{M}}(\diamond_{\leq\tau}\mathsf{scs})$
1	SUL	2	2	2	2	2	$[0.73, 0.83]$
	$\mathsf{L}^*_{\mathrm{SHA}}$		2	2	2	2	$[0.76, 0.86]$
2	SUL	4	3	4	2	3	$[0.83, 0.93]$
	$\mathsf{L}^*_{\mathrm{SHA}}$		3	4	2	3	$[0.82, 0.92]$
3	SUL	6	4	8	2	4	$[0.90, 0.99]$
	$\mathsf{L}^*_{\mathrm{SHA}}$		4	8	2	4	$[0.90, 1.00]$
4	SUL	6	4	6	2	4	$[0.90, 1.00]$
	$\mathsf{L}^*_{\mathrm{SHA}}$		4	7	2	4	$[0.90, 0.99]$
5	SUL	6	4	7	2	5	$[0.75, 0.85]$
	$\mathsf{L}^*_{\mathrm{SHA}}$		7	13	2	5	$[0.77, 0.87]$

Labels $\mathtt{start}, \mathtt{stop} \in C$ are channels marking that the human has started or stopped moving: in SHA modeling human behavior, all labels are followed by $?$ since the human always *receives* an instruction by the robot. In all learned SHA, events "*walk*" and "*run*" correspond to pairs $\langle \neg\gamma_{\mathrm{run}}, \mathtt{start} \rangle$ and $\langle \gamma_{\mathrm{run}}, \mathtt{start} \rangle$, while "*stand*" and "*sit*" correspond to pairs $\langle \neg\gamma_{\mathrm{sit}}, \mathtt{stop} \rangle$ and $\langle \gamma_{\mathrm{sit}}, \mathtt{stop} \rangle$.

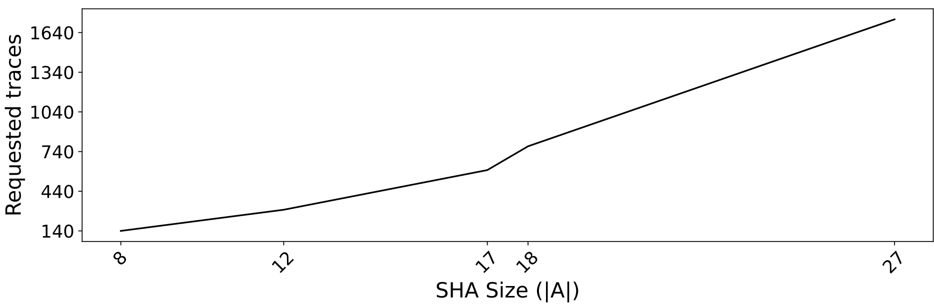
Table 15.1 reports metrics comparing the original SUL to the $\mathsf{L}^*_{\mathrm{SHA}}$ outcome. The comparison accounts for the same SHA features as in Table 13.1, indicating whether $\mathsf{L}^*_{\mathrm{SHA}}$ learns the SHA graph accurately. Exp. 4 and 5 report discrepancies. As for Exp. 4, in the original SHA, if the human is running, they switch to the same location whether they simply stand or they sit. The switch is captured through a single edge representing a disjunction between the conditions of events "*stand*" and "*sit*".

As per Section 12.2, in SHA learned by $\mathsf{L}^*_{\mathrm{SHA}}$, each edge is labeled by a *single* event (conjunctions and disjunctions are not explicitly learned). The edge with a disjunction in the original SHA is realized in the learned SHA by two edges, each equivalent to a disjunct. In Exp. 5, the original SHA features a source of inconsistency as described for Exp. $6, 9, 10$ in Section 13.2, which causes 3 out of the original 4 locations to be split into two locations. Therefore, a manual inspection of learned SHA highlights that they behave correspondingly to the reference models for all experiments.

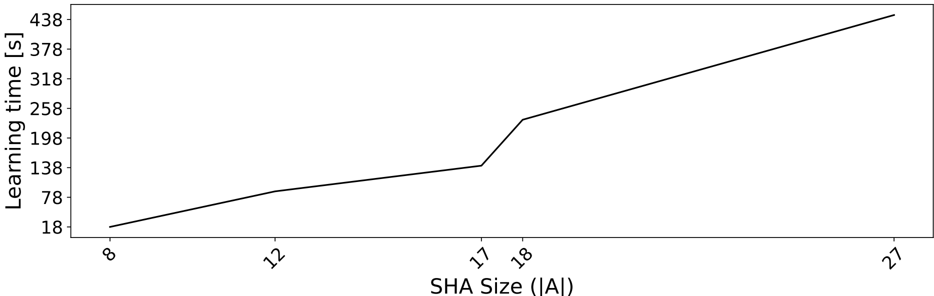
To determine whether the distributions learned through ht queries are

Table 15.2: *Comparison between the simulated human behavior (labeled as SUL) and the learned SHA (labeled as* $\mathsf{L}^*_{\mathrm{SHA}}$*) for the* 4 *simulation-driven experiments. Data concerning the size of the learned SHA and the probability of success have the same meaning as in Table 15.1.* $\mathrm{E}_{\leq\tau}[\mathsf{max}(F)]$ *is the expected maximum fatigue value (*$[\%]$*) calculated for each service requested by human subjects.*

Exp.		**Events**	$\lvert L\rvert$	$\lvert\mathcal{E}\rvert$	$\lvert M'\rvert$	$\lvert\mathcal{Z}\rvert$	$\mathrm{E}_{\leq\tau}[\mathsf{max}(F)]$ **HUM 1** to point A	$\mathrm{E}_{\leq\tau}[\mathsf{max}(F)]$ **HUM 2**	$\mathrm{E}_{\leq\tau}[\mathsf{max}(F)]$ **HUM 1** to point C	$\mathbb{P}_{\mathrm{M}}(\diamond_{\leq\tau}\mathsf{scs})$
1	SUL	2	–	–	–	–	0.336	0.0315	0.431	0.785
	$\mathsf{L}^*_{\mathrm{SHA}}$		2	2	2	2	0.369 ± 0.03	0.0344 ± 0.003	0.408 ± 0.08	$[0.751, 0.851]$
2	SUL	4	–	–	–	–	0.328	0.0308	0.338	0.928
	$\mathsf{L}^*_{\mathrm{SHA}}$		3	4	2	3	0.36 ± 0.04	0.0319 ± 0.004	0.303 ± 0.04	$[0.893, 0.992]$
3	SUL	6	–	–	–	–	0.324	0.127	0.262	0.944
	(a) $\mathsf{L}^*_{\mathrm{SHA}}$		3	4	2	3	0.311 ± 0.07	0.107 ± 0.04	0.389 ± 0.09	$[0.753, 0.853]$
	(b) $\mathsf{L}^*_{\mathrm{SHA}}$		4	6	2	4	0.337 ± 0.03	0.158 ± 0.06	0.278 ± 0.02	$[0.902, 1]$
4	SUL	14	–	–	–	–	0.522	0.1468	0.494	0.556
	$\mathsf{L}^*_{\mathrm{SHA}}$		10	24	2	10	0.515 ± 0.154	0.113 ± 0.08	0.482 ± 0.08	$[0.536, 0.636]$
	(bis) $\mathsf{L}^*_{\mathrm{SHA}}$		10	24	2	10	–	0.116 ± 0.08	0.281 ± 0.07	$[0.952, 1.0]$



(a) *Traces requested by $\mathsf{L}^*_{\mathrm{SHA}}$ to complete the learning for all model-driven experiments.*



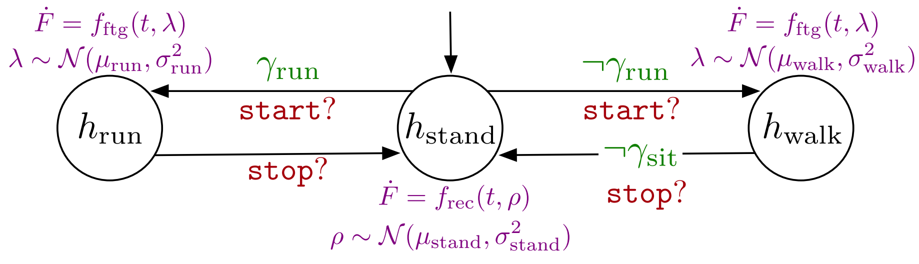
(b) *Time ([s]) requested by $\mathsf{L}^*_{\mathrm{SHA}}$ to complete the learning for all model-driven experiments.*

Figure 15.3: *Plots reporting on $\mathsf{L}^*_{\mathrm{SHA}}$ performance with increasing SUL complexity, measured through the learned SHA size ($|\mathcal{A}|$).*

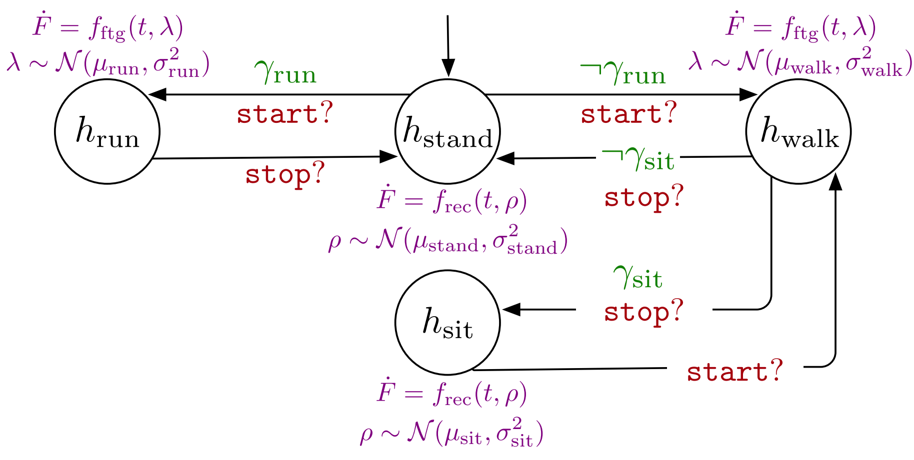
acceptable, we perform a Kolmogorov-Smirnov test on the normal distributions associated with the reference SHA locations and the empirical distributions identified by $\mathsf{L}^*_{\mathrm{SHA}}$ associated with the corresponding locations on the learned SHA. With $\alpha = 0.05$, in 94% of the cases, it cannot be refuted that the two sets are drawn from the same distribution. The percentage grows to 100% with $\alpha = 0.1$.

For each learned SHA, we estimate the success probability of the robotic mission, indicated as $\mathbb{P}_{\mathrm{M}}(\diamond_{\leq\tau}\mathsf{scs})$, where τ is the time-bound for the SMC experiment. The same success probability is then estimated with the original SHA. Table 15.1 shows the two sets of success probability ranges estimated through SMC with the same time-bound $\tau = 220$s. The resulting confidence intervals (CIs) for the success probability overlap in all 5 experiments. Although there is no ground to establish whether the CIs obtained with the reference and learned models are *equal*, the overlap guarantees that they are not incompatible.

Finally, metrics concerning $\mathsf{L}^*_{\mathrm{SHA}}$ performance are reported in Fig. 15.3



(a)



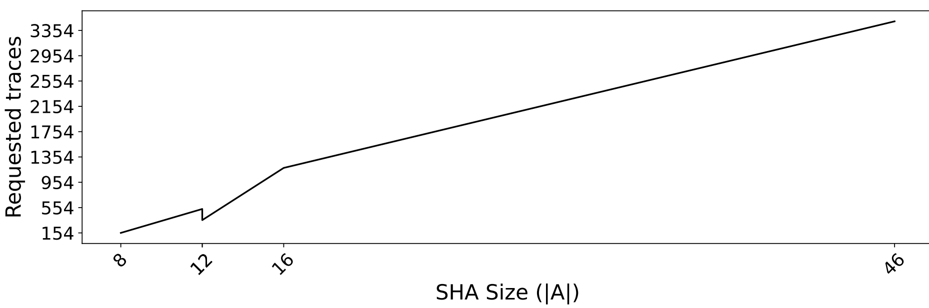
(b)

Figure 15.4: *SHA learned from simulation traces for Exp.3a (a) (with insufficient traces) and 3b (b) (with sufficient traces, leading to the correct result).*

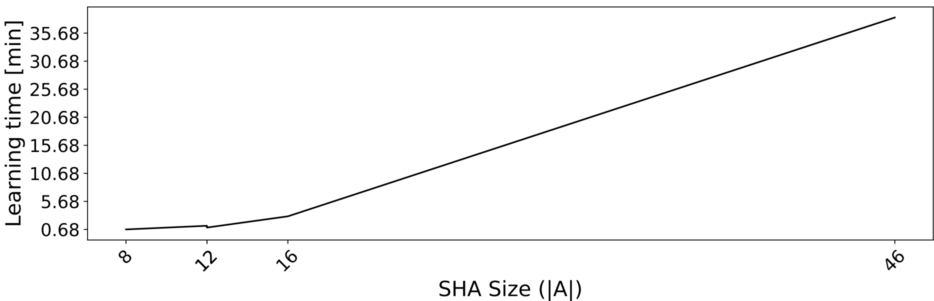
as a function of SHA size $|\mathcal{A}|$, which is computed as described in Section 13.2. Learning time ranges from approximately 18s to $7,5\text{min}$. The number of traces collected before termination ranges from 140 to 1740. Note that, for comparable SHA sizes and requested traces (both batches of experiments are performed with $n_{\text{min}} = 20$), the human-robot interaction CPS exemplar requires a longer learning time than CPS1 (see Section 13.2). The reason is that the Uppaal model from which traces are generated features a more complex SHA network than CPS1; thus, each round of knowledge refinement (i.e., the ref query) lasts longer.

15.2 Simulation-Driven Experiments

The model-driven validation approach provides evidence of the accuracy of $\mathsf{L}^*_{\text{SHA}}$ while learning known reference SHA. With the simulation-driven



(a) *Traces requested by $\mathsf{L}^*_{\mathrm{SHA}}$ to complete the learning for all simulation-driven experiments.*



(b) *Time ([min]) requested by $\mathsf{L}^*_{\mathrm{SHA}}$ to complete the learning for all simulation-driven experiments.*

Figure 15.5: *Plots reporting on $\mathsf{L}^*_{\mathrm{SHA}}$ performance with increasing SUL complexity, measured through the learned SHA size ($|\mathcal{A}|$).*

approach described in the following, we test the algorithm in a realistic setup without a reference SHA modeling human behavior. For this reason, $\mathsf{L}^*_{\mathrm{SHA}}$'s goal is to infer a model of human behavior from realistic sensor logs to analyze the application offline.

We exploit the deployment framework presented in Chapter 10 to repeatedly **simulate** the benchmark application in a realistic virtual environment. With each experiment, we enrich the range of actions ("events" in Table 15.2) the user can choose from to increase the SUL complexity: for instance, in Exp. 1, the only action that increases human fatigue is "*walk*", while Exp. 4 also features alternatives "*run*", "*carry a load*", "*walk/run in an uncomfortable environment*." The collected simulation traces constitute the pool for $\mathsf{L}^*_{\mathrm{SHA}}$ to draw from when the teacher performs a ref query.

Table 15.2 displays the results with the simulation-driven validation approach. All learned SHA are shown in Appendix A. Similar to Table 13.1 and Table 15.1, metrics concerning SHA size are reported for the learned SHA; however, they are not reported for the original one since, in this case,

it does not exist. Therefore, to asses how accurate the learning is with this validation approach, we perform formal verification experiments on the model learned with each experiment. Formal verification results are compared with metrics extracted from the real system's behavior observed at runtime.

Firstly, we evaluate whether human fatigue values are compatible with those observed in simulations; secondly, whether the success probability within a time-bound estimated through SMC is compatible with the observations. The expected maximum value of fatigue for a subject within time τ corresponds to the value of expression $\mathrm{E}_{\leq\tau}[\mathsf{max}(F)]$. For this metric, for each experiment, three values are reported (one for each service in the mission): for **HUM 1** while following the robot to point A, for **HUM 2**, and for **HUM 1** while following the robot to point C. For the learned SHA, the value is calculated in Uppaal, whereas for the SUL, it corresponds to the sample average extracted from traces.

Success probability is estimated through SMC for the learned SHA, whereas, for the SUL, it is calculated as the ratio between traces reporting success before time τ and all available traces.

During this validation phase, traces are generated through simulation, not Uppaal. Therefore, function $\mathsf{resample}(s, n_{\min})$ (see Algorithm 7) extracts new observations of trace s from the *available* pool of data but cannot trigger the generation of a new trace. If $\textsc{Teacher}.\Sigma_{\text{obs}}$ already contains the entire available pool of traces, resample returns the empty set (i.e., $\textsc{Teacher}.\Sigma_{\text{obs}}$ is not updated in subsequent learning rounds). When no new counterexample is available (i.e., the termination condition in Algorithm 9), $\mathsf{L}^*_{\mathrm{SHA}}$ terminates, returning a closed and consistent SHA that only captures the behaviors inferrable from the *available* data pool.

Exp. 3 is an example of such an occurrence. Therefore, it has been broken into two steps to provide an example of $\mathsf{L}^*_{\mathrm{SHA}}$'s result in case a ref query is necessary, but all available traces have been processed. The learned automaton for Exp. $3a$ is shown in Fig. 15.4a, while Fig. 15.4b shows the SHA learned with a larger pool of traces in Exp. $3b$. Exp. $3a$ returns a SHA that does not capture *all* possible behaviors of the CPS. Specifically, $\mathsf{L}^*_{\mathrm{SHA}}$ is not able to learn the h_{sit} location in Fig. 15.4b. Since the model does not capture the possibility that the human may sit and rest, the expected value of fatigue while walking back from the waiting room to the office (i.e., $\mathrm{E}_{\leq\tau}[\mathsf{max}(F)]$ for **HUM 1** to point C in Table 15.2) is higher. The mission success probability is lower than the values recorded at runtime. Running new simulations provides a larger pool of traces to $\mathsf{L}^*_{\mathrm{SHA}}$ to refine the SHA and obtain accurate results, as shown in the row labeled as $3b$ in Table

15.2. Specifically, $\mathsf{L}^*_{\mathrm{SHA}}$ runs with $n_{\min} = 20$ to balance the requirements of $\mathsf{L}^*_{\mathrm{SHA}}$ queries and the cost of executing the benchmark scenario: a run of the application in the simulation environment lasts approximately $4\mathrm{min}$ and, in a physical setting, would involve two subjects, potentially in critical conditions.

The maximum fatigue expected value and success probability are estimated and compared with the SUL behavior as a reference. The average estimation error for the maximum fatigue value for all subjects across all experiments is 9%, and the sample means of fatigue calculated from SUL traces falls within the estimated range in 11 cases out of 12 (i.e., 3 human subject for 4 experiments). Given that fatigue is highly subject to variability, small oscillations and estimation errors are acceptable as long as they do not impact the success probability. The success rate within $\tau = 220\mathrm{s}$ recorded at runtime falls within the estimated range for all 4 experiments.

This validation phase provides an example of how the learning phase impacts the robotic mission design, as described in Chapter 4. At the end of Exp. 2, $\mathsf{L}^*_{\mathrm{SHA}}$ returns a SHA with the additional behavioral state modeling the case in which the subject is *sitting* (i.e., recovering at a higher pace). Formal analysis with the new model results in a higher success rate within the same time bound compared to Exp. 1 (CI $[0.893, 0.992]$ compared to $[0.751, 0.851]$). Similar conclusions can be drawn about Exp. $3b$: $\mathsf{L}^*_{\mathrm{SHA}}$ learns the *running* behavioral state, causing the second subject to reach—on average—a higher fatigue value (CI 0.158 ± 0.03 compared to 0.0319 ± 0.004), but their service is completed in less time, leading to a slightly higher success rate compared to Exp. 2 within the same time bound (CI $[0.902, 1]$ compared to $[0.893, 0.992]$).

The results of Exp. 4 reverse this trend. In this experiment, simulations feature a much broader spectrum of human actions, significantly boosting the variability of fatigue rates. This setup yields a lower estimated success rate at design time—within the range $[0.536, 0.636]$—which would urge the analyst to redesign the robotic mission. A possible reconfiguration consists of a **simplification** of the benchmark scenario: the patient *follows* the robot directly to the doctor's office, and, subsequently, the robot can serve the doctor and follow them to the cupboard's location. This re-planning solution exposes the patient to a single relocation instead of two, reducing their overall cumulative fatigue. Repeating the SMC experiment with the updated mission design and the same time bound as Exp. 4 (indicated as Exp. $4bis$ in Table 15.2) leads to a success probability higher than 95%.

Part IV

Overall Framework Validation

CHAPTER *16*

Model-Driven Framework Validation

This chapter presents the results obtained while validating the overall framework.[a] *Firstly, the accuracy of the design-time analysis phase is assessed through three case studies by comparing the formal verification outcomes with field data. Estimation errors for the physical variables and scenario outcomes are all smaller than 10%.*

For the second phase, realistic service robotic scenarios are mined from the literature to assess whether they could be analyzed through the framework, showing a coverage percentage of 80%. Three scenarios are then presented that synthesize common features from real-life examples and put through design-time analysis and deployment to assess the framework's efficacy with more complex missions.

Finally, the added value of the model adjustment phase is assessed. Specifically, the three scenarios from the second phase are re-assessed with a refined model of human behavior obtained through automata learning. The accuracy gain ranges from 2.4% in the worst case to 30% in the best case.

[a] The content presented in this chapter also partially appears in [129] and [131]. The author of this thesis declares to have also authored the reproduced text, figures, and data and to have the right to reproduce such content in a dissertation according to the license under which both articles are published.

16.1 Validation Goals

The validation process addresses the following questions:

G1. Is the formal model presented in Chapter 6 adherent to the physical robotic system, and how accurate are SMC results?

G2. Is the model-driven framework described in Chapter 4 practical and useful while developing interactive scenarios with multiple subjects and services? More specifically, we evaluate:

(a) how the DSL supports designers in configuring complex scenarios;

(b) how the design-time analysis phase provides reliable and valuable insights into the modeled scenario;

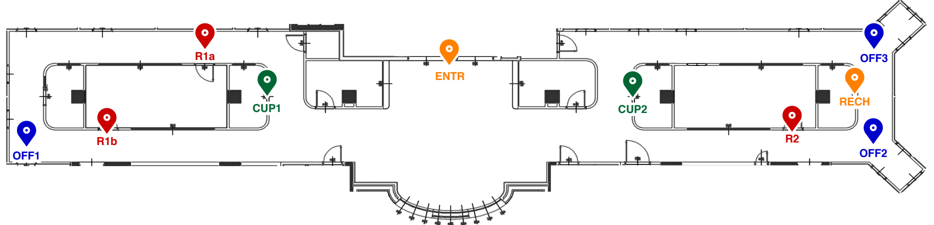
(c) how scenarios can be reconfigured to improve key indicators (the probability of success and estimated fatigue level of human subjects).

G3. What is the impact of the model adjustment phase, thus of learning a refined model of human behavior, on design-time analysis results?

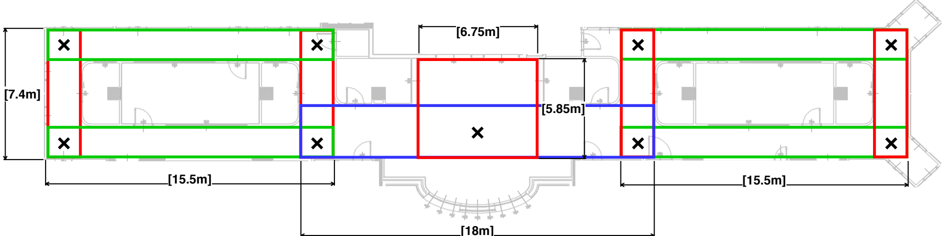
Both validation phases have been carried out following the framework workflow in Fig. 4.1. Firstly, we model the scenarios through the DSL described in Section 5.2. The case studies feature one mobile robot providing services (in compliance with the patterns presented in Section 5.1) to one or multiple humans. Agents operate within the floor layout represented in Fig. 16.1, corresponding to the third floor of Building 22 of Politecnico di Milano. Specifically, Fig. 9.1 highlights the POIs, while Fig. 9.2 shows how the real layout is abstracted as a set of rectangular areas as described in Section 5.2.1. While the physical layout where the robotic device moves is a university building, its areas are repurposed in the simulation environment to reproduce a healthcare setting. The layout features a main entrance ENTR where the robot meets patients to assist them and two side aisles, each with cupboards containing medical kits (CUP1 and CUP2), two rooms serving either as waiting rooms for the patients or examination rooms where doctors administer medications (R1a, R1b, and R2), and doctors' offices (OFF1, OFF2, and OFF3).

DSL models[1] are automatically converted into a JSON file and finally into a Uppaal model implementing the SHA network described in Chapter

[1] The DSL sources are available at https://github.com/LesLivia/hri_dsl.

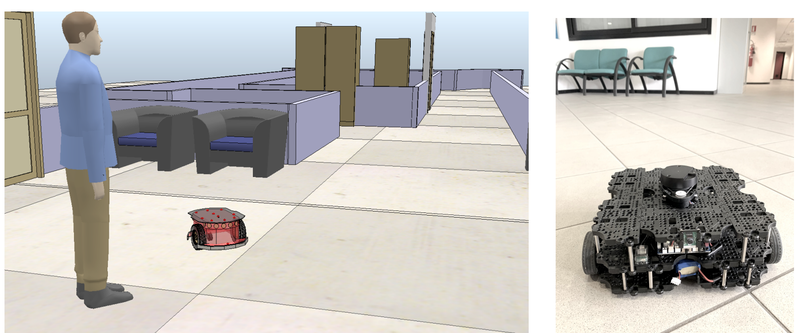


(a) *Points of interest within the experimental setting: the entrance and recharge station are in orange, cupboards are in green (CUP1 and CUP2), examination/waiting room entrances are in red (R1a, R1b, and R2), and doctors' offices in blue (OFF1, OFF2, and OFF3).*



(b) *Representation of the layout abstraction as a set of rectangular areas, with wall sizes ([m]) and intersection points between areas marked by $\times$ symbols.*

Figure 16.1: *Layout used for the experimental validation.*



(a) *Portion of the simulation scene with the simulated human and the* phantom *robot replicating the real robot's behavior.*

(b) *Real TurtleBot3 operating in the physical environment and reacting to the human in the simulation scene.*

Figure 16.2: *Hybrid real/simulated deployment environment.*

6 modeling the specific scenario.[2] The framework also automatically sets up and runs the SMC experiment. For the case studies discussed in this

[2]The tool to generate the Uppaal model is available at https://github.com/LesLivia/hri_designtime.

section, we perform SMC through Uppaal v.4.1.26 on a machine running macOS v.10.15.7 with 4 cores and 8GB of RAM.

All case studies are subsequently deployed as described in Chapter 10 [128].[3] We have adopted the digital-twin deployment pattern [161] (see Fig. 16.2) with a real mobile robot operating in the physical environment (shown in Fig. 16.2b) and reacting to virtual human subjects in the simulation scene (of which a portion is shown in Fig. 16.2a). The hybrid deployment environment allows us to verify the adherence of the robotic system's model and the orchestrator's efficacy with a real device while also performing several runs with (virtual) subjects exhibiting critical fatigue profiles. Electromyography signals serving as the dataset to simulate fatigue curves in the simulation environment are provided by [110]. The mobile device is a TurtleBot3 Waffle Pi.[4] Scenarios are deployed using V-REP v.3.6.2 for the simulation scene, Python v.3.6.9 for the orchestrator script, and ROS Melodic to communicate with virtual agents and the TurtleBot3 [176]. The deployment software tools run on a single machine running Ubuntu v.18.04 with 2 cores and 4GB of RAM.

16.2 G1: Formal Model Validation

The purpose of the hereby presented experiments is to assess the accuracy of the formal model presented in Chapter 6 and of the SMC results (validation goal **G1**). To this end, we perform the design-time analysis as presented in Chapter 5 on three scenarios, referred to as HF ("Human Follower"), HL ("Human Leader") and LB ("Low Battery"), all taking place in the experimental setup in Fig. 16.1 and described in detail in Table 16.1. The three scenarios are structured to validate the main features of the formal model: human-robot dyadic leader/follower dynamics, human fatigue model, robot's battery model, and orchestrator policies. Therefore, the three experiments test the formal model with both critical (low battery in LB and high fatigue in HL) and non-critical elements (high battery in HF/HL and low fatigue in HF/LB).

We perform SMC with decreasing values of time-bound τ to estimate the success probability of the three scenarios (i.e., query with QueryType P_SCS). This corresponds to the value of expression $\mathbb{P}_{\mathrm{M}}(\diamond_{\leq\tau}\ \mathsf{scs})$, where M is the SHA network modeling each mission. SMC experiments are performed in Uppaal with default statistical parameters, thus with $\epsilon = \alpha = 0.05$. We also estimate the maximum human fatigue value (expression

[3] The tool implementing the deployment approach is available at https://github.com/LesLivia/hri_deployment
[4] Full documentation is available at https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/.

Table 16.1: *Scenarios used for the formal model validation phase (abbreviation and detailed description). For each service, we indicate the starting location of the human and the target location as* START $\rightarrow$ TARGET.

SCENARIO	DESCRIPTION	MISSION
HF	The robot (Tbot) *leads* the human (H1) from OFF1 to CUP1. The robot is sufficiently charged to complete the mission, and the human exhibits a non-critical fatigue profile (Young/Healthy).	H1 Follower OFF1 $\rightarrow$ CUP1
HL	The robot (Tbot) *follows* the human (H1) from OFF1 to CUP1. The robot is sufficiently charged to complete the mission, and the human exhibits a critical fatigue profile (Elderly/Sick).	H1 Leader OFF1 $\rightarrow$ CUP1
LB	The robot (Tbot) *leads* the human (H1) from OFF1 to CUP1. The robot gets fully discharged during the mission, while the human exhibits a non-critical fatigue profile (Young/Healthy).	H1 Follower OFF1 $\rightarrow$ CUP1

Algorithm 10 Estimation of the success probability CI for a set of deployment traces $\mathcal{DT}$.

Input: $\mathcal{DT}$, τ, $\mathsf{N_h}$, $\mathsf{T_{int}}$, α
1: $\mathcal{DT}_{\mathrm{scs}} \leftarrow \emptyset$
2: **for** $dt \in \mathcal{DT}$ **do**
3: $\quad \mathcal{SVD}_{dt} \leftarrow \{t | t \in \mathbb{N} \wedge i \in [1, \mathsf{N_h}] \wedge \mathrm{h}_{i,\mathrm{svd}} \in dt(t)\}$ ▷ Timestamps corresponding to the end of a service.
4: $\quad$ **if** $|\mathcal{SVD}| = \mathsf{N_h} \wedge \mathsf{max}(\mathcal{SVD}) \leq \tau - \mathsf{T_{int}}$ **then**
5: $\quad\quad \mathcal{DT}_{\mathrm{scs}} \leftarrow \mathcal{DT}_{\mathrm{scs}} \cup \{dt\}$ ▷ All humans have been served within $\tau - \mathsf{T_{int}}$.
6: $p_l \leftarrow \mathsf{ppf}(\alpha/2, |\mathcal{DT}_{\mathrm{scs}}|, |\mathcal{DT}| - |\mathcal{DT}_{\mathrm{scs}}| + 1)$
7: $p_u \leftarrow \mathsf{ppf}(1 - \alpha/2, |\mathcal{DT}_{\mathrm{scs}}| + 1, |\mathcal{DT}| - |\mathcal{DT}_{\mathrm{scs}}|)$
8: $\epsilon \leftarrow (p_u - p_l)/2$
9: $p \leftarrow p_l + \epsilon$
Output: p, ϵ

$\mathrm{E}_{M,\tau}[\mathsf{max}(\mathrm{f})]$), and the robot's residual charge at the end of the mission (expression $\mathrm{E}_{M,\tau}[\mathsf{min}(\mathrm{b}_{\mathrm{chg},\%})]$), i.e., queries with QueryType E_FTG and E_CHG, respectively.

Subsequently, we deploy the three scenarios in the digital-twin setting (see Fig. 16.2) to collect runtime observations, compute the same metrics,

and compare the results with those obtained at design time. To this end, we apply a partial replication of SMC (summarized by Algorithm 10) to the traces collected through deployment to estimate the success probability range observed at runtime. We refer to the simulation log and sensor logs collected during a single run (described in Chapter 10) as **deployment trace**. Given deployment trace dt, we indicate as $dt(t)$ the set of data (sensor readings and milestones recorded by the orchestrator, if any) logged at time $t \in \mathbb{N}$. Since the orchestrator records the timestamp at which each human is served, it is possible to infer from a deployment trace dt whether the mission ended successfully. If human $i \in [1, \mathsf{N_h}]$ has been served in trace dt, there exists $t \in \mathbb{N}$ such that $\mathrm{h}_{i,\mathrm{svd}} \in dt(t)$ holds. We indicate as $\mathcal{DT}$ the set of all deployment traces collected for a given scenario. Set $\mathcal{SVD}_{dt} = \{t | t \in \mathbb{N} \wedge i \in [1, \mathsf{N_h}] \wedge \mathrm{h}_{i,\mathrm{svd}} \in dt(t)\}$ at Line 3 in Algorithm 10 contains the timestamps corresponding to the completion of a service in a specific trace dt. Similar to SMC, given a time-bound τ and the set of collected deployment traces $\mathcal{DT}$, for each deployment trace $dt \in \mathcal{DT}$, we check whether the mission has ended with success within τ (i.e., whether $\diamond_{\leq \tau}$ scs holds for dt). Algorithm 10 checks through the condition on Line 4 whether set $\mathcal{SVD}_{dt}$ has $\mathsf{N_h}$ elements (i.e., *all* humans have been served), and the maximum of $\mathcal{SVD}_{dt}$ is smaller than $\tau - \mathsf{T_{int}}$ (i.e., the *last* human to be served has been served within the time bound minus the time required by the orchestrator to process the information). If the condition on Line 4 is verified, trace dt constitutes a success and is added to set $\mathcal{DT}_{\mathrm{scs}}$ by the instruction on Line 5.

Algorithm 10 computes $\mathbb{P}_{\mathcal{DT}}(\diamond_{\leq \tau}\ \mathsf{scs})$ in terms of a confidence interval of the form $p \pm \epsilon$ with confidence level $1 - \alpha$. We adopt the Clopper-Pearson approach for binomial distributions to compute the CI, as it is also exploited by the Uppaal tool. Specifically, $p_l = p - \epsilon$ can be calculated as the α-quantile of a Beta distribution with parameters $\mathsf{successes}$ and $\mathsf{trials} - \mathsf{successes} + 1$ (Line 6), while $p_u = p + \epsilon$ can be calculated as the γ-quantile with $\gamma = 1 - \alpha$ and parameters $\mathsf{successes} + 1$ and $\mathsf{trials} - \mathsf{successes}$ (Line 7) [194].[5] Unlike point estimator $\mathsf{successes}/\mathsf{trials}$, this procedure also provides an insight into the variability of the success rate (i.e., the value of ϵ) and how its value changes as more runs are performed.

The results of the SMC experiments, the time and runs necessary to complete it with the required level of confidence, and the fatigue and charge estimations are reported in Table 16.2 (marked as DT, "Design Time"). The

[5] The Python implementation exploits the scipy.stats.distributions.beta.ppf function (referred to as ppf in Algorithm 10) from the SciPy library to calculate the required quantiles. Full documentation available at: https://docs.scipy.org.

Table 16.2: *Comparison between the results obtained through SMC at design time (DT) and the results obtained by deploying the three model validation scenarios (DEPL). For decreasing values of time-bound τ ([s]), the table contains the verification time ([min]), the probability CI estimated through Uppaal, the CI observed at runtime, and the runs necessary to compute such estimations. The table also contains the estimated maximum human fatigue values ([0 − 1]) and minimum charge levels ([%]). Configurations leading to the least accurate results for each metric are highlighted in grey.*

SC.	τ	Ver.Time	Success Probability		Runs		Max. Fatigue		Min. Charge	
		[min]	DT	DEPL	DT	DEPL	DT	DEPL	DT	DEPL
HF	75	0.41	0.952 ± 0.05	0.950 ± 0.05	29	110	0.0208 ± 0.004	0.0211 ± 0.007	95.54%	95.75%
	53	3.42	0.859 ± 0.05	0.865 ± 0.06	199	110	0.0177 ± 0.005	0.0179 ± 0.008	96.86%	96.27%
	50	3.23	0.812 ± 0.05	0.800 ± 0.07	250	110	0.0167 ± 0.003	0.0168 ± 0.008	97.05%	96.51%
	40	4.25	0.433 ± 0.05	0.500 ± 0.10	395	110	0.0125 ± 0.004	0.0125 ± 0.008	97.65%	97.11%
	34	2.19	0.239 ± 0.05	0.252 ± 0.08	296	110	0.0112 ± 0.003	0.0114 ± 0.003	98.01%	98.10%
HL	50	0.31	0.952 ± 0.05	0.950 ± 0.05	29	120	0.2015 ± 0.037	0.1939 ± 0.050	73.71%	73.32%
	42	2.72	0.826 ± 0.05	0.840 ± 0.07	236	120	0.1763 ± 0.038	0.1623 ± 0.051	74.19%	74.17%
	38	2.46	0.676 ± 0.05	0.664 ± 0.09	354	120	0.1599 ± 0.014	0.1577 ± 0.025	74.43%	74.38%
	35	3.94	0.409 ± 0.05	0.395 ± 0.09	389	120	0.1545 ± 0.042	0.1561 ± 0.027	74.61%	74.55%
	33	2.13	0.216 ± 0.05	0.208 ± 0.07	277	120	0.1451 ± 0.045	0.1434 ± 0.025	74.74%	74.91%
LB	150	1.02	0.000 ± 0.05	0.000 ± 0.05	29	107	–	–	0.001%	−0.001%

success probability ranges estimated for scenarios HF, HL, and LB through Algorithm 10 are reported in Table 16.2 (marked as DEPL, "Deployment").

Results in Table 16.2 corroborate the intuition that, for decreasing values of τ, the probability of success decreases both at design time and during deployment. Experimental results with the largest difference between design time and deployment estimations are highlighted in grey. We select values of parameter τ to be displayed in Table 16.2 corresponding to probability ranges in three macro-intervals: *high* success probability (i.e., with $p > 80\%$), *average* success probability (i.e., with $40\% < p < 70\%$), and *low* success probability (i.e., with $p < 25\%$). Scenarios HF and HL require $75\mathrm{s}$ and $50\mathrm{s}$, respectively, to end successfully with probability approximately equal to 1. At the same time, it drops to approximately 20% when the analysis is bounded to $34\mathrm{s}$ and $33\mathrm{s}$.

The variability of the success probability between runs within $34\mathrm{s}$ (for HF) or $33\mathrm{s}$ (for HL) and those requiring up to $75\mathrm{s}$ is due to the human stopping haphazardly during the mission as described in Section 6.3, causing a delay in the completion of the mission. As shown in Table 16.2, the configurations with the largest difference between design-time and runtime estimations of the success probability (highlighted in grey) are also the ones requiring the largest number of traces for the SMC experiment (395 and 389 compared to the 110 and 120 performed in the physical setting). This is due to how $\mathbb{P}_{\mathrm{M}}(\diamond_{\leq\tau}\ \mathsf{scs})$ and $\mathbb{P}_{\mathcal{DT}}(\diamond_{\leq\tau}\ \mathsf{scs})$ are calculated: if traces that have been generated or collected are consistent with each other (e.g., they are all successful), it takes a smaller set of traces to obtain a certain value of ϵ than when the number of successes fluctuates. Indeed, the estimated success probabilities resulting from the configurations requiring the largest number of runs (395 for HF and 389 for HL) are also the closest to 50%. In the worst case, the values of p estimated at design-time and runtime differ by 6.7% (for HF) and 1.4% (for HL) while this drops to 0.2% in the best case.

Data collected through the three scenarios are also necessary to assess whether the formal model accurately captures the robot's battery voltage drop (i.e., the battery discharging while the robot is operative). To estimate the expected value of the minimum charge for the same decreasing values of τ used for the success probability ranges, we calculate the value of expression $\mathrm{E}_{M,\tau}[\mathsf{min}(\mathrm{b}_{\mathrm{chg},\%})]$ in Uppaal (column DT) and the average of minimum values logged for each deployment trace $\mathrm{E}_{\mathcal{DT},\tau}[\mathsf{min}(\mathrm{b}_{\mathrm{chg},\%})]$ (column DEPL). Table 16.2 reports the battery charge percentage estimations at time τ for the three scenarios, highlighting, as in previous cases, the configurations leading to the largest estimation error. We recall that

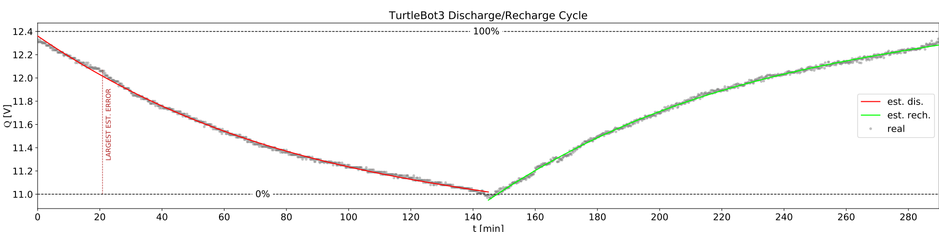


Figure 16.3: *Graph representing the battery voltage (Q [V]) evolution during a complete TurtleBot3 discharge/recharge cycle (approximately 140min each). Dots represent real voltage sensor readings. The red and green lines are the fitted discharge and recharge curves. Black dashed lines mark the voltage values corresponding to 100% (about 12.4V) of the charge and 0% (11.0V). The red dashed line marks the time instant where the design-time estimation is the least accurate.*

the differential equations obtained by deriving Eq.6.3 and Eq.6.4 constrain the battery voltage ([V]), whose value in the real system is directly measured through a sensor. The percentages shown in Table 16.2 are calculated according to Eq.16.1, where $\mathrm{b_{chg}}$ represents the dense counter presented in Section 6.2.2 (for column DT) or the value shared by the real sensor (for column DEPL), $\mathsf{C_{fail}}$ is the lowest voltage value allowed by the device (as presented in Section 6.2.2), and $\mathsf{C_{full}}$ is the (approximate) voltage value when the battery is fully charged.

$$\mathrm{b_{chg,\%}} = \frac{\mathrm{b_{chg}} - \mathsf{C_{fail}}}{\mathsf{C_{full}} - \mathsf{C_{fail}}} \cdot 100 \tag{16.1}$$

For the three scenarios, we estimate the residual battery charge, assuming that $\mathsf{C_0}$ is set to 99%, 75%, 0.8% for HF, HL, and LB, respectively. The largest estimation error is 0.61% for HF and 0.53% for HL. Unsurprisingly, charge-related estimations are more accurate than human fatigue, which has a higher degree of variability and is subject to the human's unforeseeable choices. Nevertheless, the time span required for the scenarios is orders of magnitude shorter than a full battery discharge cycle (approximately 2.5h). Therefore, while these estimations provide insights into how accurate the model is in the range of seconds, we have also assessed its accuracy in the longer run.

We have recorded the real battery sensor readings over the course of three full discharge/recharge cycles. This set of data has been used to fit parameters in Eq.6.3 and Eq.6.4 governing the evolution of real-valued variable Q in the SHA. Fig. 16.3 shows the voltage curves (modeled in SHA through real-valued variable Q) resulting from the fitting (red and green lines), compared against the actual sensors readings of a fourth complete

discharge/recharge cycle (grey dots). Sensor readings used to fit the curve parameters and those shown in Fig. 16.3 are different data sets. The largest estimation difference (also highlighted in Fig. 16.3) is 0.54%, comparable to the previously described differences obtained with the three scenarios. The fitted model must be evaluated assuming that experiments were carried out in a layout that is relatively free of obstacles and does not require complex trajectories (i.e., the layout mostly consists of straight corridors). Therefore, the impact of turns or obstacle avoidance in this situation is negligible, and the model shows good accuracy without considering these factors. However, should the framework be deployed in a more critical setting, different parameters or a more complex discharge cycle model should be employed to keep the same degree of accuracy.

Scenario LB requires a separate analysis. This experiment aims to assess whether the formal model accurately captures reality in a boundary condition where the robot's charge is insufficient to complete the mission (as previously mentioned, in this case, C_0 is 0.8%). When the mission starts, since the charge level is too low ($c \leq \mathsf{C}_{\mathsf{low}}$ holds), the orchestrator immediately instructs the robot to start moving towards the recharge station, which, however, requires about $3.5\mathsf{min}$ to be reached. In contrast, the robot has only $2.5\mathsf{min}$ of battery life left. The design-time analysis correctly predicts this outcome as the mission has 0% success probability within 150s (see Table 16.2), and all the collected deployment traces end in failure.

No fatigue estimation is provided in this case since the human never starts moving. As a matter of fact, the robot needs to recharge as soon as the mission starts, thus the orchestrator immediately enters submachine $\mathsf{r}_{\mathsf{rech}}$ (see Fig. 6.14) without sending any instructions to the human. As per Formula 6.15, failure occurs when voltage drops sufficiently close to 0%, which is why the estimation reported in Table 16.2 is not exactly 0%. However, the estimated probability of failure is still 1. On the other hand, the negative percentage estimated from deployment traces is due to how the real device works. As soon as the detected battery voltage equals 11V, the device will start emitting an acoustic signal to notify the need to recharge, it will beep for a few seconds and then stop sending power to the motors (thus, no motion is possible). From the moment it starts beeping to the moment it stops moving, the voltage drops slightly *below* 11V, leading to the negative percentage (see Eq.16.1).

Concerning the estimation of human fatigue, Table 16.2 reports the maximum value estimated through Uppaal (column DT) and from deployment traces (column DEPL). To estimate the maximum fatigue expected value at design time, we compute the value of expression $\mathrm{E}_{M,\tau}[\mathsf{max}(\mathrm{f})]$ in Uppaal

(Table 16.2, column DT), whose result is a 95% confidence interval of the requested value [46]. The same procedure is applied to the set of deployment traces $\mathcal{DT}$. For each deployment trace, we calculate the maximum value of fatigue of the human subject within time bound τ. The so-obtained values constitute the set of independent samples, of which we subsequently calculate the 95% confidence level (results are reported in Table 16.2, column DEPL).

In scenario HF, the human has the least critical fatigue profile (Young/ Healthy), thus it only reaches a fatigue value of approximately 2%. For this scenario, in the worst case, the largest design-time estimation error (calculated as the difference between the average value at design time and observed during deployment) is 1.75%. In HL, the human reaches higher fatigue values (up to approximately 20%). All intervals calculated from deployment traces fall within the range estimated at design time. In the worst case, also highlighted in Table 16.2, the estimation error is 8.6%. Although design-time estimations pertaining to fatigue are promising, it is important to remark that the simulator scripts governing human behavior during deployment directly result from the model-to-code transformation (presented in [128]) of the SHA described in Section 6.3. Therefore, these results do not constitute conclusive empirical proof that SHA modeling humans accurately capture reality. Nevertheless, since simulated sensors share their readings with the orchestrator over actual ROS topics, the limited design time-to-deployment errors indicate that modeling patterns dealing with readings update and publishing (i.e., the pattern in Fig. 6.4 and the RosPubNode pattern presented in [128]) are reliable.

Concerning performance and scalability, given the limited complexity of the scenarios analyzed in this batch of SMC experiments, verification experiments performed through Uppaal last between 0.31 and 4.25min. For the deployment phase, we have performed a total of 337 real runs (110 for HF, 120 for HL, 107 for LB). By factoring in the time required to perform each run, reset the layout to its starting configuration at the end of each run, and the time required to recharge the robot, this corresponds to approximately 64h of non-stop deployment and runtime data collection. Considering that, in a real healthcare facility, employees have multiple tasks to deal with, robots are not actively deployed 100% of the time, and there are time breaks between shifts, so the data collection phase would take longer.

Table 16.3: *Scenarios used for the framework validation phase (abbreviation, detailed description, and sequence of services). For each service, we indicate the starting location of the human and the target location as* START → TARGET.

SCENARIO	DESCRIPTION	MISSION
DPa	The robot (Tbot) serves a patient-doctor pair (P1/D1, respectively). The robot meets the patient by the entrance (ENTR) and *leads* them to the waiting room (R1b) to wait for the doctor to visit them. The robot *follows* the doctor to CUP1 where they fetch required tools, and *follows* them back (carrying the tools) to the examination room (R2) where the patient will receive the treatment. Finally, the robot returns to R1b and *escorts* the patient to R2, where the doctor is waiting.	P1 Follower ENTR → R1b, D1 Leader R2 → CUP1, D1 Leader CUP1 → R2, P1 Follower Rb1 → R2
DPb	The robot (Tbot) serves a patient-doctor pair (P1/D1, respectively). The robot meets the patient by the entrance (ENTR) and *leads* them to the waiting room (R1a) to wait for the doctor to visit them. The robot approaches CUP2 to retrieve a required medical kit, and then *delivers* it to D1 at OFF2. The robot *follows* the doctor to the examination room (R2) where the patient will receive the treatment. Finally, the robot returns to R1a and *escorts* the patient to R2 to be treated.	P1 Follower ENTR → R1a, D1 Recipient OFF2 ↔ CUP2, D1 Leader OFF2 → R2, P1 Follower R1a → R2
DPc	The robot (Tbot) serves two patient-doctor pairs (P1 and P2 are patients, D1 and D2 are doctors). The robot meets P1 by the entrance (ENTR) and *leads* them to the waiting room (R1a), then it performs the same task for P2 *leading* them from the entrance to R1b. The robot fetches the first required medical kit from CUP1 and *delivers* it to D1 at OFF1. The robot then serves D2 by *following* them to CUP2 and back to their office (OFF3) while carrying the kit. Finally, the robot *leads* P1 to OFF1 and P2 to OFF3 as both doctors are ready to visit them.	P1 Follower ENTR → R1a, P2 Follower ENTR → R1b, D1 Recipient OFF1 ↔ CUP1, D2 Leader OFF3 → CUP2, D2 Leader CUP2 → OFF3, P1 Follower R1a → OFF1, P2 Follower R1b → OFF3

16.3 G2: DSL and Design-Time Analysis Validation

After the analysis of the accuracy of the formal model, we focus on the overall efficacy of the model-driven framework (goal **G2**).

16.3.1 Real Case Studies Coverage Analysis

To this end, we have analyzed 41 real-world scenarios describing service robotic applications extracted from [12, 14, 18, 70, 99, 155]. We remark that given that service robot deployment is not widespread at the time of writing, there is no structured repository collecting natural language specifications of such scenarios; to the best of the authors' knowledge, the RoboMAX repository (providing 14 of the 41 identified scenarios) is the first attempt in this direction [12]. Therefore, the scenario collection phase has been performed manually by surveying related works in the literature and commercial service robots' documentation.

Within the set of eligible scenarios, we consider 14 scenarios to fall outside of the scope of our framework. As our work targets service robot applications featuring mobile robots that *interact* with humans, we consider out-of-scope all scenarios featuring robots that operate autonomously (e.g., performing patrolling or automated room cleaning), are teleoperated or provide information without expecting any reaction on the human side (e.g., periodical medication reminders).

As for the 27 scenarios within the framework's scope, we assess that 24 can be modeled through our framework, leading to a coverage percentage of 88.8%. The three scenarios that our framework does not cover feature: a) *exoskeletons*, which are considered service robots by standard ISO 13482 [103] (thus, they are in-scope with respect to our framework) but require different software development practices than mobile robots that our framework targets; b) *cognitive* interaction (e.g., comforting children or rehabilitating cognitive skills of patients recovering from strokes), whereas our framework targets *physical coordination* between humans and robots.

16.3.2 DTA of Realistic Healthcare Scenarios

To address goal **G2**, we have developed three more sophisticated scenarios, referred to as DPa ("Doctor-Patient"), DPb, and DPc, collecting the most frequent elements of the 24 real-world scenarios found in the literature (fetch-and-delivery tasks, doctor-patients dynamics, patient greeting, and transporting items). The three scenarios are described in detail in Table 16.3.

In all three scenarios, the robot has to serve one (in DPa and DPb) or two (DPc) pairs of human subjects representing a doctor and a patient. The robot always accompanies the patient to the waiting room (R1a, R1b, or R2) first (adhering, thus, to the Follower pattern), and then supports the doctor in retrieving the instrumentation needed to treat the patient. Doctors are either Leaders (D1 in DPa and DPb, and D2 in DPc) or Recipients (D1 in DPc) depending on whether they personally lead the robot to the destination, or it moves independently and then delivers the resource. For this experimental phase, the robot's charge is always sufficient (the opposite boundary condition has already been investigated with scenario LB), and patients exhibit more critical fatigue profiles than doctors.

As per Fig. 4.1, the entry point to the design-time phase analysis is the specification of the scenarios through the DSL presented in Section 5.2. The complete DSL file for the three scenarios is reported in Appendix B. All scenarios are set in the same layout (shown in Fig. 16.1), thus, there is a single definition.

Agents participating in the three scenarios are fixed; specifically, the DSL features one robot definition (identified as Tbot) and eight human definitions (P1 and D1 for DPa, P1 and D1 for DPb, and P1, D1, P2, and D2 for DPc). We recall that the maximum velocity and acceleration for the robot are directly derived from its type parameter, which, in this case, is turtlebot3_wafflepi.

Each scenario in Table 16.3 corresponds to a robotic *mission*, thus, there are three mission definition blocks defining the sequence of services that the robot must provide to complete the mission with success.

Finally, queries are defined to compute the metrics required to carry out this design-time analysis, i.e., the probability of success for decreasing values of τ, estimated fatigue for all human subjects, and residual battery charge. Parameter R (the bound on runs) is set to auto, to indicate that Uppaal should generate as many runs as necessary to compute estimations with the requested confidence level. As per Fig. 5.2, for each mission (thus, in our case, DPa, DPb, DPc), a JSON file is automatically generated and converted into a pair of Uppaal model/query files to perform verification.

We assess the "efficiency" of the DSL in terms of effort saved compared to manually drafting the SHA network modeling each scenario. To this end, we calculate the ratio (indicated as DSL2SHA in Table 16.4) between the size of a DSL instance and the size of the corresponding SHA network. We compute the size of a DSL model as the number of words needed to configure the scenario. Counting words rather than abstract elements captured by the DSL gives us a more accurate indication of the DSL's verbosity: note

that, since the declaration of each element in Section 5.2 requires at least one word, counting abstract elements would result in more favorable ratios. Given a SHA $\mathcal{A}$, we compute its size, indicated as $|\mathcal{A}|$ according to Eq.16.2:

$$|\mathcal{A}| = |\mathcal{E}| + |\Gamma(W)| + |C_{!?}| + |\Xi(W)| + |L| + |\mathcal{D}| + |\mathcal{F}| + |W| \quad (16.2)$$

The size of a network of SHA equals the sum of the sizes of all the SHA that compose it. Table 16.4 reports the resulting ratios.

SMC results are reported in Table 16.4 and discussed in the following. For this validation phase, the duration of verification experiments performed through Uppaal ranges from 16.36min to 175.68min in the worst case (i.e., scenario DPc, which has the most complex robotic mission and highest τ values). Unlike the previous phase, in this case, the goal is to test the framework's efficacy when developing realistic scenarios. Therefore, we do not collect a large batch of deployment traces to keep the duration of the deployment phase more practical (i.e., shorter than 1h). Given the smaller number of deployment traces that have been collected, the probability of success of the deployed system is not calculated through Algorithm 10, as it would yield scarcely significant CIs.

In this case, we adopt *point estimator* $\overline{p}$ given by the percentage of successful runs as specified by Eq.16.3, where $\mathcal{DT}$ is the set of deployment traces and set $\mathcal{SVD}_{dt}$ is calculated from a deployment trace $dt \in \mathcal{DT}$ as in Algorithm 10, Line 3.

$$\overline{p} = \frac{|\{dt | dt \in \mathcal{DT} \wedge |\mathcal{SVD}_{dt}| = \mathsf{N_h} \wedge \max(\mathcal{SVD}_{dt}) \leq \tau - \mathsf{T_{int}}\}|}{|\mathcal{DT}|} \cdot 100 \quad (16.3)$$

Metrics related to fatigue (for each human subject) and battery charge are computed as in the previous validation phase.

Through the design-time analysis, we estimate that the three scenarios require a τ of approximately 7min, 9min, and 25min, respectively, to end in success with probability greater than 90%. As previously mentioned, the robot's charge is not critical for any of the scenarios. Although DPc is the most demanding in terms of the robot's power since the initial charge $\mathsf{C_0}$ is 99%, the estimated residual charge at the end of the mission is greater than 60%. Doctors (i.e., agents D1 and D2) all adhere to the Elderly/Healthy fatigue profile, thus they do not constitute a criticality to the mission. As per Table 16.4, they reach an estimated maximum fatigue level between 2.18% and 3.7%, in particular, D1 in DPc (who participates in the Recipient pattern) reaches the lowest fatigue value (0.6%) as they only move haphazardly out of free will while waiting for the robot to deliver the resource (see Section 6.3.3). On the other hand, as expected, patients reach

Table 16.4: *Results of the DSL2SHA calculation, of the design-time analysis (DT) and of the deployment phase (DEPL) for scenarios DPa, DPb, and DPc. For decreasing values of τ ([s]), the table contains the verification time ([min]), the success probability CI estimated through Uppaal, the mean success rate observed at runtime, the estimated maximum fatigue value for all humans, and the estimated minimum charge value for the robot. Fatigue and charge level are estimated for each scenario's maximum value of τ. Configurations leading to the least accurate results for each metric are highlighted in grey.*

SC.	DSL2SHA	τ	Ver.Time [min]	Success Probability DT	Success Probability DEPL ($\overline{p}$)	HUM.	Max. Fatigue DT	Max. Fatigue DEPL	ROB.	Min. Charge DT	Min. Charge DEPL
DPa	235/863 (27.2%)	400	16.36	0.933 ± 0.05	1.00	P1	0.2664 ± 0.014	0.2385	Tbot	82.4%	82.77%
		350	54.67	0.706 ± 0.05	0.60	D1	0.0372 ± 0.004	0.0332			
		300	59.09	0.419 ± 0.05	0.40						
DPb	235/876 (26.8%)	520	22.69	0.909 ± 0.05	1.00	P1	0.2469 ± 0.009	0.2191	Tbot	80.4%	79.99%
		450	64.29	0.597 ± 0.05	0.50	D1	0.0248 ± 0.007	0.0235			
		400	26.88	0.227 ± 0.05	0.20						
DPc	283/1161 (15.7%)	1500	54.54	0.920 ± 0.05	1.00	P1	0.2860 ± 0.063	0.3070	Tbot	64.3%	67.85%
		1400	123.52	0.792 ± 0.05	0.80	D1	0.0064 ± 0.002	0.0067			
		1300	175.68	0.421 ± 0.05	0.40	P2	0.6028 ± 0.044	0.6610			
						D2	0.0218 ± 0.002	0.0261			

more critical values. We remark that, although they walk for longer, patient P1 in all three scenarios reaches fatigue levels compatible with those estimated for HL because they have time to rest while the robot is assisting the doctor.

The design-time analysis highlights that the most concerning aspect among the three scenarios is the fatigue level reached by patient P2 in DPc, as they also adhere to a critical fatigue profile (Elderly/Sick) and have to cover a significant distance from R1b to OFF3. For all experiments, threshold F_{high} (see Table 6.4) is set to 0.6. Therefore, some traces of the formal model feature the orchestrator instructing P2 to stop and rest (when $f \geq F_{high}$ holds), causing a delay in the mission. We remark that this safety measure embedded in the orchestrator is necessary to prevent the patient from reaching the maximum value of fatigue. Still, it is not sufficient to prevent them from reaching a significant (average) fatigue level (i.e., approximately 60%).

Results observed by deploying the scenarios in the hybrid setting corroborate the outcomes predicted at design time. As in the first validation phase, Table 16.4 highlights the results corresponding to larger estimation errors. Specifically, success probability ranges estimated at design time and reported in column DT (we recall that rates in column DEPL are point estimators and, thus, not reported as ranges) are the least accurate when the average success rate is closer to 50% or 60% (DPa with $\tau = 350$s and DPb with $\tau = 450$s, also highlighted in grey). On the other hand, estimations of the fatigue level have design time-to-deployment differences ranging from approximately 5% in the best case (D1 in DPc) to 16% in the worst case (D2 in DPc, also highlighted in grey). Nevertheless, we recall that, although errors are larger than those obtained with the first three scenarios, these are not an indication of inaccuracies within the formal model as only 5 deployment traces are performed for DPa, DPb, and DPc (compared to more than 100 for the previous validation phase).

Given the results of the first design-time analysis round and the data collected during deployment, the designer in charge of developing and maintaining these scenarios may choose to apply reconfiguration measures and refine the three robotic missions as described in Section 4.2.

The reconfiguration measures applied to the three scenarios (hereinafter referred to as R-DPa, R-DPb, and R-DPc) and the updated sequences of services are described in Table 16.5. Since the robot's battery was not a critical element in the first round of analysis, replacing the robot with a different one or recharging it would not impact the updated results.

For scenarios DPa and DPb, the sequence of services (i.e., the robot's

Table 16.5: *Reconfiguration measures applied to scenarios DPa, DPb, and DPc, and updated sequence of services.*

SCENARIO	RECONFIGURATION MEASURES	MISSION
R-DPa	The robot (Tbot) *leads* P1 directly to R2, then it serves D1 by *following* them to CUP1 and back to R2.	P1 Follower ENTR → R2, D1 Leader R2 → CUP1, D1 Leader CUP1 → R2
R-DPb	The robot (Tbot) serves D1 first by *fetching* the resource from CUP2 and *follows* them to R2, then it serves P1 and *leads* them to R2.	D1 Recipient OFF2 ↔ CUP2, D1 Leader OFF2 → R2, P1 Follower ENTR → R2
R-DPc	The robot (Tbot) *leads* P2 to R1b first and then provides the same sequence of services as scenario DPc.	P2 Follower ENTR → R1b, P1 Follower ENTR → R1a, D1 Recipient OFF1 ↔ CUP1, D2 Leader OFF3 → CUP2, D2 Leader CUP2 → OFF3, P1 Follower R1a → OFF1, P2 Follower R1b → OFF3

mission) is modified to reduce the time required to complete the mission or, in other words, to obtain a high probability of success for smaller values of τ. In these two cases, the patient is led straight to the examination room rather than to the waiting room first.[6]

As for the third scenario, the goal is to lighten the strain on the patient in the most delicate condition. Therefore, the robot serves P2 first and leads them to the doctor's office last to allow them a longer recovery time while in the waiting room. Table 16.6 reports the DSL2SHA ratio for the reconfigured scenarios. Note that the ratio is unvaried for R-DPc since only the order in which humans are served is changed. On the other hand, for R-DPa and R-DPb, removing one human declaration (13 words) reduces the SHA network size by 11% and 10.8%, respectively.

The results of the second round of design-time analysis are reported in Table 16.6. Quality metrics report a slight improvement compared to the first round of analysis since the robotic mission is shorter for R-DPa and R-DPb. In this case, verification time ranges from 10.54min to approximately 149.31min in the worst case, as scenario R-DPc is substantially unvaried in terms of performance.

[6]Note that, in a real healthcare facility, this may not be feasible in all cases: the examination room must either be empty when P1 is served or equipped to host more than one patient simultaneously.

Table 16.6: *Results of DSL2SHA calculation and of the design-time analysis of scenarios R-DPa, R-DPb, and R-DPc. For decreasing values of τ ([s]), the table contains the verification time ([min]), the success probability CI estimated through Uppaal, the estimated maximum fatigue value for all humans, and the estimated minimum charge value for the robot. Fatigue and charge level are only estimated for the maximum value of τ for each scenario.*

SC. (M)	DSL2SHA	τ	**Ver.Time** [min]	**Success Probability** ($\mathbb{P}_M(\diamond_{\leq\tau}\mathsf{scs})$)	HUM.	**Max. Fatigue** ($\mathrm{E}_{M,\mathsf{max}(\tau)}[\mathsf{max}(F_i)]$)	ROB.	**Min. Charge** ($\mathrm{E}_{M,\mathsf{max}(\tau)}[\mathsf{min}(C)]$)
R-DPa	227/768 (29.5%)	300 250 200	6.86 32.56 30.10	0.950 ± 0.05 0.498 ± 0.05 0.258 ± 0.05	P1 D1	0.1042 ± 0.014 0.0367 ± 0.004	Tbot	86.51%
R-DPb	227/781 (29.0%)	350 320 300	10.54 55.46 40.69	0.950 ± 0.05 0.741 ± 0.05 0.206 ± 0.05	P1 D1	0.1387 ± 0.008 0.0235 ± 0.001	Tbot	85.45%
R-DPc	283/1161 (15.7%)	1500 1400 1300	20.60 121.59 149.31	0.950 ± 0.05 0.854 ± 0.05 0.556 ± 0.05	P1 D1 P2 D2	0.3034 ± 0.061 0.0032 ± 0.001 0.3761 ± 0.029 0.0245 ± 0.016	Tbot	64.28%

Estimations inform us that for R-DPa and R-DPb, the updated mission can be completed successfully in less time as we obtain a success probability $> 90\%$ for τ equal to 300s and 350s, respectively (compared to 400s and 520s for the initial configuration). Since the patient only walks from the entrance to R2, their estimated maximum level of fatigue is also approximately 60% (in R-DPa) and 43% (in R-DPb) lower than fatigue estimations obtained with DPa and DPb, respectively. In contrast, the value remains essentially unchanged for D1 as they perform the same actions as in the original scenario.

For R-DPc, we observe that allowing P2 more time to rest in the waiting room reduces their maximum fatigue level by 37%. Furthermore, as they do not reach the critical threshold $\mathsf{C}_{\mathsf{high}} = 60\%$ anymore, the orchestrator does not instruct them to stop mid-service, leading to a slight reduction in the duration of the mission.

16.4 G3: Model Adjustment Impact Analysis

We illustrate the effectiveness of the model adjustment phase through six experimental scenarios from the healthcare setting (goal **G3**). This section mainly discusses how the learning procedure based on data collected at runtime reduces the error of verification results when exploiting automata learned through a batch of deployment traces to estimate the outcome of new scenarios.

Scenarios take place in the layouts shown in Fig. 16.1 and Fig. 7.2a. DP0 is the benchmark scenario used in Chapter 15, while DPa, DPb, and DPc are described in Table 16.3. MR1 and MR2 result from the two alternative paths for the scenario presented in Section 7.2, with $\mathsf{N}_\mathsf{r} = 2$ and both robots set at $\mathsf{C}_0 = 30\%$ (thus, two task handovers occur).

The workflow in Fig. 4.1 is followed. The designer configures the scenario (i.e., the layout, the involved agents, humans and robots, and the services constituting the robotic mission), and the tool automatically generates SHA network $\mathbb{M}$. For the first round of design-time analysis, the SHA network features the manually drafted version of automata modeling human behavior, of which Fig. 3.1 shows a snippet. Subsequently, SMC experiments are performed to estimate the probability of success of the robotic mission for decreasing values of time-bound τ. For all scenarios, we calculate through Uppaal the value of expression $\mathrm{E}_{\leq\hat{\tau}}[\mathsf{max}(F_{\mathsf{HUM}})]$, which is the maximum fatigue reached by each human (indicated as HUM) with the highest time bound considered for that scenario (indicated as $\hat{\tau}$, e.g., 340s in DP0). The results of the first round of design-time analysis for all scenarios

CS	τ	**Success Probability** ([%]) $\mathbb{P}_{\mathrm{M}}(\diamond_{\leq \tau}\mathsf{scs})$				**Complexity** $\lvert\mathrm{M}\rvert$ ($[\times 10^3]$)		**Verification Time** [min]		HUM	**Exp. Fatigue** ([%]) $\mathrm{E}_{\leq \hat{\tau}}[\mathsf{max}(F_{\mathsf{HUM}})]$			
		DT-1	DT-2	DEPL	$\Delta\mathrm{E}_p(\tau)$	DT-1	DT-2	DT-1	DT-2		DT-1	DT-2	DEPL	$\Delta\mathrm{E}_f(\hat{\tau})$
DP0	340s	95.0 ± 5	83.5 ± 5	80.5	-14.3	175.9	267.5	6.74	14.4	P1	24.6 ± 4	39.2 ± 7	41.4	-35.2
	260s	56.5 ± 5	44.5 ± 5	41.3	-29.1			4.50	10.0	D1	5.4 ± 1	7.4 ± 1	6.8	-11.4
	180s	12.5 ± 5	9.48 ± 5	8.75	-34.5			2.47	6.11					
DPa	500s	95.0 ± 5	79.3 ± 5	74.0	-21.2	175.9	267.5	16.4	34.8	P1	20.4 ± 3	38.4 ± 7	31.1	-10.7
	325s	58.6 ± 5	46.7 ± 5	40.0	-29.8			9.09	20.9	D1	5.5 ± 1	6.2 ± 1	5.88	-1.0
	200s	6.5 ± 5	5.0 ± 5	5.0	-30.0			4.67	14.8					
DPb	580s	95.0 ± 5	75.3 ± 5	81.0	-10.2	179.3	291.5	14.8	32.0	P1	21.7 ± 1	41.0 ± 3	34.2	-16.7
	450s	60.2 ± 5	45.5 ± 5	49.0	-15.7			8.08	14.7	D1	7.6 ± 2	13.4 ± 2	10.9	-7.5
	300s	6.67 ± 5	11.0 ± 5	10.0	-23.3			4.75	8.34					
DPc	1500s	93.7 ± 5	74.1 ± 5	80.0	-9.8	193.4	397.0	31.6	92.1	P1	22.6 ± 4	32.7 ± 2	28.6	-13.2
	1400s	74.2 ± 5	55.6 ± 5	61.0	-12.8			24.9	70.9	D1	5.9 ± 1	4.4 ± 3	5.1	-2.0
	1300s	37.9 ± 5	25.2 ± 5	28.0	-25.4			20.9	54.3	P2	51.9 ± 3	62.9 ± 9	61.7	-13.9
	1200s	6.5 ± 5	5.0 ± 5	5.0	-30.0			18.5	37.6	D2	2.2 ± 1	2.8 ± 1	2.4	8.3
MR1	1200s	95.0 ± 5	85.1 ± 5	89.0	-2.4	371.7	626.8	26.0	89.9	P1	39.4 ± 3	48.3 ± 4	46.9	-12.9
	1000s	63.5 ± 5	45.5 ± 5	51.0	-13.7			16.3	51.6	D1	2.8 ± 1	2.9 ± 1	3.1	-3.2
	800s	27.9 ± 5	23.7 ± 5	22.0	-19.1			12.9	33.4	D2	1.2 ± 1	1.8 ± 1	1.5	6.9
	700s	6.89 ± 5	6.34 ± 5	6.0	-9.2			8.70	14.5					
MR2	1200s	91.2 ± 5	81.3 ± 5	80.0	-12.4	368.3	602.8	28.7	83.8	P1	35.7 ± 2	43.6 ± 2	40.4	-11.6
	1000s	58.6 ± 5	42.4 ± 5	47.0	-14.9			17.1	50.4	D1	4.0 ± 3	7.4 ± 3	5.8	-3.4
	800s	24.5 ± 5	18.7 ± 5	21.0	-5.7			13.7	37.5	D2	1.5 ± 1	1.8 ± 1	1.6	5.0
	700s	7.68 ± 5	6.81 ± 5	5.0	-17.4			7.46	12.9					

Table 16.7: *Experimental results obtained from the case studies (CS). For each scenario, the complexity of the SHA network ($\lvert\mathrm{M}\rvert$) is the cumulative size of its SHA (one for each agent), which is a function of the number of locations, edges, and of the cardinality of variables' domains.*

are shown in Table 16.7 (columns DT-1).

Secondly, we exploit the virtual deployment environment to gather a large amount of data without putting a strain on real subjects. Through the simulation environment, we replicate a broader spectrum of human behaviors affecting the fatigue curve throughout the scenario.[7] These actions range from running and carrying loads (which are daily tasks for healthcare professionals) to sitting or standing in an uncomfortable environment (i.e., with a very low or very high room temperature or a high degree of humidity, such as in field hospitals). We perform multiple simulations of scenario DP0 to collect a large batch of runtime observations (i.e., traces), serving as *training* data as human subjects perform a wider range of actions than the ones captured by the initial SHA. For this specific experiment, 5407 traces (with *automated* human avatars) have been collected, each lasting approximately $340s$. To obtain the same amount of data in a physical environment, running the application non-stop for 21 workdays would be necessary.

Collected runs are fed to $\mathsf{L}^*_{\mathrm{SHA}}$ to learn human behavior. With this training dataset, $\mathsf{L}^*_{\mathrm{SHA}}$ terminates in approximately $37\mathsf{min}$ returning an SHA modeling human behavior that has five times the number of locations than the original one (i.e., the outcome of Exp.**5** described in Section 15.2). The updated SHA modeling human actions are then plugged into the SHA network to repeat the design-time analysis, for all scenarios, with the refined model (columns DT-2 in Table 16.7).

The learned model is tested with scenarios different than DP0 (i.e., the source of training data). DPa, DPb, and DPc are deployed in the hybrid setting with a real robotic device, while MR1 and MR2 are fully simulated. Runtime observations collected while deploying these five scenarios (collectively, 500 runs) serve as *testing* data.

Table 16.7 reports the success rate and maximum fatigue level observed at runtime (columns DEPL). Testing scenarios are newly verified with Uppaal with the refined model of human behavior to evaluate how the learning phase reduces the estimation error of SMC results, as explained in the following. Let $\overline{\mathrm{X}}_{\mathsf{DT}\text{-}i}(\tau)$ be the average of metric X (the success rate or fatigue level) calculated in Uppaal with time bound τ for phase $i \in \{1, 2\}$. Let $\overline{\mathrm{X}}_{\mathsf{DEPL}}(\tau)$ be the average value of metric X observed at *runtime* within time bound τ. Let $\mathrm{E}_{\mathrm{X},i}(\tau)$ be the error between estimation $\overline{\mathrm{X}}_{\mathsf{DT}\text{-}i}(\tau)$ and $\overline{\mathrm{X}}_{\mathsf{DEPL}}(\tau)$, calculated as in Eq.16.4.

[7] The extended set of human behaviors is not exploited for the experimental validation of **G1**, given that its goal is to assess the accuracy of the formal model (specifically, the robot's and the battery's model) when it behaves correspondingly to the real setting, i.e., in ordinary conditions.

$$\mathrm{E}_{\mathrm{X},i}(\tau) = \frac{|\overline{\mathrm{X}}_{\mathsf{DT}\text{-}i}(\tau) - \overline{\mathrm{X}}_{\mathsf{DEPL}}(\tau)|}{\overline{X}_{\mathsf{DEPL}}(\tau)} \cdot 100 \tag{16.4}$$

We indicate as $\Delta\mathrm{E}_\mathrm{X}(\tau) = \mathrm{E}_{\mathrm{X},2}(\tau) - \mathrm{E}_{\mathrm{X},1}(\tau)$ the difference between the error with the model learned through $\mathsf{L}^*_{\mathrm{SHA}}$ and the initial model. When $\Delta\mathrm{E}_\mathrm{X}(\tau)$ is negative, SMC results obtained through the SHA network (partially) learned by $\mathsf{L}^*_{\mathrm{SHA}}$ are more accurate than the manually drafted network ($\mathrm{E}_{\mathrm{X},1}(\tau) > \mathrm{E}_{\mathrm{X},2}(\tau)$ holds). Since $\mathsf{L}^*_{\mathrm{SHA}}$ is the first automata learning algorithm targeting SHA, we can internally assess its impact on our design and development framework. Still, we cannot evaluate its performance against that of alternative algorithms.

The source of the large estimation errors of DT-1 experiments is the broader range of actions that humans perform at runtime, which are not captured by the initial SHA network but learned through $\mathsf{L}^*_{\mathrm{SHA}}$ and taken into consideration for DT-2 experiments. To highlight the gap between known human actions before and after the learning phase, Fig. 16.4 shows how long, on average, the subjects from scenario DP0 spend in each state of the SHA modeling human behavior (e.g., $stand$ and $walk$ in Fig. 3.1). The upper bar plot shows the distributions for the initial behavior model, whereas the bottom bar plot shows the distributions for the SHA learned by $\mathsf{L}^*_{\mathrm{SHA}}$. Distributions are obtained by analyzing 1000 traces of the SHA network generated through Uppaal. The plot highlights how human behavior has a significantly higher degree of variability than the one accounted for by the initial model. More specifically, the refined model features locations with more critical fatigue rate distributions (e.g., walking in uncomfortable environments), which justify the gap between the two rounds of fatigue estimations.

When exploiting the model learned through $\mathsf{L}^*_{\mathrm{SHA}}$ to estimate fatigue for all subjects of scenarios other than DP0 (i.e., the source of traces used for learning), we obtain $\Delta\mathrm{E}_f(\hat{\tau}) < 0$ in 11 cases out of 14, indicating that the learning procedure results in a formal model that more accurately captures reality. The three subjects for which $\Delta\mathrm{E}_f(\hat{\tau}) \geq 0$ holds are professionals with non-critical fatigue profiles, indicating that the learned model may lead to an overestimation of the fatigue level for this category of subjects. We remark that, in both rounds, for these subjects, the estimated fatigue levels are not critical ($< 10\%$). Thus, small fluctuations do not overturn the result of the design-time analysis.

As shown in Table 16.7, DT-1 experiments establish that all scenarios can be completed successfully with probability greater than 90% with τ ranging from about 6min (for DP0) to 25min (for DPc). For decreasing

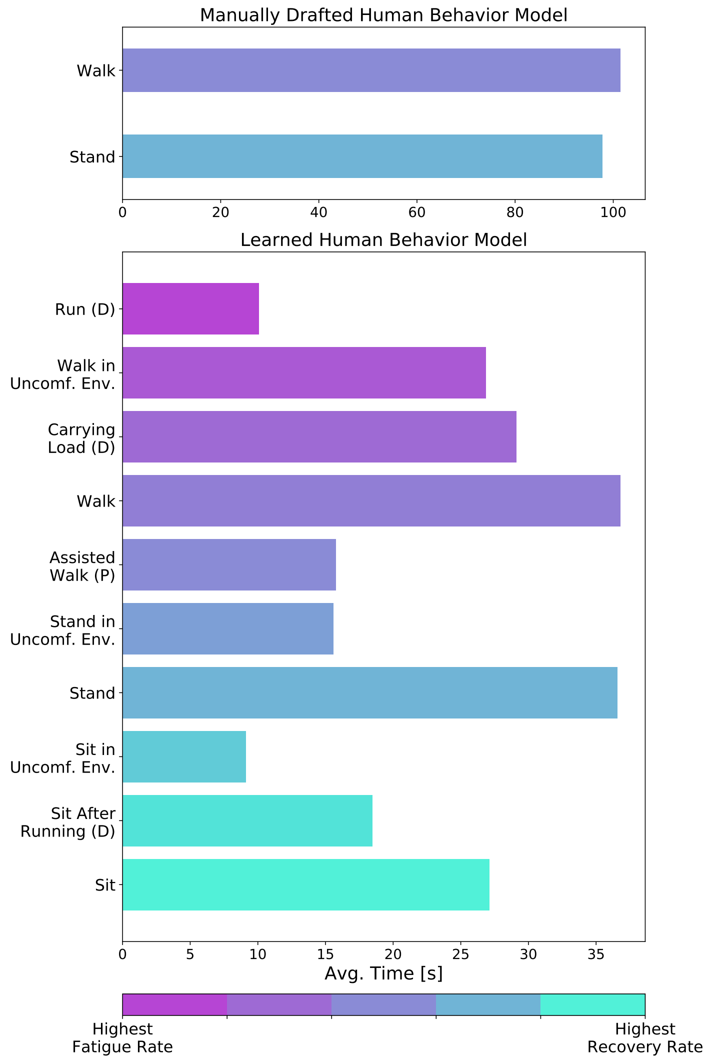


Figure 16.4: *Average time ([s]) spent by the case study subjects in each state of the human behavior model state. States that are only feasible to a professional (resp. patient) are marked with a (D) (resp. (P)). The bars' color code is shown at the bottom.*

values of τ, the success probability decreases since the agents do not have sufficient time to complete their tasks. Nevertheless, as per Table 16.7, the success rate observed at runtime (column DEPL) is lower than its initial estimation. Fatigue impacts the probability of success since the orchestrator (i.e., the robot's controller) instructs humans to stop and rest when their fatigue level exceeds a critical threshold, leading to a delay in complet-

ing the mission (thus, the success probability within a specific time-bound decreases). The error reduction of fatigue estimation corresponds to an improvement in the success probability calculation. As for success probability, we obtain $\Delta \mathrm{E}_p(\tau) < 0$ for all scenarios and all time-bounds.

16.5 Discussion

We can summarize how we have addressed the validation goals as follows:

G1. We have performed more than 300 runs of three experimental scenarios in a digital-twin environment involving simulated humans and a real robotic device communicating via ROS. Collected deployment traces have been exploited to assess the accuracy of the formal model and SMC results.

G2. We have assessed the coverage of our development framework with respect to existing real-world scenarios in the service robotics domain. Then, we collected the most recurring tasks within the set of real applications into three scenarios to be analyzed and developed through our framework. In this regard:

(a) We have assessed the efficiency of the presented DSL by calculating the number of words necessary to configure the whole SHA network (i.e., the DSL2SHA metric).

(b) We have analyzed the three scenarios at design time and the results of such analysis are reflected by the deployment traces.

(c) We have reconfigured the three scenarios in light of the collected deployment traces and iterated the design-time analysis.

G3. We have exploited the outcome of automata learning resulting from deployment traces of a benchmark scenario to perform the design-time analysis of five different scenarios and assessed the accuracy gain resulting from the refined model of human behavior.

Concerning the analysis of the formal model accuracy (goal **G1**), as discussed in Section 11.4, we obtain relative estimation errors for the probability of success and charge level smaller than 10% also in boundary conditions, e.g., involving subjects with a critical fatigue profile or a robot close to complete discharge. Success probability and minimum battery charge ranges provide empirical evidence of the reliability of the SHA modeling the robotic system. Since we have only performed experiments with virtual human agents whose model derives from literature analysis, the validation

of the formal model of human behavior needs further investigation. As future work, the validation process is to be completed by performing experiments with real human subjects to assess the accuracy of SHA modeling human behavior.

Coverage analysis (enabling the pursuit of goal **G2**) yields that more than 80% of the collected real-world scenarios within the scope of this work can be designed and deployed through the presented framework. The analysis carried out on scenarios DPa, DPb, and DPc shows how the framework supports practitioners throughout the entire development process by automating the generation of the formal model and the deployment of the resulting application.

The analysis of the DSL2SHA ratio (smaller than 30% in all cases) shows that the DSL requires less effort than manually drafting the formal model (goal **G2**a). DSL2SHA values show that the DSL grows more efficient than the manual creation of the formal model as the size of the SHA network in question increases. This is because the portion of DSL configuring the geometrical layout (which is the same for all scenarios in Section 11.5, regardless of the mission's complexity) is the most verbose element. This issue will be addressed in the future by automatically acquiring the information regarding the layout from planimetry data to boost the DSL's efficiency significantly.

Indicators estimated through the design-time analysis phase of the three scenarios are corroborated by the observations collected during deployment (goal **G2**b). With a small number of deployment traces (i.e., 5), relative estimation errors do not exceed 16%. As for goal **G2**c, reconfiguration measures applied to scenarios DPa and DPb (through minor modifications to the DSL specification) improve the estimated success probabilities with a time-bound smaller by 25% (300s compared to 400s) and 33% (350s compared to 570s), respectively. As previously discussed, reconfiguring DPc reduces the physical effort imposed on subject P2.

Concerning the model adjustment impact (i.e., goal **G3**), the accuracy gain in estimating success probability ranges from 2.4% to 30% in the best case (except for the training scenario DP0). These results indicate that the learning phase, although time-consuming (both in terms of data collection and $\mathsf{L}^*_{\mathrm{SHA}}$ running time), is beneficial to the development process, also for scenarios different than the training one.

Naturally, reducing the error has a price in terms of verification time given the increased SHA network complexity. Phase DT-1 takes between 2.47min and 31.6min, while, for phase DT-2, SMC experiments last from 6min to 92min in the worst case. For this specific phase, experiments are

performed with a bound on runs with different values of α to obtain $\epsilon = 0.05$ (as shown in Table 16.7) but isolate the dependency on τ and the size of the SHA network (indicated as $|\mathrm{M}|$ in Table 16.7). Results show that verification time increases more steeply with τ than with $|\mathrm{M}|$. As Table 16.7 highlights, smaller (in terms of $|\mathrm{M}|$), longer (in terms of τ) missions (e.g., DPc with $\tau = 1500$s) take longer to verify than bigger, shorter ones (e.g., MR1 with $\tau = 1200$s).

CHAPTER 17

Conclusions and Future Work

This chapter summarizes the contributions presented in this thesis and the main results that have been obtained and illustrates future research directions.

17.1 Conclusions

This thesis introduces a model-driven development toolchain for service robotics applications in which human-robot interaction is a prominent factor. The ever increasing demand for services, particularly in healthcare due to a progressively aging society, constitutes a significant challenge in terms of scalability and effectiveness. A strategic deployment of service robots in these settings can improve the quality of life of those who request the service. At the same time, robots and professionals can work cohesively to lift the latter from clerical and time-consuming tasks.

However, the technological barrier between robotics and the technical skillset of practitioners remains considerable [60]. The thesis addresses the lack of software development techniques that consider factors related to human subjects' well-being with guarantees of reliability and accessibility.

The presented framework supports practitioners (or designers) from an

early stage of development of the robotic application through iterative refinements until a conclusive design is devised.

A custom DSL enables the specification of the interactive scenarios, precisely the layout, the agents' characteristics, and the robotic mission. The presented formal modeling approach is then exploited to automatically generate the SHA network capturing the specified scenario and perform formal analysis through SMC. Verification results provide insight into the outcome of the mission and the strain it poses on the involved subjects.

The scenario can then be iteratively re-configured until it is ready for deployment, either on the field or in a simulated environment for further investigation. Collected data is then fed to a novel active automata learning algorithm ($\mathsf{L}^*_{\mathrm{SHA}}$) to exploit the knowledge accumulated on the field and adjust the model of human behavior, which constitutes the primary source of uncertainty in the targeted domain.

In the following, we summarize and discuss the framework's main features showcased throughout the thesis.

Dependability

The framework relies on formal methods throughout all phases. Since the framework targets everyday settings, people without prior training in interacting with robots or proper safety measures are likely to be widely involved. Therefore, the targeted domain is strongly safety-critical and poses high dependability demands, making the soundness of formal models and formal verification techniques a valuable asset. However, clear box models of human decisions (even in contained scenarios) would be intractable with exhaustive techniques given the considerable volume of possible contingencies. To this end, the probabilistic formalization and the use of SMC lead to a reasonable trade-off between performance and (*probabilistic*) guarantees.

The empirical validation presented in Section 16.2 assesses how accurately the formal model captures the real agents on benchmark scenarios. Results show that, in the worst cases, estimation errors amount to 6.7% for the success probability, 0.61% for the robot's battery charge, and 8.6% for human fatigue.

The accuracy of the deployment framework, exploited to obtain such results, is assessed in Chapter 11, highlighting errors no higher than 5.35%. We remark that no standardization exists on maximum allowable errors in this domain, and thus the decision on whether these thresholds fit a facility's policies is left to human investigation.

Flexibility

Although some of its features are tailored to the healthcare domain, the presented framework is sufficiently general to be applied to any service setting. Furthermore, it applies to a wide variety of scenarios of which the different experimental validation activities presented throughout the thesis analyze 12 with further variations.

The flexibility of the approach is more thoroughly investigated through the coverage analysis presented in Section 16.3.1. The analysis assesses that 24 out of 27 realistic scenarios collected by surveying the literature and industrial use cases (also serving as a source of inspiration for the scenarios used for empirical validation) are covered by the framework, leading to a coverage rate higher than 85%.

Furthermore, we remark that, as outlined in Section 12.1, the $\mathsf{L}^*_{\mathrm{SHA}}$ algorithm developed for this thesis is domain-agnostic and applicable to any CPS that meets a specific set of requirements. As a matter of fact, besides human behavior learning, which is the main focus of this thesis, it has been tested on a different real-life use case dealing with the energy consumption of machining centers as presented in Section 13.3.

Adaptability

As previously discussed, humans constitute a critical source of uncertainty for the targeted domain. For a model-driven approach, the unavoidable model-to-reality gap constitutes a significant threat to the validity of the approach itself. Therefore, given that the starting model of human behavior is necessarily an underapproximation, introducing the automated model adjustment phase is a helpful countermeasure.

As shown by the experimental results in Section 16.4, learning a refined model of human behavior from field data increases the accuracy of the formal analysis. Specifically, the success probability estimation across the six benchmark scenarios with the refined model of human behavior is more accurate by 18.1% while the accuracy increase for the fatigue estimation is, on average, 7.7%.[1]

We remark that the experiments in Section 16.4 also assess how the model learned from data related to a particular scenario increases the accuracy of the formal analysis of different scenarios. Therefore, the relevance of these results also relates to the fact that all costs (in terms of time and machinery/space employment) endured to test a scenario on the field, collect

[1] The two metrics are calculated as the average of columns $\Delta\mathrm{E}_p(\tau)$ and $\Delta\mathrm{E}_f(\hat{\tau})$ of Table 16.7, respectively.

data, and train a new model are damped when depreciate when exploiting the same refined model on different scenarios.

User-Friendliness

As discussed in Chapter 1, target users of the framework are professionals without prior training in formal methods or software development. Therefore, one of the primary objectives of this research work is to keep the technological barrier as low as possible.

To this end, we have developed a lightweight textual notation (i.e., , the DSL presented in Section 5.2) to serve as a friendly interface to the overall development pipeline. The DSL2SHA indicator presented in Section 16.3.2 (smaller than 30% on all scenarios, including their reconfigured versions) is an estimate of the manual effort saved on the practitioner's side when configuring a scenario through the DSL rather than directly drafting the formal model in Uppaal.

Furthermore, the workflow is highly automated as per Fig. 4.1. Starting from the DSL file, the following tasks are performed automatically: generating the formal model, running SMC, setting up the deployment environment, mining traces, learning the refined model, and iterating the formal analysis. The activities currently performed by the designer are the configuration of the scenario (which, as previously argued, cannot be automated), the assessment of the verification results, and the selection of potential reconfiguration measures.

17.2 Future Research Outlook

In the future, the work can be extended in different directions.

Firstly, the current limitations of the approach ought to be addressed. The framework's experimental validation presented in Chapter 16 must be completed by assessing the formal model's accuracy in capturing real human behavior. Similarly, a user study can be carried out involving practitioners from the targeted user audience to empirically assess the framework's accessibility, specifically concerning the DSL configuration phase.

The current phases of the framework can also be further extended. A study on how cognitive models of the human decision-making process can be formally modeled and incorporated into the analysis is currently underway. Such models will constitute an alternative to the erroneous behavior model presented in Chapter 8, highlighting different ways the robotic mission can unravel. The designer will be in charge of selecting the desired behavioral model during the configuration phase.

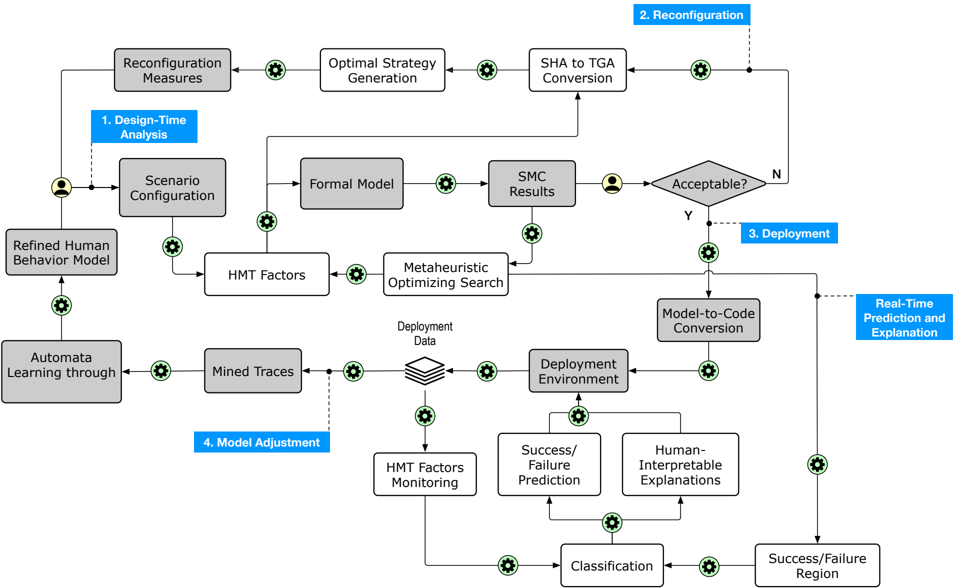


Figure 17.1: *Framework's workflow highlighting the tasks planned as future extensions. Existing tasks are represented as grey boxes, while future extensions are represented as white boxes.*

On the other hand, SHA modeling human behavior can be further extended to capture physiological factors other than muscle fatigue, such as heart rate variability, breathing rate, or cognitive load. Different patterns may impact each physiological factor differently (for example, an action that does not cause significant physical strain may stress the subject mentally), and this should emerge from the formal analysis.

Although its coverage of existing case studies is satisfactory, the predetermined set of interaction patterns may become a limitation in the long run as more case studies with different interaction contingencies emerge. To this end, we plan to extend the DSL and the formal model generation mechanism with new primitives that allow designers to define new interaction patterns and generate the corresponding SHA automatically.

The L^*_{SHA} algorithm and, from a higher-level perspective, the model adjustment phase can also be extended. Firstly, the current limitation on the predetermined candidate flow conditions ought to be lifted. This requires extending the mi query to identify a possible flow condition for the set of segments under analysis and assess its similarity with previously identified functions through proper techniques. Secondly, the algorithm can also be improved to extend the learning to probability weights on edges (i.e.,

function $\mathcal{P}$ from Definition 1). To this end, the results obtained by Tappler et al. [207] in developing algorithm $\mathsf{L}^*_{\text{MDP}}$ could be exploited. Finally, the user-friendliness and flexibility of the overall framework can be further improved by extending the DSL with primitives to customize the learning process, i.e., to define which signals are available in the environment and how they can be exploited to mine traces fed to $\mathsf{L}^*_{\text{SHA}}$.

The overall development framework can also be extended to broaden the analysis as represented in Fig. 17.1.

The deployment phase can be enriched with a technique to monitor the scenario *while* it is executing and provide an estimation of its outcome in real time. However, although SMC is cheaper and faster than exhaustive model-checking, verification times are typically in the order of minutes or higher for more complex scenarios. Therefore, obtaining a real-time prediction through SMC before the mission's end (either in success or failure) may not be feasible.

To this end, Artificial Intelligence techniques targeting Human-Machine Teaming (HMT) and explainability can be exploited. The variables of a scenario that characterize the interaction between the human and the robot and are highly volatile (i.e., they may significantly vary during deployment due to environmental reasons) are referred to as HMT factors, which may be both categorical and continuous. HMT factors include the human's fatigue profile or the robot's speed. The scenario configuration defined by the designer originates a model space containing any possible combination of HMT factor values. Each combination is subject to SMC to estimate the corresponding success/failure probability. Since the size of the HMT factor space is likely extensive, performing SMC for all combinations may not be feasible; to this end, metaheuristic optimizing search can be exploited to guide the exploration [23]. The so-obtained success/failure regions are used to train a classification model (e.g., a Random Forest or Neural Network) that monitors HMT factors at runtime and yields a success/failure prediction. The classification model also generates a local explanation interpretable by the designer of how HMT factors impact the prediction.

Finally, having to manually choose and apply one or multiple reconfiguration measures when SMC results are not acceptable is also a current limitation of the framework. To this end, model abstraction and optimization techniques can be exploited. The previously described HMT factor space can be exploited to abstract the SHA network into a TGA network with uncontrollable edges capturing the transitions subject to uncertainty. Tools such as Uppaal Stratego [45] can be exploited to calculate the optimal strategy, i.e., the mission plan that, also in the event of HMT factor

configuration change, maximizes the probability of success. The resulting strategy must then be re-converted into a SHA and incorporated into the original network.

APPENDIX $\mathcal{A}$

Learned SHA

All learned SHA described in Chapter 13 and Chapter 15 are shown in full in the following. Each learned SHA is paired with the reference one (if it exists) for visual inspection.

A.1 Thermostat CPS

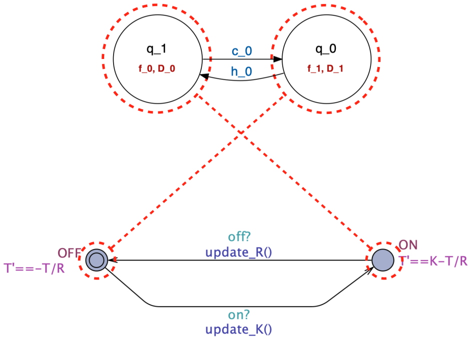


Figure A.1: *Mapping between learned SHA (top) and reference SHA (bottom) for Exp.1.*

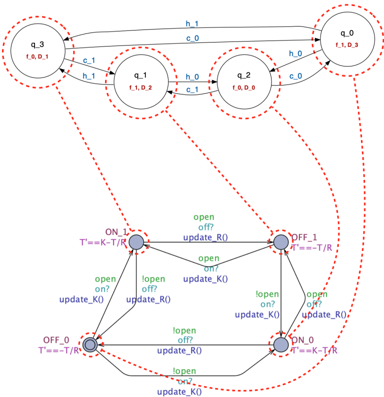


Figure A.2: *Mapping between learned SHA (top) and reference SHA (bottom) for Exp.2.*

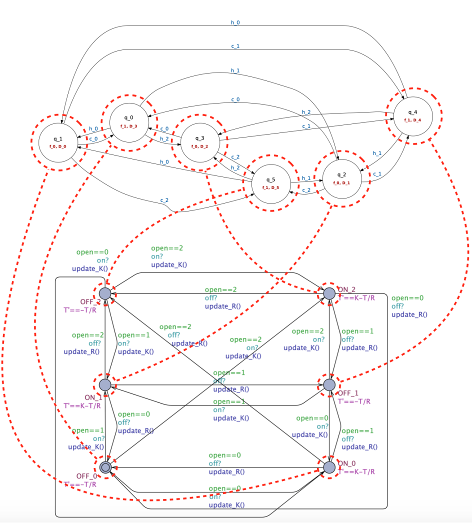


Figure A.3: *Mapping between learned SHA (top) and reference SHA (bottom) for Exp.3.*

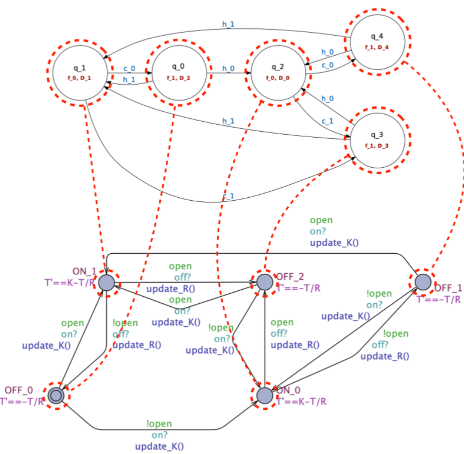


Figure A.4: *Mapping between learned SHA (top) and reference SHA (bottom) for Exp.4.*

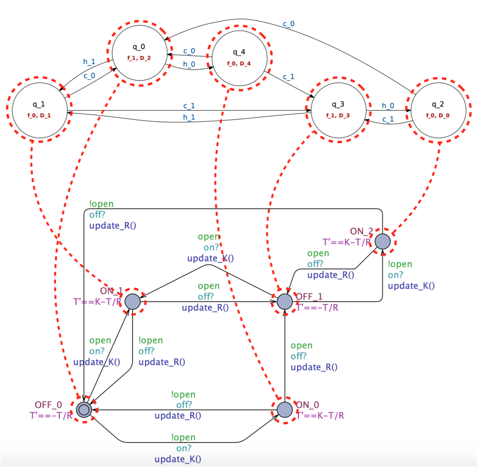


Figure A.5: *Mapping between learned SHA (top) and reference SHA (bottom) for Exp.5.*

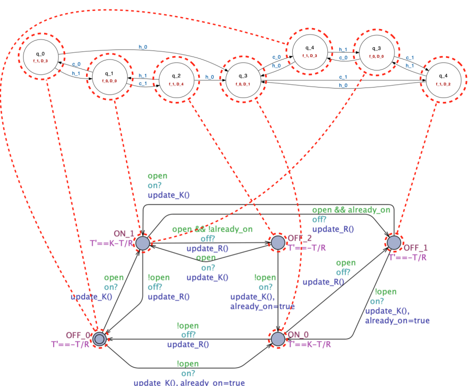


Figure A.6: *Mapping between learned SHA (top) and reference SHA (bottom) for Exp.6.*

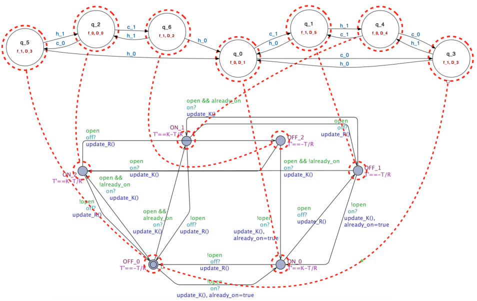


Figure A.7: *Mapping between learned SHA (top) and reference SHA (bottom) for Exp.7.*

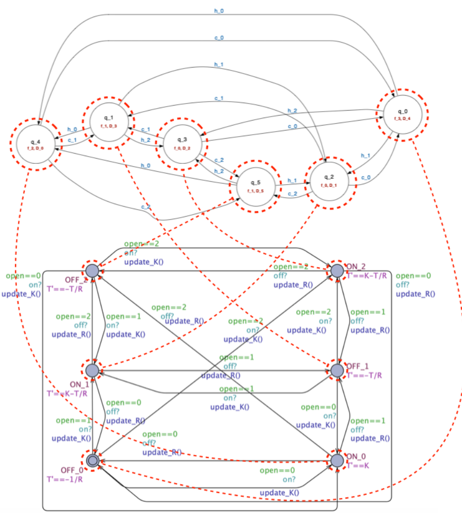


Figure A.8: *Mapping between learned SHA (top) and reference SHA (bottom) for Exp.8.*

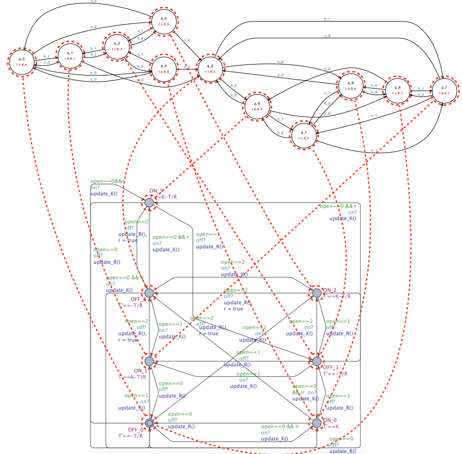


Figure A.9: *Mapping between learned SHA (top) and reference SHA (bottom) for Exp.9.*

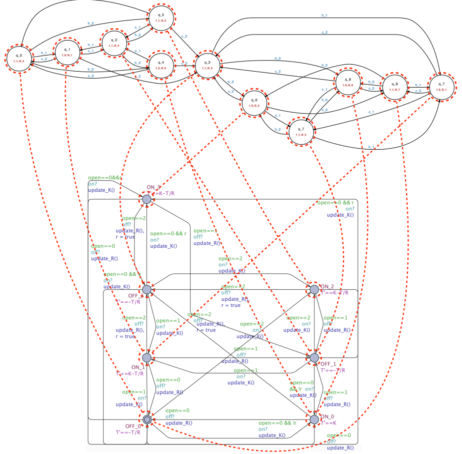


Figure A.10: *Mapping between learned SHA (top) and reference SHA (bottom) for Exp.10.*

A.2 Human-Robot Interaction CPS (Model-Driven Experiments)

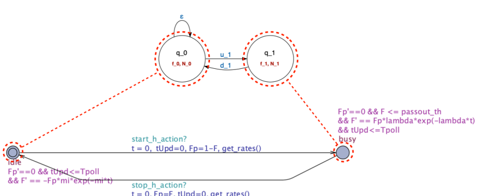


Figure A.11: *Mapping between learned SHA (top) and reference SHA (bottom) for Exp.1.*

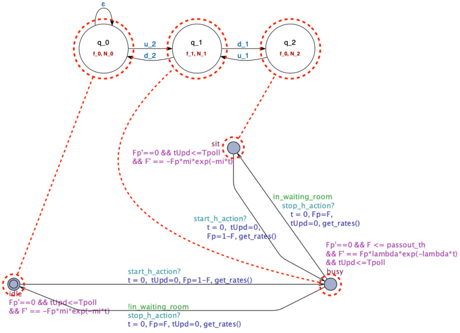


Figure A.12: *Mapping between learned SHA (top) and reference SHA (bottom) for Exp.2.*

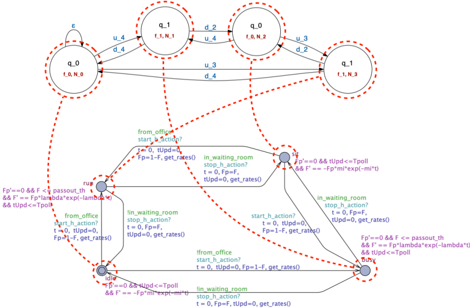


Figure A.13: *Mapping between learned SHA (top) and reference SHA (bottom) for Exp.3.*

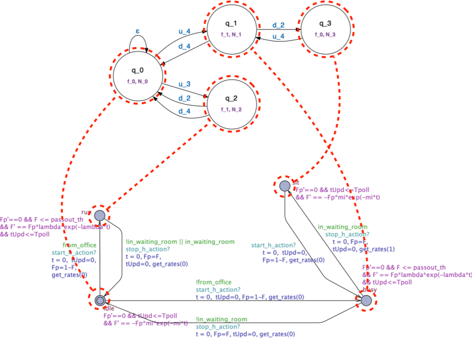


Figure A.14: *Mapping between learned SHA (top) and reference SHA (bottom) for Exp.4.*

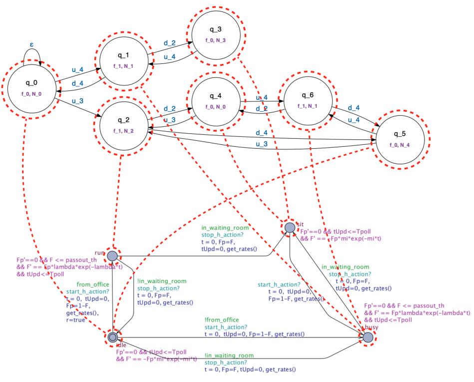


Figure A.15: *Mapping between learned SHA (top) and reference SHA (bottom) for Exp.5.*

APPENDIX B

Full DSL Sources

This Appendix contains the DSL configuration of scenarios DPa, DPb, and DPc from Chapter 16. The complete .dsl file is constituted by the concatenation of Listings B.1 through B.5.

Listing B.1: *DSL section defining layout areas (i.e., the rectangles higlighted in Fig. 9.2) and POIs: specifically, the entrances to the three offices, to the waiting room and emergency room, the two cupboards, main entrance, and robot's recharge station. As all scenarios are set in the same layout, the DSL features only one layout definition.*

```
1  param measurement_unit cm
2  define layout:
3  area a1 in (0.0,110.0) (1550.0,299.5)
4  area a2 in (0.0,110.0) (185.0,850.0)
5  area a3 in (0.0,672.5) (1550.0,850.0)
6  area a4 in (1352.0,110.0) (1550.0,850.0)
7  area a5 in (2970.0,110.0) (4512.5,299.5)
8  area a6 in (2970.0,110.0) (3155.0,850.0)
9  area a7 in (2970.0,672.5) (4512.5,850.0)
10 area a8 in (4322.0,110.0) (4512.5,850.0)
11 area a9 in (1945.0,0.0) (2670.0,695.0)
12 area a10 in (1352.0,110.0) (3155.0,425.0)
13
14 poi OFF1 in (200.0, 200.0)
15 poi OFF2 in (4400.0, 200.0)
```

```
16  poi OFF3 in (4400.0, 700.0)
17  poi R1a in (1200.0, 680.0)
18  poi R1b in (400.0, 270.0)
19  poi R2 in (4000.0, 270.0)
20  poi CUP1 in (1400.0, 450.0)
21  poi CUP2 in (3000.0, 450.0)
22  poi ENTR in (2300.0, 600.0)
23  poi RECH in (4250.0, 450.0)
```

Listing B.2: *DSL section defining robot* Tbot *and its features. As illustrated in Chapter 16, it is a TurtleBot3 Waffle Pi starting with* 90% *of charge.*

```
1  define robots:
2  robot Tbot in (2300.0, 400.0) id 1 type turtlebot3_wafflepi
     charge 90
```

Listing B.3: *DSL section defining the human subjects and their features. Patients (*P1a, P1b, P1c*, and* P2c*) all have* sick *fatigue profiles, and only* P2c *belongs to the* elderly *age group. Doctors (*D1a, D1b, D1c*, and* D2c*) all have* healthy *fatigue profiles and belong to the* elderly *age group, except for* D2c*. Walking speeds are set to* $40\mathrm{cm/s}$ *for patients, and* $100\mathrm{cm/s}$ *for doctors.*

```
1  define humans:
2  human P1a in (2300.0, 600.0) id 1 speed 40.0 is young_sick
     freewill normal
3  human D1a in (4400.0, 700.0) id 2 speed 100.0 is elderly_healthy
       freewill low
4
5  human P1b in (2300.0, 600.0) id 1 speed 40.0 is young_sick
     freewill normal
6  human D1b in (4400.0, 700.0) id 2 speed 100.0 is elderly_healthy
       freewill normal
7
8  human P1c in (2290.0, 600.0) id 1 speed 40.0 is young_sick
     freewill high
9  human P2c in (2400.0, 580.0) id 2 speed 40.0 is elderly_sick
     freewill normal
10 human D1c in (200.0, 200.0) id 3 speed 100.0 is elderly_healthy
     freewill low
11 human D2c in (4400.0, 700.0) id 4 speed 100.0 is young_healthy
     freewill normal
```

Listing B.4: *DSL section defining the service sequences (i.e., the robotic missions). As described in Section 11.5, each scenario corresponds to a mission declaration. Service sequences are defined as in Table 16.3 and Table 16.5.*

```
1  define mission DPa:
2  do robot_leader for P1a with target R1b
3  do robot_follower for D1a with target CUP1
4  do robot_follower for D1a with target R2
5  do robot_leader for P1a with target R2
```

```
6
7   define mission DPb:
8   do robot_leader for P1b with target R1a
9   do robot_transporter for D1b with target CUP2
10  do robot_follower for D1b with target R2
11  do robot_leader for P1b with target R2
12
13  define mission DPc:
14  do robot_leader for P1c with target R1a
15  do robot_leader for P2c with target R2
16  do robot_transporter for D1c with target CUP1
17  do robot_follower for D2c with target CUP2
18  do robot_follower for D2c with target OFF3
19  do robot_leader for P1c with target OFF1
20  do robot_leader for P2c with target OFF3
21
22  define mission R–DPa:
23  do robot_leader for P1a with target R2
24  do robot_follower for D1a with target CUP1
25  do robot_follower for D1a with target R2
26
27  define mission R–DPb:
28  do robot_transporter for D1b with target CUP2
29  do robot_follower for D1b with target R2
30  do robot_leader for P1b with target R2
31
32  define mission R–DPc:
33  do robot_leader for P2c with target R1b
34  do robot_leader for P1c with target R1a
35  do robot_transporter for D1c with target CUP1
36  do robot_follower for D2c with target CUP2
37  do robot_follower for D2c with target OFF3
38  do robot_leader for P1c with target OFF1
39  do robot_leader for P2c with target OFF3
```

Listing B.5: *DSL section defining the queries to be performed for the design-time analysis. Queries defined in this Listing yield the results shown in Table 16.4 and Table 16.6.*

```
1   define queries of mission DPa:
2   compute probability_of_success with duration 400 runs auto
3   compute probability_of_success with duration 350 runs auto
4   compute probability_of_success with duration 300 runs auto
5   compute expected_charge with duration 400 runs auto
6   compute expected_fatigue with duration 400 runs auto
7
8   define queries of mission DPb:
9   compute probability_of_success with duration 520 runs auto
10  compute probability_of_success with duration 450 runs auto
11  compute probability_of_success with duration 400 runs auto
12  compute expected_charge with duration 520 runs auto
13  compute expected_fatigue with duration 520 runs auto
```

```
14
15  define queries of mission DPc:
16  compute probability_of_success with duration 1500 runs auto
17  compute probability_of_success with duration 1400 runs auto
18  compute probability_of_success with duration 1300 runs auto
19  compute expected_charge with duration 1500 runs auto
20  compute expected_fatigue with duration 1500 runs auto
21
22  define queries of mission R-DPa:
23  compute probability_of_success with duration 300 runs auto
24  compute probability_of_success with duration 250 runs auto
25  compute probability_of_success with duration 200 runs auto
26  compute expected_charge with duration 300 runs auto
27  compute expected_fatigue with duration 300 runs auto
28
29  define queries of mission R-DPb:
30  compute probability_of_success with duration 350 runs auto
31  compute probability_of_success with duration 320 runs auto
32  compute probability_of_success with duration 300 runs auto
33  compute expected_charge with duration 350 runs auto
34  compute expected_fatigue with duration 350 runs auto
35
36  define queries of mission R-DPc:
37  compute probability_of_success with duration 1500 runs auto
38  compute probability_of_success with duration 1400 runs auto
39  compute probability_of_success with duration 1300 runs auto
40  compute expected_charge with duration 1500 runs auto
41  compute expected_fatigue with duration 1500 runs auto
```

Bibliography

[1] Sara Abbaspour Asadollah, Rafia Inam, and Hans Hansson. A survey on testing for cyber physical system. In *International Conference on Testing Software and Systems*, pages 194–207, Sharjah and Dubai, UAE, 2015. Springer.

[2] Carole Adam, Wafa Johal, Damien Pellier, Humbert Fiorino, and Sylvie Pesty. Social human-robot interaction: A new cognitive and affective interaction-oriented architecture. In *Intl. Conf. on Social Robotics*, volume 9979 of *Lecture Notes in Computer Science*, pages 253–263, Kansas City, MO, USA, 2016. Springer.

[3] Masahiro Adomi, Yumi Shikauchi, and Shin Ishii. Hidden markov model for human decision process in a partially observable environment. In *Intl. Conference on Artificial Neural Networks*, pages 94–103. Springer, 2010.

[4] Gopika Ajaykumar, Maureen Steele, and Chien-Ming Huang. A survey on end-user robot programming. *ACM Computing Surveys*, 54(8):1–36, 2021.

[5] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *TCS*, 138(1):3–34, 1995.

[6] John R Anderson. ACT: A simple theory of complex cognition. *American psychologist*, 51(4):355, 1996.

[7] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[8] Dejanira Araiza-Illan, Anthony G. Pipe, and Kerstin Eder. Intelligent agent-based stimulation for testing robotic software in human-robot interactions. In *Workshop on Model-Driven Robot Software Engineering*, pages 9–16, Leipzig, Germany, 2016. ACM.

[9] Mehrnoosh Askarpour. How to formally model human in collaborative robotics. *arXiv preprint arXiv:2012.01647*, 2020.

[10] Mehrnoosh Askarpour, Dino Mandrioli, Matteo Rossi, and Federico Vicentini. Formal model of human erroneous behavior for safety analysis in collaborative robotics. *Robotics and Computer-Integrated Manufacturing*, 57:465–476, 2019.

[11] Mehrnoosh Askarpour, Matteo Rossi, and Omer Tiryakiler. Co-simulation of human-robot collaboration: from temporal logic to 3d simulation. *arXiv preprint arXiv:2007.11737*, 2020.

[12] Mehrnoosh Askarpour, Christos Tsigkanos, Claudio Menghi, Radu Calinescu, Patrizio Pelliccione, Sergio García, Ricardo Caldas, Tim J von Oertzen, Manuel Wimmer, Luca Berardinelli, et al. RoboMAX: Robotic Mission Adaptation eXemplars. In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 245–251. IEEE, 2021.

[13] American Nurses Association. Robotics and the impact on nursing practice, 2020.

[14] Markus Bajones, David Fischinger, Astrid Weiss, Paloma De La Puente, Daniel Wolf, Markus Vincze, Tobias Körtner, Markus Weninger, Konstantinos Papoutsakis, Damien Michel, et al. Results of field trials with a mobile service robot for older adults in 16 private households. *ACM Transactions on Human-Robot Interaction (THRI)*, 9(2):1–27, 2019.

[15] Stanley Bak, Omar Ali Beg, Sergiy Bogomolov, Taylor T Johnson, Luan Viet Nguyen, and Christian Schilling. Hybrid automata: from verification to implementation. *STTT*, 21(1):87–104, 2019.

[16] Chris L Baker, JB Tenenbaum, and Rebecca R Saxe. Goal inference as inverse planning. In *Proceedings of the Annual Meeting of the Cognitive Science Society, 29 (29)*, 2007.

[17] Amitav Banerjee, UB Chitnis, SL Jadhav, JS Bhawalkar, and S Chaudhury. Hypothesis testing, type i and type ii errors. *Industrial Psychiatry Journal*, 18(2):127, 2009.

[18] Kim Baraka and Manuela M Veloso. Mobile service robot state revealing through expressive lights: formalism, design, and evaluation. *International Journal of Social Robotics*, 10(1):65–92, 2018.

[19] Gerd Behrmann, Agnes Cougnard, Alexandre David, Emmanuel Fleury, Kim G Larsen, and Didier Lime. Uppaal-tiga: Time for playing games! In *Intl. Conf. on Computer Aided Verification*, pages 121–125. Springer, 2007.

[20] Marcello M Bersani, Matteo Camilli, Livia Lestingi, Raffaela Mirandola, and Matteo Rossi. Explainable human-machine teaming using model checking and interpretable machine learning. In *Accepted for presentation at Intl. Conf. on Formal Methods in Software Engineering*, 2023.

[21] Marcello M. Bersani, Matteo Camilli, Livia Lestingi, Raffaela Mirandola, Matteo Rossi, and Patrizia Scandurra. Towards better trust in human-machine teaming through explainable dependability. In *Intl. Conf. on Software Architecture Companion*, pages 86–90, 2023.

[22] Marcello M Bersani, Matteo Soldo, Claudio Menghi, Patrizio Pelliccione, and Matteo Rossi. PuRSUE-from specification of robotic environments to synthesis of controllers. *Formal Aspects of Computing*, 32(2):187–227, 2020.

[23] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[24] Andreea Bobu, Dexter RR Scobee, Jaime F Fisac, S Shankar Sastry, and Anca D Dragan. Less is more: Rethinking probabilistic models of human behavior. In *Intl. Conf. on Human-Robot Interaction*, pages 429–437, 2020.

[25] Henrik Bohnenkamp, Pedro R d'Argenio, Holger Hermanns, and J-P Katoen. Modest: A compositional modeling formalism for hard and softly timed systems. *IEEE Transactions on Software Engineering*, 32(10):812–830, 2006.

[26] Matthew L Bolton, Kylie A Molinaro, and Adam M Houser. A formal method for assessing the impact of task-based erroneous human behavior on system safety. *Reliability Engineering & System Safety*, 188:168–180, 2019.

[27] Stefano Borgo, Amedeo Cesta, Andrea Orlandini, and Alessandro Umbrico. Knowledge-based adaptive agents for manufacturing domains. *Engineering with Computers*, 35(3):755–779, 2019.

[28] Ron Breukelaar and Thomas Bäck. Using a genetic algorithm to evolve behavior in multi dimensional cellular automata: emergence of behavior. In *Genetic and Evolutionary Computation Conference*, pages 107–114, Washington DC, USA, 2005. ACM.

[29] Kyle Brown, Katherine Driggs-Campbell, and Mykel J Kochenderfer. Modeling and prediction of human driver behavior: A survey. *arXiv preprint arXiv:2006.08832*, 2020.

[30] Davide Brugali. Software product line engineering for robotics. *Software Engineering for Robotics*, pages 1–28, 2021.

[31] Cristian Calude, Frederick Kroon, and Nemanja Poznanovic. Free will is compatible with randomness. *Philosophical Inquiries*, 4(2):37–52, 2016.

[32] Javier Cámara, Radu Calinescu, Betty HC Cheng, David Garlan, Bradley Schmerl, Javier Troya, and Antonio Vallecillo. Addressing the uncertainty interaction problem in software-intensive systems: challenges and desiderata. In *Intl. Conf. on Model Driven Engineering Languages and Systems*, pages 24–30, 2022.

[33] Steven Carr, Nils Jansen, Ralf Wimmer, Jie Fu, and Ufuk Topcu. Human-in-the-loop synthesis for partially observable markov decision processes. In *Annual American Control Conference*, pages 762–769. IEEE, 2018.

[34] Rafael C Carrasco and Jose Oncina. Learning stochastic regular grammars by means of a state merging method. In *Intl. Colloquium on Grammatical Inference*, pages 139–152. Springer, 1994.

[35] George Casella, Christian P Robert, and Martin T Wells. Generalized accept-reject sampling schemes. *Lecture Notes-Monograph Series*, pages 342–347, 2004.

[36] Antonio Cerone, Peter A Lindsay, and Simon Connelly. Formal analysis of human-computer interaction using model-checking. In *Intl. Conf. on Software Engineering and Formal Methods*, pages 352–361. IEEE, 2005.

[37] Taolue Chen, Vojtěch Forejt, Marta Kwiatkowska, David Parker, and Aistis Simaitis. PRISM-games: A model checker for stochastic multi-player games. In *Intl. Conf. on TOOLS and Algorithms for the Construction and Analysis of Systems*, pages 185–191, Rome, Italy, 2013. Springer.

[38] Yushan Chen, Jana Tumova, and Calin Belta. LTL robot motion control based on automata learning of environmental dynamics. In *IEEE International Conference on Robotics and Automation*, pages 5177–5182, St. Paul, Minnesota, USA, 2012. IEEE.

[39] Taha Chettibi, Moussa Haddad, HE Lehtihet, and Wisama Khalil. Suboptimal trajectory generation for industrial robots using trapezoidal velocity profiles. In *IROS*, pages 729–735. IEEE, 2006.

[40] Amir Khorrami Chokami, Mauro Gasparini, and Roberto Merletti. Identification of periodic bursts in surface EMG: Applications to the erector spinae muscles of sitting violin players. *Biomedical Signal Processing and Control*, 65:102369, 2021.

[41] Huai Chuangfeng, Liu Pingan, and Jia Xueyan. Measurement and analysis for lithium battery of high-rate discharge performance. *Procedia Engineering*, 15:2619–2623, 2011.

[42] Federico Ciccozzi, Davide Di Ruscio, Ivano Malavolta, and Patrizio Pelliccione. Adopting MDE for specifying and executing civilian missions of mobile multi-robot systems. *IEEE Access*, 4:6451–6466, 2016.

[43] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *International conference on computer aided verification*, pages 359–364, Copenhagen, Denmark, 2002. Springer, Springer.

[44] Charles J Clopper and Egon S Pearson. The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, 26(4):404–413, 1934.

[45] Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marius Mikučionis, and Jakob Haahr Taankvist. Uppaal stratego. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 206–211. Springer, 2015.

[46] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal SMC tutorial. *STTT*, 17(4):397–415, 2015.

[47] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, Danny Bøgsted Poulsen, Jonas Van Vliet, and Zheng Wang. Statistical model checking for networks of priced timed automata. In *Intl. Conf. on Formal Modeling and Analisys of Timed Systems*, pages 80–96. Springer, 2011.

[48] Fabio De Felice, Federico Zomparelli, and Antonella Petrillo. Functional Human Reliability Analysis: A Systems Engineering Perspective. In *CIISE*, pages 23–29, 2017.

[49] Asaf Degani and Michael Heymann. Formal verification of human-automation interaction. *Human factors*, 44(1):28–43, 2002.

[50] Peter Detzner, Thomas Kirks, and Jana Jost. A novel task language for natural interaction in human-robot systems for warehouse logistics. In *Intl. Conf. on Computer Science & Education*, pages 725–730, Toronto, ON, Canada, 2019. IEEE.

[51] Valentina Di Pasquale, Raffaele Iannone, Salvatore Miranda, and Stefano Riemma. An overview of human reliability analysis techniques in manufacturing operations. *Operations management*, pages 221–240, 2013.

[52] Hao Ding, Jakob Heyn, Bjoern Matthias, and Harald Staab. Structured collaborative behavior of industrial robots in mixed human-robot environments. In *Intl. Conf. on Automation Science and Engineering*, pages 1101–1106, Madison, WI, USA, 2013. IEEE.

[53] Hao Ding, Malte Schipper, and Bjoern Matthias. Collaborative behavior design of industrial robots for multiple human-robot collaboration. In *IEEE International Symposium on Robotics*, pages 1–6, Seoul, Korea (South), 2013. IEEE.

[54] Ed M Dougherty and Joseph R Fragola. *Human Reliability Analysis*. New York, NY; John Wiley and Sons Inc., 1988.

[55] Vibekananda Dutta and Teresa Zielinska. Predicting the intention of human activities for real-time human-robot interaction (HRI). In *Int. Conf. on Social Robotics*, volume 9979 of *Lecture Notes in Computer Science*, pages 723–734, Kansas City, MO, USA, 2016. Springer.

[56] Johan Eddeland, Javier Gil Cepeda, Rick Fransen, Sajed Miremadi, Martin Fabian, and Knut Åkesson. Automated mode coverage analysis for cyber-physical systems using hybrid automata. *IFAC-PapersOnLine*, 50(1):9260–9265, 2017.

[57] Klaus Ehrlenspiel, Alfons Kiewert, Udo Lindemann, and Mahendra S Hundal. *Cost-efficient design*, volume 544. Springer, 2007.

[58] Juhan Ernits, Evelin Halling, Gert Kanter, and Jüri Vain. Model-based integration testing of ros packages: A mobile robot case study. In *European Conference on Mobile Robots (ECMR)*, pages 1–7. IEEE, 2015.

[59] Javier Esparza, Martin Leucker, and Maximilian Schlund. Learning workflow petri nets. *Fundamenta Informaticae*, 113(3-4):205–228, 2011.

[60] euRobotics. Robotics 2020 Multi-Annual Roadmap. `https://eu-robotics.net/divi_overlay/roadmap/`, 2020.

[61] Muhammad Fahad, Zhuo Chen, and Yi Guo. Learning how pedestrians navigate: A deep inverse reinforcement learning approach. In *IROS*, pages 819–826. IEEE, 2018.

[62] Sergio Feo-Arenis, Milan Vujinović, and Bernd Westphal. On implementable timed automata. In *Intl. Conf. on Formal Techniques for Distributed Objects, Components, and Systems*, pages 78–95. Springer, 2020.

[63] Cameron Finucane, Gangyuan Jing, and Hadas Kress-Gazit. LTLMoP: Experimenting with language, temporal logic and robot control. In *Intl. Conf. on Intelligent Robots and Systems*, pages 1988–1993, Taipei, Taiwan, 2010. IEEE.

[64] Association for Advancing Automation. Robots and healthcare saving lives together, 2022.

[65] Peter Forbrig and Alexandru-Nicolae Bundea. Modelling the collaboration of a patient and an assisting humanoid robot during training tasks. In *Human-Computer Interaction*, volume 12182 of *Lecture Notes in Computer Science*, pages 592–602, Copenhagen, Denmark, 2020. Springer.

[66] Peter Forbrig, Anke Dittmar, and Mathias Kühn. A textual domain specific language for task models: Generating code for CoTaL, CTTE, and HAMSTERS. In *Symposium on Engineering Interactive Computing Systems*, pages 5:1–5:6, Paris, France, 2018. ACM.

[67] Mohammed Foughali, Félix Ingrand, and Cristina Seceleanu. Statistical model checking of complex robotic systems. In *International Symposium on Model Checking Software*, volume 11636 of *Lecture Notes in Computer Science*, pages 114–134, Beijing, China, 2019. Springer.

[68] Marlena R Fraune, Steven Sherrin, Selma Šabanović, and Eliot R Smith. Is human-robot interaction more competitive between groups than between individuals? In *Intl. Conf. on Human-Robot Interaction*, pages 104–113. IEEE, 2019.

[69] Fraunhofer Institute for Manufacturing Engineering and Automation. EFFIROB: Economic feasibility studies on innovative service robot applications, 2010.

[70] Fraunhofer Institute for Manufacturing Engineering and Automation. Care-O-bot 3 Application Scenarios. `https://www.care-o-bot.de/en/care-o-bot-3/application.html`, 2012.

[71] Carl Benedikt Frey and Michael A Osborne. The future of employment: How susceptible are jobs to computerisation? *Technological forecasting and social change*, 114:254–280, 2017.

[72] Carlo A Furia, Dino Mandrioli, Angelo Morzenti, and Matteo Rossi. *Modeling time in computing*. Springer Science & Business Media, 2012.

[73] Mitchell H Gail and Sylvan B Green. Critical values for the one-sided two-sample kolmogorov-smirnov statistic. *Journal of the American Statistical Association*, 71(355):757–760, 1976.

[74] Paul Gainer, Clare Dixon, Kerstin Dautenhahn, Michael Fisher, Ullrich Hustadt, Joe Saunders, and Matt Webster. Cruton: Automatic verification of a robotic assistant's behaviours. In *Critical Systems: Formal Methods and Automated Verification*, pages 119–133. Springer, Turin, Italy, 2017.

[75] Rinat R Galin, Roman V Meshcheryakov, and Mark V Mamchenko. Analysis of intersection of working areas within the human-robot interaction in a shared workspace. In *Proceedings of the Computational Methods in Systems and Software*, pages 749–759, Czech Republic, 2021. Springer.

[76] Sergio García, Patrizio Pelliccione, Claudio Menghi, Thorsten Berger, and Tomás Bures. PROMISE: high-level mission specification for multiple robots. In *Intl. Conf. on Software Engineering*, pages 5–8, Seoul, South Korea, 2020. ACM.

[77] Sergio García, Daniel Strüber, Davide Brugali, Thorsten Berger, and Patrizio Pelliccione. Robotics software engineering: A perspective from the service robotics domain. In *ES-EC/FSE*, pages 593–604, USA, 2020. ACM.

[78] Sergio García, Daniel Strüber, Davide Brugali, Alessandro Di Fava, Philipp Schillinger, Patrizio Pelliccione, and Thorsten Berger. Variability modeling of service robots: Experiences and challenges. In *Intl. Workshop on Variability Modelling of Software-Intensive Systems*, pages 1–6, 2019.

[79] Ivan Gavran, Ortwin Mailahn, Rainer Müller, Richard Peifer, and Damien Zufferey. Tool: accessible automated reasoning for human robot collaboration. In *Intl. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 44–56, 2018.

[80] Carlo Ghezzi, Mauro Pezzè, Michele Sama, and Giordano Tamburrelli. Mining behavior models from user-intensive web applications. In *Intl. Conf. on Software Engineering*, pages 277–287, 2014.

[81] ZS Givi, Mohamad Y Jaber, and W Patrick Neumann. Modelling worker reliability with learning and fatigue. *Applied Mathematical Modelling*, 39(17):5186–5199, 2015.

[82] Mario Gleirscher and Diego Marmsoler. Formal methods in dependable systems engineering: a survey of professionals from Europe and North America. *Empirical Software Engineering*, 25(6):4473–4546, 2020.

[83] GlobeNewswire. Global healthcare assistive robot market, 2022.

[84] Teofilo Gonzalez, Sartaj Sahni, and William R. Franta. An efficient algorithm for the kolmogorov-smirnov and lilliefors tests. *ACM Transactions on Mathematical Software (TOMS)*, 3(1):60–64, 1977.

[85] Ulf Grenander. Stochastic processes and statistical inference. *Arkiv för matematik*, 1(3):195–277, 1950.

[86] Raju Halder, José Proença, Nuno Macedo, and André Santos. Formal verification of ROS-based robotic applications using timed-automata. In *Intl. FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, pages 44–50. IEEE, 2017.

[87] Léo Henry, Thierry Jéron, and Nicolas Markey. Active learning of timed automata with unobservable resets. In *Intl. Conf. on Formal Modeling and Analysis of Timed Systems*, pages 144–160. Springer, 2020.

[88] Benjamin Herd, Simon Miles, Peter McBurney, and Michael Luck. Quantitative analysis of multiagent systems through statistical model checking. In *Int. Workshop on Engineering Multi-Agent Systems*, pages 109–130. Springer, 2015.

[89] Sahar Heydaryan, Joel Suaza Bedolla, and Giovanni Belingardi. Safety design and development of a human-robot collaboration assembly process in the automotive industry. *Applied Sciences*, 8(3):344, 2018.

[90] John L Hodges. The significance probability of the Smirnov two-sample test. *Arkiv för Matematik*, 3(5):469–486, 1958.

[91] Erik Hollnagel. The phenotype of erroneous actions: Implications for HCI design. *Human-computer interaction and complex systems*, pages 73–121, 1991.

[92] Erik Hollnagel. The phenotype of erroneous actions. *International Journal of Man-Machine Studies*, 39(1):1–32, 1993.

[93] Erik Hollnagel. *Cognitive reliability and error analysis method (CREAM)*. Elsevier, 1998.

[94] Daniel V Holt and Magda Osman. Approaches to cognitive modeling in dynamic systems control. *Frontiers in Psychology*, 8:2032, 2017.

[95] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[96] Lin-Xiu Hou, Ran Liu, Hu-Chen Liu, and Shan Jiang. Two decades on human reliability analysis: a bibliometric analysis and literature review. *Annals of Nuclear Energy*, 151:107969, 2021.

[97] Falk Howar and Bernhard Steffen. Active automata learning in practice. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits*, pages 123–148. Springer, 2018.

[98] Bin Hu and Jing Chen. Optimal task allocation for human–machine collaborative manufacturing systems. *IEEE Robotics and Automation Letters*, 2(4):1933–1940, 2017.

[99] IFR International Federation of Robotics. Case Studies - Service Robots. `https://ifr.org/case-studies/service-robots-case-studies`, 2018.

[100] Daniel Imbeau, Bruno Farbos, et al. Percentile values for determining maximum endurance times for static muscular work. *Intl. Journ. of Industrial Ergonomics*, 36(2):99–108, 2006.

[101] Jairo Inga, Florian Köpf, Michael Flad, and Sören Hohmann. Individual human behavior identification using an inverse reinforcement learning method. In *IEEE SMC*, pages 99–104, 2017.

[102] Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: a redundancy-free approach to active automata learning. In *Intl. Conf. on Runtime Verification*, pages 307–322. Springer, 2014.

[103] ISO 13482. *Robots and robotic devices - Safety requirements for personal care robots*. ISO, 2014.

[104] ISO/PAS 21448:2019. *Road vehicles - Safety of the intended functionality*. ISO/PAS, 2019.

[105] Neziha Jaouedi, Noureddine Boujnah, and Med Salim Bouhlel. A new hybrid deep learning model for human action recognition. *Journal of King Saud University-Computer and Information Sciences*, 32(4):447–453, 2020.

[106] Cyrille Jegourel, Kim G Larsen, Axel Legay, Marius Mikučionis, Danny Bøgsted Poulsen, and Sean Sedwards. Importance sampling for stochastic timed automata. In *Dependable Software Engineering: Theories, Tools, and Applications*, pages 163–178. Springer, 2016.

[107] Bonnie E John and David E Kieras. The GOMS family of user interface analysis techniques: Comparison and contrast. *ACM Transactions on Computer-Human Interaction*, 3(4):320–351, 1996.

[108] Sebastian Junges, Nils Jansen, Joost-Pieter Katoen, and Ufuk Topcu. Probabilistic model checking for complex cognitive tasks–a case study in human-robot interaction. *arXiv preprint arXiv:1610.09409*, 2016.

[109] Daniel Kahneman and Vernon Smith. Foundations of behavioral and experimental economics. *Nobel Prize in Economics Documents*, 1(7), 2002.

[110] Hyun Gu Kang and Jonathan B Dingwell. Differential changes with age in multiscale entropy of electromyography signals from leg muscles during treadmill walking. *PloS one*, 11(8):e0162034, 2016.

[111] Gert Kanter and Jüri Vain. Model-based testing of autonomous robots using testit. *Journal of Reliable Intelligent Environments*, pages 1–16, 2020.

[112] Joost-Pieter Katoen, Ivan S Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N Jansen. The ins and outs of the probabilistic model checker mrmc. *Performance evaluation*, 68(2):90–104, 2011.

[113] Eamonn J Keogh and Michael J Pazzani. Derivative dynamic time warping. In *Intl. Conf. on Data Mining*, pages 1–11. SIAM, 2001.

[114] Alaa Khamis, Ahmed Hussein, and Ahmed Elmogy. Multi-robot task allocation: A review of the state-of-the-art. *Cooperative robots and sensor networks*, pages 31–51, 2015.

[115] Man Cheol Kim, Poong Hyun Seong, and Erik Hollnagel. A probabilistic approach for determining the control mode in CREAM. *Reliability Engineering & System Safety*, 91(2):191–199, 2006.

[116] Yochan Kim and Wan C Yoon. Generating task-oriented interactions of service robots. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 44(8):981–994, 2014.

[117] Steffen Knoop, Michael Pardowitz, and Rüdiger Dillmann. Automatic robot programming from learned abstract task knowledge. In *Intl. Conf. on Intelligent Robots and Systems*, pages 1651–1657, California, USA, 2007. IEEE.

[118] Stephan Konz. Work/rest: Part ii-the scientific basis (knowledge base) for the guide 1. *EGPS*, 1(401):38, 2000.

[119] Hadas Kress-Gazit, Tichakorn Wongpiromsarn, and Ufuk Topcu. Correct, reactive, high-level robot control. *IEEE Robotics & Automation Magazine*, 18(3):65–74, 2011.

[120] Lars Kunze, Tobias Roehm, and Michael Beetz. Towards semantic robot description languages. In *Intl. Conf. on Robotics and Automation*, pages 5589–5595, Shanghai, China, 2011. IEEE.

[121] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591, Snowbird, UT, USA, 2011. Springer.

[122] John E Laird. *The Soar cognitive architecture*. MIT press, 2019.

[123] Kim G Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Int. J. on Softw. Tools for Tech. Transf.*, 1(1-2):134–152, 1997.

[124] Livia Lestingi. Design-Time Analysis Module. `https://github.com/LesLivia/hri_designtime`, 2020.

[125] Livia Lestingi, Mehrnoosh Askarpour, Marcello M Bersani, and Matteo Rossi. Formal verification of human-robot interaction in healthcare scenarios. In *SEFM*, pages 303–324. Springer, 2020.

[126] Livia Lestingi, Mehrnoosh Askarpour, Marcello M Bersani, and Matteo Rossi. A model-driven approach for the formal analysis of human-robot interaction scenarios. In *IEEE SMC*, pages 1907–1914, 2020.

[127] Livia Lestingi, Mehrnoosh Askarpour, Marcello M Bersani, and Matteo Rossi. Statistical model checking of human-robot interaction scenarios. In *First Workshop on Agents and Robots for reliable Engineered Autonomy*, pages 9–17, 2020.

[128] Livia Lestingi, Mehrnoosh Askarpour, Marcello M. Bersani, and Matteo Rossi. A deployment framework for formally verified human-robot interactions. *IEEE Access*, 9:136616–136635, 2021.

[129] Livia Lestingi, Marcello M Bersani, and Matteo Rossi. Model-driven development of service robot applications dealing with uncertain human behavior. *IEEE Intelligent Systems*, 2022.

[130] Livia Lestingi, Cristian Sbrolli, Pasquale Scarmozzino, Giorgio Romeo, Marcello M Bersani, and Matteo Rossi. Formal modeling and verification of multi-robot interactive scenarios in service settings. In *Intl. Conf. on Formal Methods in Software Engineering*, pages 80–90, 2022.

[131] Livia Lestingi, Davide Zerla, Marcello M Bersani, and Matteo Rossi. Specification, stochastic modeling and analysis of interactive service robotic applications. *Robotics and Autonomous Systems*, page 104387, 2023.

[132] Lestingi, Livia. HRI Deployment. `https://github.com/LesLivia/hri_deployment`, 2020.

[133] Lestingi, Livia. L*_SHA Implementation. `https://github.com/LesLivia/lsha`, 2022.

[134] Congbo Li, Shaoqing Wu, Qian Yi, Xikun Zhao, and Longguo Cui. A cutting parameter energy-saving optimization method considering tool wear for multi-feature parts batch processing. *The International Journal of Advanced Manufacturing Technology*, 121(7):4941–4960, 2022.

[135] Daoliang Li, Ying Wang, Jinxing Wang, Cong Wang, and Yanqing Duan. Recent advances in sensor fault diagnosis: A review. *Sensors and Actuators A: Physical*, 309:111990, 2020.

[136] Kai Li, Quan Liu, Wenjun Xu, Jiayi Liu, Zude Zhou, and Hao Feng. Sequence planning considering human fatigue for human-robot collaboration in disassembly. *Procedia CIRP*, 83:95–104, 2019.

[137] Kun Li and Joel W Burdick. Human motion analysis in medical robotics via high-dimensional inverse reinforcement learning. *Intl. Journal of Robotics Research*, 39(5):568–585, 2020.

[138] Nianyu Li, Christos Tsigkanos, Zhi Jin, Zhenjiang Hu, and Carlo Ghezzi. Early validation of cyber–physical space systems via multi-concerns integration. *Journal of Systems and Software*, 170:110742, 2020.

[139] L_ Lindstrom, ROLAND Kadefors, and INGEMAR Petersen. An electromyographic index for localized muscle fatigue. *Journal of Applied Physiology*, 43(4):750–754, 1977.

[140] Bin Liu, Liang Ma, Chi Chen, and Zhanwu Zhang. Experimental validation of a subject-specific maximum endurance time model. *Ergonomics*, 61(6):806–817, 2018.

[141] Jing Z Liu, Robert W Brown, and Guang H Yue. A dynamical model of muscle activation, fatigue, and recovery. *Biophysical Journal*, 82(5):2344–2359, 2002.

[142] Jia Lu, Minh Nguyen, and Wei Qi Yan. Deep learning methods for human behavior recognition. In *Intl. Conf. on Image and Vision Computing New Zealand*, pages 1–6. IEEE, 2020.

[143] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon, and Michael Fisher. Formal specification and verification of autonomous robotic systems: A survey. *ACM Comput. Surv.*, 52(5):100:1–100:41, 2019.

[144] Matthias Lutz, Dennis Stampfer, Alex Lotz, and Christian Schlegel. Service robot control architectures for flexible and robust real-world task execution: Best practices and patterns. In *INFORMATIK 2014*, volume P-232 of *LNI*, pages 1295–1306, Stuttgart, Germany, 2014. GI.

[145] Melinda Lyons, Sally Adams, Maria Woloshynowych, and Charles Vincent. Human reliability analysis in healthcare: a review of techniques. *Intl. Jnl. of Risk & Safety in Medicine*, 16(4):223–237, 2004.

[146] Alexander Maier. Online passive learning of timed automata for cyber-physical production systems. In *International Conference on Industrial Informatics*, pages 60–66. IEEE, 2014.

[147] Avinash Malik, Partha S Roop, Sidharta Andalam, Mark Trew, and Michael Mendler. Modular compilation of hybrid systems for emulation and large scale simulation. *Trans. on Embedded Computing Systems*, 16(5s):1–21, 2017.

[148] Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D Nielsen, Kim G Larsen, and Brian Nielsen. Learning deterministic probabilistic automata from a model checking perspective. *Machine Learning*, 105(2):255–299, 2016.

[149] Julian N Marewski and Lael J Schooler. Cognitive niches: an ecological model of strategy selection. *Psychological Review*, 118(3):393, 2011.

[150] Tiziana Margaria, Oliver Niese, Harald Raffelt, and Bernhard Steffen. Efficient test-based model generation for legacy reactive systems. In *Intl. High-Level Design Validation and Test Workshop*, pages 95–100. IEEE, 2004.

[151] George Mason, Radu Calinescu, Daniel Kudenko, and Alec Banks. Assurance in reinforcement learning using quantitative verification. *Advances in hybridization of intelligent methods*, 85:71, 2017.

[152] Catharine LR McGhan, Ali Nasir, and Ella M Atkins. Human intent prediction using markov decision processes. *Journal of Aerospace Information Systems*, 12(5):393–397, 2015.

[153] Richard D McKelvey and Thomas R Palfrey. Quantal response equilibria for normal form games. *Games and economic behavior*, 10(1):6–38, 1995.

[154] Ramy Medhat, S Ramesh, Borzoo Bonakdarpour, and Sebastian Fischmeister. A framework for mining hybrid automata from input/output traces. In *EMSOFT*, pages 177–186. IEEE, 2015.

[155] Claudio Menghi, Christos Tsigkanos, Patrizio Pelliccione, Carlo Ghezzi, and Thorsten Berger. Specification patterns for robotic missions. *IEEE Trans. Software Eng.*, 47(10):2208–2224, 2021.

[156] R Merletti, LR Lo Conte, and C Orizio. Indices of muscle fatigue. *J. of Electromyography and Kinesiology*, 1(1):20–33, 1991.

[157] Maik Merten, Falk Howar, Bernhard Steffen, and Tiziana Margaria. Automata learning with on-the-fly direct hypothesis construction. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 248–260. Springer, 2011.

[158] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, Jon Timmis, and Jim Woodcock. RoboChart: modelling and verification of the functional behaviour of robotic applications. *Software & Systems Modeling*, 18(5):3097–3149, 2019.

[159] Levente Molnar and Sandor M Veres. Hybrid automata dicretising agents for formal modelling of robots. *IFAC Proceedings Volumes*, 44(1):49–54, 2011.

[160] Ahmed Ashraf Morgan, Jordan Abdi, Mohammed AQ Syed, Ghita El Kohen, Phillip Barlow, and Marcela P Vizcaychipi. Robots in healthcare: a scoping review. *Current Robotics Reports*, pages 1–10, 2022.

[161] Tuong Nguyen, M Reynolds, R Kandaswamy, et al. Emerging technologies and trends impact radar: 2021. *Gartner Research Notes. Available at*, 2021.

[162] Stefanos Nikolaidis, Swaprava Nath, Ariel D Procaccia, and Siddhartha Srinivasa. Game-theoretic modeling of human adaptation in human-robot collaboration. In *Intl. Conf. on Human-Robot Interaction*, pages 323–331, 2017.

[163] Arne Nordmann, Nico Hochgeschwender, and Sebastian Wrede. A survey on domain-specific languages in robotics. In *Simulation, Modeling, and Programming for Autonomous Robots*, volume 8810 of *Lecture Notes in Computer Science*, pages 195–206, Bergamo, Italy, 2014. Springer.

[164] Fabrice R. Noreils and Raja Chatila. Plan execution monitoring and control architecture for mobile robots. *IEEE Trans. Robotics Autom.*, 11(2):255–266, 1995.

[165] Jason M O'Kane. A gentle introduction to ROS, 2014.

[166] José Oncina and Pedro Garcia. Inferring regular languages in polynomial updated time. In *Pattern recognition and image analysis*, pages 49–61. World Scientific, 1992.

[167] Alberto De Obeso Orendain and Sharon Wood. An account of cognitive flexibility and inflexibility for a complex dynamic task. In *Intl. Conference on Cognitive Modeling*, pages 49–54, 2012.

[168] Anshul Paigwar, Eduard Baranov, Alessandro Renzaglia, Christian Laugier, and Axel Legay. Probabilistic collision risk estimation for autonomous driving: Validation via statistical model checking. In *IEEE Intelligent Vehicles Symposium*, pages 737–743, Las Vegas, NV, USA, 2020. IEEE.

[169] Emanuel Parzen. On choosing an estimate of the spectral density function of a stationary time series. *The Annals of Mathematical Statistics*, 28(4):921–932, 1957.

[170] Fabio Paterno, Cristiano Mancini, and Silvia Meniconi. ConcurTaskTrees: A diagrammatic notation for specifying task models. In *Human-computer interaction*, pages 362–369. Springer, 1997.

[171] Philip Payne, Marcelo Lopetegui, and Sean Yu. A review of clinical workflow studies and methods. In *Cog. Inf.*, pages 47–61. Springer, 2019.

[172] Luka Peternel, Nikos Tsagarakis, Darwin Caldwell, and Arash Ajoudani. Robot adaptation to human physical fatigue in human–robot co-manipulation. *Aut. Rob.*, 42(5):1011–1021, 2018.

[173] Nhathai Phan, Dejing Dou, Hao Wang, David Kil, and Brigitte Piniewski. Ontology-based deep learning for human behavior prediction with explanations in health social networks. *Information sciences*, 384:298–313, 2017.

[174] S Pocock, MD Harrison, PC Wright, and P Johnson. THEA: A technique for human error assessment early in design. In *Intl. Conf. on Human Computer Interaction*. Newcastle University, 2001.

[175] David Porfirio, Allison Sauppé, Aws Albarghouthi, and Bilge Mutlu. Authoring and verifying human-robot interactions. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, pages 75–86, 2018.

[176] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, volume 3, page 5, 2009.

[177] Joao Quintas, Gonçalo S Martins, Luis Santos, Paulo Menezes, and Jorge Dias. Toward a context-aware human–robot interaction framework based on cognitive development. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 49(1):227–237, 2018.

[178] Michael Melholt Quottrup, Thomas Bak, and Roozbeh Izadi-Zamanabadi. Multi-robot planning: a timed automata approach. In *IEEE International Conference on Robotics and Automation*, pages 4417–4422, New Orleans, LA, USA, 2004. IEEE.

[179] Harald Raffelt, Bernhard Steffen, and Therese Berg. LearnLib: A library for automata learning and experimentation. In *Intl. Workshop on Formal methods for Industrial Critical Systems*, pages 62–71, 2005.

[180] Vasumathi Raman, Bingxin Xu, and Hadas Kress-Gazit. Avoiding forgetfulness: Structured english specifications for high-level robot control with implicit memory. In *Intl. Conf. on Intelligent Robots and Systems*, pages 1233–1238, Vilamoura, Algarve, Portugal, 2012. IEEE.

[181] Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Workshop on Modelling Autonomous Agents in a Multi-Agent World*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55, Eindhoven, The Netherlands, 1996. Springer.

[182] Adil Rasheed, Omer San, and Trond Kvamsdal. Digital twin: Values, challenges and enablers from a modeling perspective. *IEEE Access*, 8:21980–22012, 2020.

[183] James Reason. Actions not as planned: The price of automatisation. *Aspects of consciousness*, 1:67–89, 1979.

[184] Merletti Roberto, Dario Farina, Marco Gazzoni, and Maria Pia Schieroni. Effect of age on muscle functions investigated with surface electromyography. *Muscle & nerve*, 25(1):65–76, 2002.

[185] E. Rohmer, S. P. N. Singh, and M. Freese. CoppeliaSim (formerly V-REP): a versatile and scalable robot simulation framework. In *Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2013.

[186] Murray Rosenblatt. Remarks on a multivariate transformation. *The Annals of Mathematical Statistics*, 23(3):470–472, 1952.

[187] Ariel Rosenfeld and Sarit Kraus. Predicting human decision-making: From prediction to action. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 12(1):1–150, 2018.

[188] Andrey Rudenko, Luigi Palmieri, Michael Herman, Kris M Kitani, Dariu M Gavrila, and Kai O Arras. Human motion trajectory prediction: A survey. *Intl. Journal of Robotics Research*, 39(8):895–935, 2020.

[189] Patrick Rueckert, Sophie Muenkewarf, and Kirsten Tracht. Human-in-the-loop simulation for virtual commissioning of human-robot-collaboration. *Procedia CIRP*, 88:229–233, 2020.

[190] Rimvydas Rukšėnas, Paul Curzon, Ann Blandford, and Jonathan Back. Combining human error verification and timing analysis: a case study on an infusion pump. *Formal Aspects of Computing*, 26(5):1033–1076, 2014.

[191] Iman Saberi, Fathiyeh Faghih, and Farzad Sobhi Bavil. A passive online technique for learning hybrid automata from input/output traces. *arXiv preprint arXiv:2101.07053*, 2021.

[192] Sven R Schmidt-Rohr, Martin Losch, and Rudiger Dillmann. Human and robot behavior modeling for probabilistic cognition of an autonomous service robot. In *Intl. Symp. on Robot and Human Interactive Communication*, pages 635–640. IEEE, 2008.

[193] Steve Schneider. *Concurrent and real-time systems: the CSP approach*. John Wiley & Sons, 1999.

[194] Fritz Scholz. Confidence bounds and intervals for parameters relating to the binomial, negative binomial, poisson and hypergeometric distributions with applications to rare events. *2008*, 2008.

[195] Koushik Sen, Mahesh Viswanathan, and Gul Agha. On statistical model checking of stochastic systems. In *Computer Aided Verification*, pages 266–280. Springer, 2005.

[196] Koushik Sen, Mahesh Viswanathan, and Gul Agha. Vesta: A statistical model-checker and analyzer for probabilistic systems. In *Second International Conference on the Quantitative Evaluation of Systems (QEST'05)*, pages 251–252. IEEE, 2005.

[197] Dongmin Shin, Richard A Wysk, and Ling Rothrock. Formal model of human material-handling tasks for control of manufacturing systems. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 36(4):685–696, 2006.

[198] Bernard W Silverman. *Density estimation for statistics and data analysis*. Routledge, 2018.

[199] Stanislaw Solnik, Patrick Rider, Ken Steinweg, Paul DeVita, and Tibor Hortobágyi. Teager–kaiser energy operator signal conditioning improves EMG onset detection. *European Journal of Applied Physiology*, 110(3):489–498, 2010.

[200] Miriam García Soto, Thomas A Henzinger, Christian Schilling, and Luka Zeleznik. Membership-based synthesis of linear hybrid automata. In *Intl. Conf. on Computer Aided Verification*, pages 297–314. Springer, 2019.

[201] Rajan Srinivasan. *Importance sampling: Applications in communications and detection*. Springer Science & Business Media, 2002.

[202] Dale O Stahl II and Paul W Wilson. Experimental evidence on players' models of other players. *Journal of economic behavior & organization*, 25(3):309–327, 1994.

[203] Richard Stocker, Louise A. Dennis, Clare Dixon, and Michael Fisher. Verifying brahms human-robot teamwork models. In *Logics in Artificial Intelligence*, volume 7519 of *Lecture Notes in Computer Science*, pages 385–397, Toulouse, France, 2012. Springer.

[204] Mark A Sujan, David Embrey, and Huayi Huang. On the application of human reliability analysis in healthcare: opportunities and challenges. *Reliability Engineering & System Safety*, 194:106189, 2020.

[205] Ola Svenson. Decision making and the search for fundamental psychological regularities: What can be learned from a process perspective? *Organizational behavior and human decision processes*, 65(3):252–267, 1996.

[206] Alan D Swain and Henry E Guttmann. Handbook of human-reliability analysis with emphasis on nuclear power plant applications. Final report. Technical report, Sandia National Labs., Albuquerque, NM (USA), 1983.

[207] Martin Tappler, Bernhard K Aichernig, Giovanni Bacci, Maria Eichlseder, and Kim G Larsen. L*-based learning of markov decision processes. In *Intl. Symposium on Formal Methods*, pages 651–669. Springer, 2019.

[208] Danilo Tardioli, Ramviyas Parasuraman, and Petter Ögren. Pound: A multi-master ros node for reducing delay and jitter in wireless multi-robot networks. *Robotics and Autonomous Systems*, 111:73–87, 2019.

[209] Moritz Tenorth and Michael Beetz. KnowRob: A knowledge processing infrastructure for cognition-enabled robots. *Int. J. Robotics Res.*, 32(5):566–590, 2013.

[210] Moritz Tenorth, Fernando De la Torre, and Michael Beetz. Learning probability distributions over partially-ordered human everyday activities. In *Intl. Conf. on Robotics and Automation*, pages 4539–4544. IEEE, 2013.

[211] Olivier Tremblay, Louis-A Dessaint, and Abdel-Illah Dekkiche. A generic battery model for the dynamic simulation of hybrid electric vehicles. In *Vehicle Power and Propulsion Conference*, pages 284–289. IEEE, 2007.

[212] Panagiota Tsarouchi, Sotiris Makris, and George Chryssolouris. Human–robot interaction review and challenges on task planning and programming. *Intl. Jnl. of Computer Integrated Manufacturing*, 29(8):916–931, 2016.

[213] Federico Vicentini, Mehrnoosh Askarpour, Matteo G Rossi, and Dino Mandrioli. Safety assessment of collaborative robotics through automated formal verification. *IEEE Transactions on Robotics*, 2019.

[214] 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, 2018.

[215] Matt Webster, Clare Dixon, Michael Fisher, Maha Salem, Joe Saunders, Kheng Lee Koay, and Kerstin Dautenhahn. Formal verification of an autonomous personal robotic assistant. In *AAAI Spring Symposia*, pages 1–6, Palo Alto, California, USA, 2014. AAAI Press.

[216] Matt Webster, David Western, Dejanira Araiza-Illan, Clare Dixon, Kerstin Eder, Michael Fisher, and Anthony G Pipe. A corroborative approach to verification and validation of human–robot teams. *Intl. Journ. of Robotics Research*, 39(1):73–99, 2020.

[217] JC Williams. A data-based method for assessing and reducing human error to improve operational performance. In *Conference Record for 1988 IEEE Fourth Conference on Human Factors and Power Plants,*, pages 436–450. IEEE, 1988.

[218] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM computing surveys (CSUR)*, 41(4):1–36, 2009.

[219] Xiaodong Yang, Omar Ali Beg, Matthew Kenigsberg, and Taylor T Johnson. A framework for identification and validation of affine hybrid automata from input-output traces. *ACM Transactions on Cyber-Physical Systems*, 6(2):1–24, 2022.

[220] Kangfeng Ye, Ana Cavalcanti, Simon Foster, Alvaro Miyazawa, and Jim Woodcock. Probabilistic modelling and verification using RoboChart and PRISM. *Software and Systems Modeling*, pages 1–50, 2021.

[221] Kohzoh Yoshino, Tomoko Motoshige, Tsutomu Araki, and Katsunori Matsuoka. Effect of prolonged free-walking fatigue on gait and physiological rhythm. *Journal of biomechanics*, 37(8):1271–1280, 2004.

[222] Hakan Lorens Samir Younes. *Verification and planning for stochastic processes with asynchronous events*. Carnegie Mellon University, 2004.

[223] Andrea Maria Zanchettin, Elio Lotano, and Paolo Rocco. Collaborative robot assistant for the ergonomic manipulation of cumbersome objects. In *IROS*, pages 6729–6734. IEEE, 2019.

[224] Davide Zerla. A dsl for formally verified interactive robotic applications in service settings. Master's thesis, Politecnico di Milano, 2022.

[225] Kai Zhang and Xiaobo Li. Human-robot team coordination that considers human fatigue. *Int. J. of Adv. Rob. Syst.*, 11(6):91, 2014.

[226] Ying Zhang, Guohui Tian, Senyan Zhang, and Cici Li. A knowledge-based approach for multiagent collaboration in smart home: From activity recognition to guidance service. *IEEE Trans. on Instrumentation and Measurement*, 69(2):317–329, 2019.

[227] Kuanhao Zheng, Dylan F Glas, Takayuki Kanda, Hiroshi Ishiguro, and Norihiro Hagita. Designing and implementing a human–robot team for social interactions. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 43(4):843–859, 2013.

[228] Yuchen Zhou, Dipankar Maity, and John S. Baras. Timed automata approach for motion planning using metric interval temporal logic. In *European Control Conference*, pages 690–695, Aalborg, Denmark, 2016. IEEE.