Università degli Studi di Genova

DIBRIS

Department of Computer Science, Bioengineering, Robotics and Systems Engineering

Master Thesis

in

Robotics Engineering

Predictive Monitoring for Safe Deliberation

Supervisor:
Armando Tacchella - Università degli Studi di Genova

Co-Supervisors:
Giuseppe Cicala - Università degli Studi di Genova
Michele Colledanchise - Istituto Italiano di Tecnologia

Candidate:
Chiara Terrile

Abstract

Assistive robots are becoming widely used in different fields, especially for elder or disabled people. These robots are able to sense and process the collected information with the goal of providing help and assistance to people in their daily living activities.

However, differently from industrial robots, assistive robots have to deal with situations where even expected problems (e.g., a low battery level) can lead to failure (e.g., inability to reach the goal) and repeated failures may lead to user's mistrust. Novel solutions are required to ensure that robots achieve high levels of autonomy while respecting given safety and liveness requirements.

Our working hypothesis is that *behaviour trees* (BTs) are used to encode robot's deliberation strategy and coordinate specific skills (e.g., navigating to places, grasping objects, interacting with humans) that, in turn, rely on the robot's functional and hardware components. In this context, *runtime verification* (RV) can be used to monitor the execution of the BT and related components to detect requirement violations and suggest recovery actions. To improve on autonomy, it is desirable to detect impending property violations in order to enable the robot to devise corrective actions before it is too late and human intervention is required.

We call *predictive monitoring* the capability of detecting property violations in advance, and we approach the problem by (*i*) learning abstract descriptions of the environment through *automata learning* techniques and (*ii*) using the learned descriptions to check in advance whether an execution trace may lead to a property violation or not. Our experiments with simulated executions of the R1 robot show that our methodology is promising, enabling the detection of impending property violations for a simple navigation task in a domestic scenario. Future works will include experimenting with more complex tasks and scenarios to assess efficiency and effectiveness of the proposed methodology.

Contents

Introduction

1.1 Context

Nowadays the population of the world is aging and consequently supporting and caring elder people is becoming more and more important. *"For the first time in history, the number of people older than 64 years old in the world has surpassed those younger than 5 years old. By 2050, the share of the global population older than 65 is expected to rise to 16%, from 9.3% in 2020"* [1].
In the last years, social relationships, especially for older people, have become more and more difficult to handle. In particular social isolation and loneliness have increased considerably [2; 3], leading to negative effects for healing processes, including the increased risk of premature death [4]. The current COVID-19 pandemic offers a *"unique opportunity to envision, pilot or implement novel technological solutions that could have a lasting impact on the health and well-being of older adults"* [5]. Socially Assistive Robots (SARs) can provide a technological means to ameliorate some of these problems, e.g., to care for the elder, accomplish simple but monotonous tasks, or to support healthcare professionals. In general, SARs must be able to sense and process the collected information with the goal of providing help and assistance to people in their daily living activities, recognize and interpret verbal and non-verbal communications such as speech, gestures and eye contact, and respond appropriately using their own emotional intelligence and conversational abilities.
The envisioned scenario is one where humans and robots are able to work together keeping people safe and without substituting in-person care, but reducing the number of times that an intervention from the healthcare staff is needed. Currently, SARs are being developed to help address care gaps in rehabilitation

due to increased patient survival after diseases with severe functional deficits, such as stroke [6]. SARs are also a promising innovative tool for rehabilitation, where movement frequency and intensity, coupled with patient engagement, are the most important factors that influence the recovery process [7; 8; 9]. SARs are also used to encourage self-practice [10; 11; 12] and improve therapeutic compliance through verbal, non-contact, and personalized coaching [13].

1.2 Motivations

A fundamental aspect for the successful design and deployment of any assistive solution is safety, since users are often non-experts. Indeed, differently from industrial robots, assistive robots have to deal with unstructured environments populated by humans, therefore also autonomy plays a crucial role. Mating safety and autonomy turns out to be a challenging task. From a practical point of view, e.g., industrial robots are safe and autonomous, but traditionally their design does not take into account direct physical contact between the person and the robot. Another aspect is that usually industrial robots are used by trained users able to supervise the robot's operations being aware of safety guidelines. In domestic environment, instead, assistive robots are made for people that are often not expected to have any sort of specific experience or training, including doctors, physiotherapists, carers and the patients themselves. Both safety and autonomy need to be ensured, lest the users come to distrust the robot and refrain from using it.

A feasible way to strike a balance between safety and autonomy in assistive robots is Runtime Verification (RV) also known as Runtime Monitoring. RV techniques can check whether the execution of a robot is violating a property in order to stop the executions and activate corrective actions. In assistive robots monitoring is used to verify the occurrence of adverse events in the application environment [14; 15]. Unfortunately, autonomy may not be guaranteed by runtime monitoring alone. Indeed, the goal of monitoring is to detect a fault as it happens, but since our goal is to obtain an autonomous robot, knowing that a fault *is about to happen* would make room for robot's deliberation to take actions before it is too late and safe shutdown, human intervention or other autonomy-breaking actions need to be taken.

1.3 Objectives

The objective of this work is to enrich the control software of assistive robots with Predictive Monitoring capabilities. Our reference robotic architecture is composed of three levels: deliberative, functional and control. The deliberative layer is where the robot's policies and task plans are computed; the functional layer groups elements that supervise specific behaviors such as navigation, grasping and object-recognition; finally the control layer embraces the elements that are closer to the hardware such as drivers and filters. We assume that the deliberative part is implemented with Behavior Trees (BTs) that organize the robot's skills in order to accomplish specific task plots. An intermediate goal to reach our objective is to model everything that sits below the deliberative layer, i.e., functional and control layer plus the external environment, in a compact way using a finite state machine (FSM). Such FSM is not designed manually, but it is obtained via Automata Learning techniques. By logging the messages exchanged between the deliberative layer and the functional layer, we are able to feed automata learning algorithms with data that can be used to extrapolate models that interact with deliberative layer according to real execution patterns. Once we have such model, we can implement predictive monitoring by comparing actual execution traces with traces produced by the learned models. Assuming that we have learned FMSs that encode both nominal scenarios, i.e., traces leading to success, and faulty scenarios, i.e., traces leading to property violations, we can recognize impeding failures by comparing actual execution traces with those generated by the learned FSMs.

1.4 Results

Along the objectives mentioned before, in this thesis we have reached the following results:

- We have developed a methodology based on automata learning to obtain models of the environment below the deliberative layer; this required the development of a suitable abstraction and the integration of a software library for automata learning with the implementation of suitable glue code to obtain FSMs that support predictive monitoring.

- We have developed a piece of software that can parse execution traces (both off-line and on-line) and compares them with those obtained by the learned FSMs in order to spot impending property violations in advance with respect to runtime monitors.
- We have experimented our implementations in a simple but realistic scenario involving navigation in a domestic environment; we considered both an off-line simulator (to tune the methodology) and an on-line simulator to confirm the results.
- With the on-line simulator we have shown that in scenarios where a safety requirement will be violated, our software allows the robot to understands that something will go wrong before the intervention of the runtime monitoring module.

Unfortunately, due to COVID-19 restrictions, we were not able to run extensive testing on the R1 platform on which the simulators are based. However, the software connections used by our software to interface with the simulator are exactly the same as those used on the real robot, so we assume that repeating the experiments on the real setup will not require extensive additional work.

1.5 Document's Structure

The rest of this thesis is structured in the following way :

- **Chapter 2** : in this chapter we found the State of the Art, with particular attention to Behaviour Trees, Model-Based Architectures, Automata Learning and Runtime Verification.
- **Chapter 3** : in this chapter we describe the case study with particular attention to the working principle of the framework of interest.
- **Chapter 4** : in this chapter we describe the methodology, so the proposed approaches with the related algorithms.
- **Chapter 5** : in this chapter we describe the implementation together with the results obtained while applying what described in Chapter 4.

- **Chapter 6** : in this chapter we conclude the description of the thesis proposing some future works.

State of the art

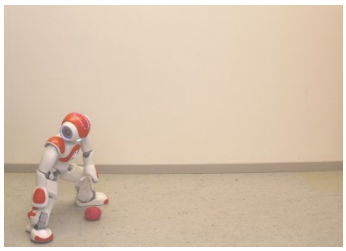
2.1 Behaviour Trees

2.1.1 Introduction to Behaviour Trees

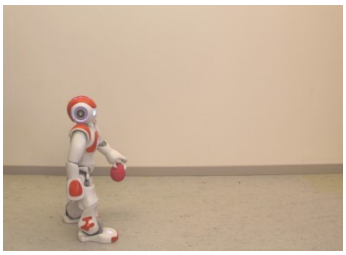
Behaviour trees (BTs) are strong tools used in the Artificial Intelligence field in order to perform deliberation is autonomous system.
A BT is a graphical model that is both *modular* and *reactive* [16]. By modular it is meant that subtrees can be reused in other BTs, since a subtree is a BT itself. By reactive is meant that the system is able to react to exogenous events.
BTs were initially used for computer games, because they allow to model autonomous actors which are the non-playing characters. This kind of characters are able to take decisions in a reactive way, being in a unknown environment.
One application of BTs in a video-game's scenario (Mario AI) has been proposed by M. Colledanchise *et al* [17]. The goal of this study is to automatically synthesize a BT, starting from a set of low-level actions in the game, and some high-level sensing (e.g. obstacles in the unknown environment).

BTs are not only applied in the field of computer games, but since they are flexible, reusable and suitable for deliberative blocks, they have also been applied in robotics.
In the autonomous robotics [18] complex behaviours are called *missions* and they are composed of a variety of skills. Skills represent a low-level abstraction, since they are for example controllers for actuators.
Initially these missions were implemented using hierarchical state machines, but with the coming of BTs, hierarchical state machines have been substituted, guaranteeing better modularity than before.

The usage of BTs in robotics has covered different fields. Starting from manipulation [19] where simple tasks such as pick and place objects are defined via different skills that are considered as independent components. More abstract skills can be used at runtime to plan (e.g. using a PDDL planner) the best sequence of actions. BTs have been applied also to task planning [20], learning [21] and also to human robot interaction (HRI). Among the HRI examples there is a study by N. Axelsson and G. Skantze [22] based on the speech interaction between human and robot. The aim of this project is to obtain an adaptive dialogue between human and robot, as it happens between humans. Regarding HRI another example is given by M. Kim *et al* [23] where a mobile robot is able to follow a person and learn from his actions at runtime.
We now focus on a typical robotic example to better understand the working principles of BTs: a pick and place task that can be easily modeled via BTs [24]. In Figure 2.1 are shown basic actions that can be performed in a task of this type.



(a) Robot picking up the ball from a starting location



(b) Robot moving toward a goal with the ball in its hand

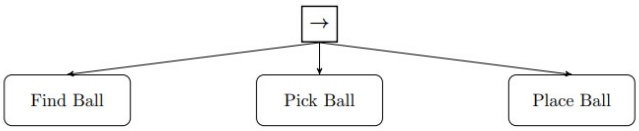
(c) External entity (human) taking the ball from the robot

(d) Robot approaching the ball in the new location

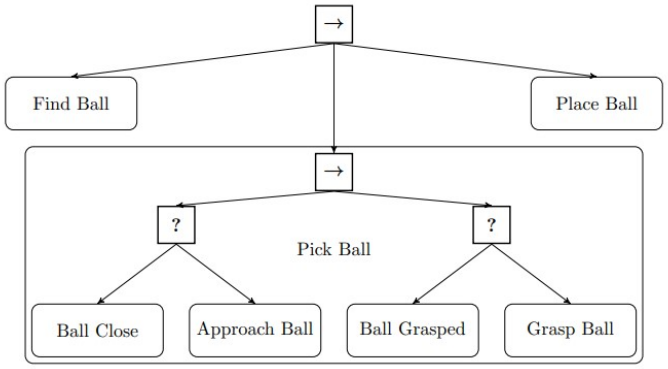
Figure 2.1: Example of pick and place tasks

This behaviour can be easily described via modules (Figure 2.2a), in fact, the behaviour is composed by a sequence of sub-behaviours that are task independent,

which means that the designer when is creating a sub-behaviour does not need to know which will be the next sub-behaviour.
Starting from Figure 2.2a it is possible to design this sub-behaviours by adding recursively more details, obtaining what shown in Figure 2.2b.



(a) High level BT of pick and place task



(b) More detailed BT of pick and place task

Figure 2.2: BT illustrations of the pick and place task with different degrees of abstraction

Because of their structure, BTs allow behaviors to be carried out re-actively, this means (referring to the example) that executing the behaviour *Place Ball* does not exclude to verify simultaneously that the ball is still detected and picked. In fact, if for an external event, the ball is no more in the robot's hand, the sub-behaviour *Place Ball* will be aborted and other behaviours such as *Pick Ball* or *Find Ball* will be re-executed.

2.1.2 Formulation of BTs

From a formal point of view, a BT is a directed rooted tree composed of internal nodes (*control nodes*) and of leaf nodes (*execution nodes*). For each connected node we talk about *parent* and *child*, where the root is the node without parents,

all other nodes have one parent and the control flow nodes have at least one child. The execution of a BT starts from the root node that generates a *tick* (a signal allowing the execution of a node) with a certain frequency and this tick is sent to the root's children. When a node receives a tick, it immediately returns to its parent *running* if the execution is ongoing, *success* if the goal has been achieved, or *failure* in the negative case. In BTs' formulation, there are four categories of control flow nodes that are *fallback*, *sequence*, *parallel* and *decorator* nodes, while for execution nodes we have *action* and *condition* nodes.

Fallback This node routes the ticks to all its children going from left to right until a child returning *success* or *running* is found, once found the node, the tick is interrupted and the node returns *success* or *running* accordingly to its parents. If all the fallback's children return *failure*, also the fallback will return *failure*. The symbol of this kind of node is represented as a box containing a "?" as shown in Figure 2.3.

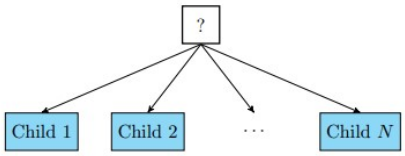


Figure 2.3: Fallback nodes with N children

Sequence This node routes the ticks to its children going from the left until a child returning *failure* or *running* is found, once found the node, the tick is interrupted and the node returns *failure* or *running* accordingly to its parents. If all the sequence's children return *success*, also the sequence will return *success*. The symbol of this kind of node is represented as a box containing a "→" as shown in Figure 2.4.

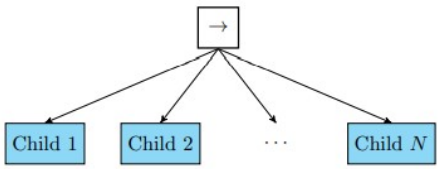


Figure 2.4: Sequence nodes with N children

Parallel This node routes the ticks to its children and returns *success* if M children return *success*, if $N - M + 1$ children return *failure* the node returns *failure* too, otherwise if none of these condition are verified, it returns *running*. Note that N is the total number of children and $M \leq N$ is a threshold defined by the user.
The symbol of this kind of node is represented as a box containing a "⇉" as shown in Figure 2.5.

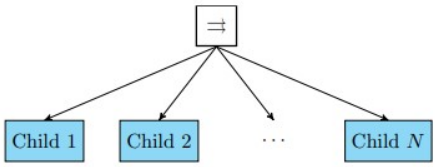
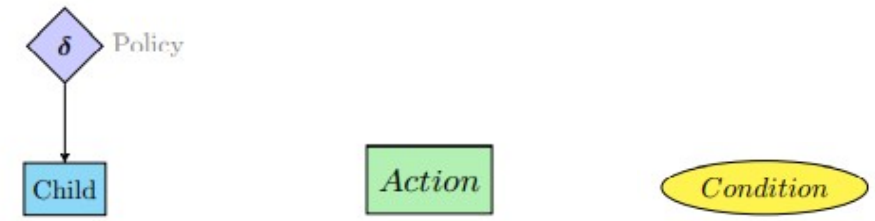


Figure 2.5: Parallel nodes with N children

Decorator This node is a control flow node with only a child that manipulates the return status of its own child according to a user-defined rule. For example, an *invert* decorator inverts the *success/failure* status of the child (see Figure 2.6a).

Action This node executes a command when receives a tick returning *success* if the action is completed or *failure* if the action is failed. If the action is instead still ongoing, it will return *running*. (see Figure 2.6b).

Condition This node when receives a tick checks a certain preposition returning *success* or *failure* depending on the preposition check. This node will never return *running* (see Figure 2.6c).



(a) Decorator node (b) Action node (c) Condition node

Figure 2.6: Decorator (a), Action (b) and Condition (c) nodes

2.2 Model-Based Architectures

Model-Based Architectures are a strong tool to design an architecture since they allow to perform runtime monitoring [25] and to obtain autonomous systems [26]. In a model-based architecture knowledge is encapsulated in the form of models that are employed at the various control layers to support the predefined system objectives (see Figure 2.7 for a simple example of model-based architecture).

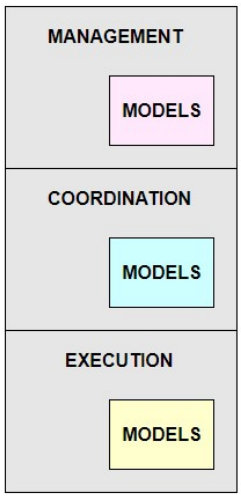


Figure 2.7: Simple example of Model-Based Architecture

A key requirement when dealing with this kind of architectures is the systematic development and integration of dynamic and symbolic models at different layers. A model according to [27] is "an approximation representation, or idealization of selected aspects of the structure, behavior, operation or other characteristics of a real-world process, concept, or system". For simplification purposes, models usually suppress detail through abstraction which can be achieved through generalization, deletion and distortion.
We can think about a model-based architecture as composed of a hierarchy of communicating components. In the architecture is not only captured structural properties but also behaviours. To enable behavior specification, components that are leaves in the hierarchy, can be equipped with behavior descriptions such as I/O automaton [28] or other Finite State Models (that will be later investigated in Subsection 2.3.1).

Model-based architectures can be applied is several fields, starting from medical applications where they can be employed for testing medical Cyber-Physical Systems [29], then going through model-based Adaptation for Self-Healing Systems [30], until the application to vehicles, and in particular the design of a model-based vehicular prognostics framework [31].
Regarding robotics applications, model-based architectures can be found in control architectures for attentive robots in rescue scenarios [32] or learning (based on neural networks) applied to a mobile robot navigating in an environment [33].

2.3 Automata Learning

Automata Learning is a learning approach based on Finite State Machines (FSM). The key concept is to represent the learned model (that can be real software components) as a state machine that resumes the behaviour of the system.
There are two main approaches of automata learning, that are *active learning* and *passive learning* and their implementations will be explained later.

2.3.1 Finite State Models

The Finite State Models [34] that will be surveyed in this thesis are three : Deterministic Finite Automaton (DFA), Mealy State Machine, and I/O Automaton. In particular, because of the purposes of the project we will mainly consider the last two, and the relative mapping between them.

Deterministic Finite Automaton. A *Deterministic Finite Automaton* (DFA) refers to a machine that reads only one symbol at a time (this is why we speak about determinism).
A DFA is defined as a quintuple $(Q, \Sigma, \delta, F, q_0)$ where:

- Q is the non-empty finite state of states
- Σ is the alphabet, i.e. the finite set of letters that represent the inputs
- δ is the transition function defined as :

$$\delta : Q \times \Sigma \to Q$$

- $F \subseteq Q$ is the set of final states
- $q_0 \in Q$ is the initial state

In Figure 2.8 is shown an example of DFA over the alphabet $\Sigma = \{a, b\}$, with $Q = \{q_0, q_1, q_2, q_3\}$ and initial state q_0 represented by double lined circles.

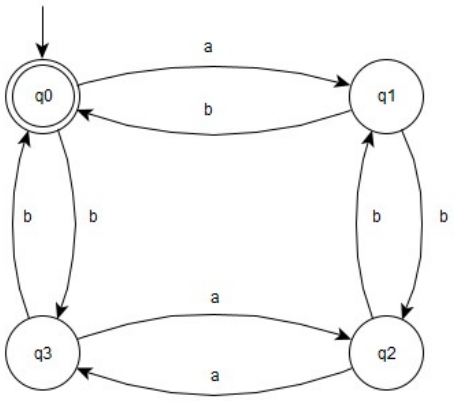


Figure 2.8: Example of DFA

Mealy State Machine. A *Mealy State Machine* is defined as a sextuple $(Q, I, O, \delta, \lambda, q_0)$ where:

- Q is the non-empty finite state of states
- I is the finite set of input symbols
- O is the finite set of output symbols
- δ is the transition function defined as :

$$\delta : Q \times I \to Q$$

- λ is the output function defined as :

$$\lambda : Q \times I \to O$$

- $q_0 \in Q$ is the initial state

When the Mealy machine is in a state $q \in Q$ it can move to another state once received an input $i \in I$, the target state is specified by $\delta(q, i)$ and the produced output is given by $\lambda(q, i)$.
In Figure 2.9 is shown an example of Mealy machine over the sets $I = \{a, b\}$ and $O = \{1, 0\}$, with $Q = \{q_0, q_1, q_2, q_3\}$ and initial state q_0.

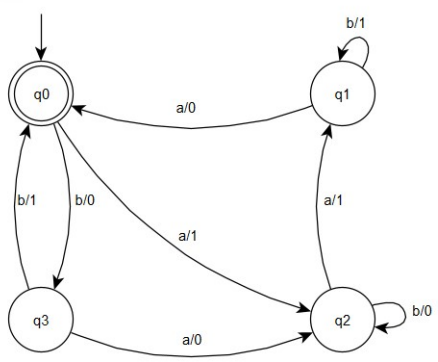


Figure 2.9: Example of Mealy machine

I/O Automaton. An *I/O Automaton* [35] is defined as a quintuple (Q, I, O, δ, q_0) where:

- Q is the non-empty finite state of states
- I is the finite set of input symbols
- O is the finite set of output symbols
- δ is the transition function that can be an *input transition* (2.1) or an *output transition* (2.2) :

$$\delta : Q \times I \to Q \tag{2.1}$$

$$\delta : Q \times O \to Q \tag{2.2}$$

- $q_0 \in Q$ is the initial state

When the I/O Automaton is in a state $q \in Q$ it can move to another state once received an element $i \in I$ or $o \in O$. We speak about input transition if the

symbols received is of type i, otherwise, if the symbol is of type o, we have an output transition. The target state is specified by $\delta(q, a)$ where a is an action that can be $i \in I$ or $o \in O$. In Figure 2.10 is shown an example of I/O Automaton over the sets $I = \{a, b\}$ and $O = \{1, 0\}$, with $Q = \{q_0, q_1, q_2, q_3\}$ and initial state q_0. The input transitions are highlighted in blue, while the output transitions are represented in red.

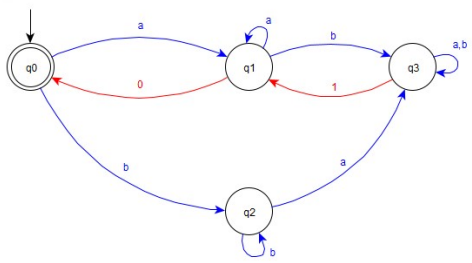


Figure 2.10: Example of I/O Automaton

2.3.2 Mapping

In some cases it may be necessary to convert an I/O Automaton into a Mealy state machine or vice-versa [36]. If we have an I/O automaton defined as $P = (Q, I, O, \delta, q_0)$, we can define a *transition function* $I2M(P) = (Q_P, \Sigma_I, \Sigma_O, q_{P0}, \tau_P)$. This transition function describes a Mealy state machine with :

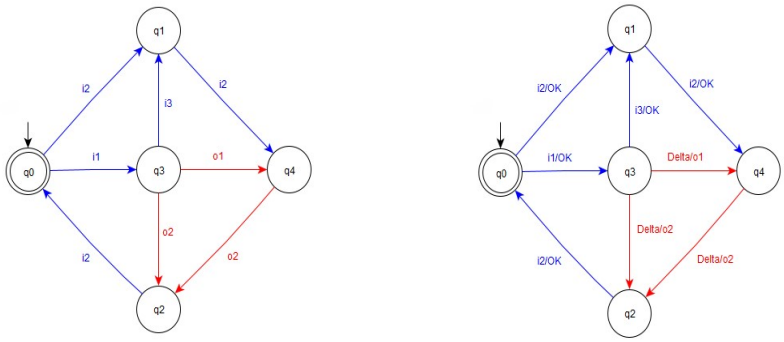
- $Q_P = Q$ as the set of states
- $q_{P0} = q_0$ as the initial state
- $\Sigma_I = I \cup \{\Delta\}$ as the input alphabet, where Δ is a fresh *delay* symbol
- $\Sigma_O = O \cup \{\surd\}$ as the output alphabet, where $\surd$ is a fresh *accept* symbol
- for every transition $q \xrightarrow{a} q'$ in δ where $a \in I$ we have that $\tau_P(q, a) = \{(q', \surd)\}$
- for every transition $q \xrightarrow{a} q'$ in δ where $a \in O$ we have that $(q', a) \in \tau_P(q, \Delta)$

So, the translation for each input transition $q \xrightarrow{i} q'$ is $\tau_P(q, i) = (q', \surd)$, where $\surd$ means that the input i is accepted by the system.
The delay input action Δ, instead, represents investigation about all possible

outputs of the system. So, each output transitions $q \xrightarrow{o} q'$ is translated to $\tau_P(q, \Delta) = (q', o)$. Intuitively, the **backward translation** operates on a Mealy state machine $M = (Q, I \cup \{\Delta\}, O \cup \{\surd\}, q_0, \tau)$ and obtains the corresponding I/O Automaton $P = (Q, I, O, \delta, q_0)$ by applying the following rules for the transitions :

$$\tau(q, i) = (q', \surd) \Rightarrow i \in I \wedge q \xrightarrow{i} q'$$
$$\tau(q, \Delta) = (q', o) \Rightarrow o \in O \wedge q \xrightarrow{o} q'$$

To better understand this procedure we now consider a simple example.
In Figure 2.11a we have an I/O Automaton that we want to translate into a Mealy state machine. For simplicity we write the symbol as *Delta* and $\surd$ as *OK*, then the resulting Mealy state machine is shown in Figure 2.11b.



(a) I/O Automaton (b) Mealy state machine

Figure 2.11: Translation from I/O Automaton to Mealy state machine

2.3.3 Active Learning

Active learning [37] is a learning approach based on the alternation of two phases: an *exploration* phase and a *testing* phase. In the first one we talk about *membership queries*, while in the second one about *equivalence queries*. Membership queries are used to build hypothesis models of the system during the learning, while equivalence queries are used to accept or reject the obtained hypothesis model by comparing it to the real system. In active learning, these two phases are iterated until is found a valid model representing the system. This kind of learning is based on two "entities", a *Teacher* that is the source of examples, and a *Learner* that is the learning algorithm used, such as L*.

L* Algorithm. The largest part of active state machine learning is based on the L* algorithm, introduced for the first time by D. Angluin [38]. Since L* is used for active learning, it is based on the dialogue between a *Teacher* and a *Learner*. The Teacher in question is a "minimally adequate Teacher" while the Learner is able to learn from an initially unknown regular set. The *minimally adequate Teacher* can answer two types of questions coming from the Learner regarding the unknown regular set. We talk about the aforementioned *membership queries*, that consist of a string t, the Teacher can answer *yes* or *no* depending on if t belongs to the unknown regular set or not.
The other type of questions is the *conjecture* that is the description of regular set S, the Teacher will answer yes if the set S is equal to the unknown regular set, otherwise it produces a *counterexample* to show that the conjecture is not adequate to represent the unknown regular set. To understand how L* works (i.e. how the queries can be answered), it is necessary to introduce the concept of *Observation Table*. During the execution, L* has information about a finite collection of strings over an alphabet A, and is able to classify them as belonging or not to the unknown regular set U.
This is then stored into an observation table composed of three parts : a nonempty finite prefix-closed set S of strings, a nonempty finite suffix-closed set E of strings,and a finite function T mapping $((S \cup S \cdot A) \cdot E)$ to $\{0, 1\}$. So we can denote the observation table as (S, E, T), and consider $T(u)$ as equal to 1 if and only if $u \in U$.
In Algorithm 1 is shown the working principle of the Learner L*.

Algorithm 1: L* algorithm

Initialize S and E to $\{\lambda\}$
Ask membership queries for λ and each $a \in A$
Construct the initial observation table (S, E, T)
  repeat
    while (S, E, T) is not closed or not consistent **do**
      if (S, E, T) is not consistent **then**
        find s_1 and s_2 in S, $a \in A$ and $e \in E$ s.t.
        $row(s_1) = row(s_2)$ and $T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$
        extend T to $(S \cup S \cdot A) \cdot E$ using membership queries.
      end if
      if (S, E, T) is not closed **then**
        find $s_1 \in S$ and $a \in A$ s.t.
        $row(s_1 \cdot a)$ is different from $row(s)$ for all $s \in S$
        add $s_1 \cdot a$ to S
        extend T to $(S \cup S \cdot A) \cdot E$ using membership queries
      end if
    end while
    Once (S, E, T) is closed and consistent, let $M = M(S, E, T)$.
    Make the conjecture M.
    if the Teacher replies with a counter-example t **then**
      add t and all its prefixes to S
      extend T to $(S \cup S \cdot A) \cdot E$ using membership queries.
    end if
  until the Teacher replies *yes* to the conjecture M
  Halt and output M

2.3.4 Passive Learning

Passive learning is an approach where, differently from active learning, the model to learn is obtained from a set of traces, so the learning procedure is done offline. There are several algorithms to implement passive learning and most of them are based on the concept state merging. The one that will be surveyed in this thesis is the *Regular Positive and Negative Inference* (RPNI) algorithm [39], this algorithm was firstly applied to DFA but it can be also extended to Mealy state machine, depending on the type of model that we are learning.

RPNI whit DFA. The RPNI algorithm is a polynomial time algorithm to identify a DFA compatible with a given sample $S = S^+ \cup S^-$, where with S^+ are denoted the positive samples, and with S^- the negative ones. The algorithm builds a *prefix tree acceptor* $PTA(S^+)$ for the positive examples in S^+. We define $\bar{N}$ as the number of states of $PTA(S^+)$.
The algorithm performs an ordered search in the space of partitions of the set of states of $PTA(S^+)$ under the control of the set of negative examples in S^-. Then we denote the partition, π_0, corresponding to the automation $PTA(S^+)$ as $\{\{0\}, \{1\}, ..., \{\bar{N} - 1\}\}$. Below the reader can find the pseudo-code of the RPNI algorithm (Algorithm 2).

Algorithm 2: RPNI algorithm

Input : A sample $S = S^+ \cup S^-$
Output : A DFA compatible with S

procedure :
$\pi = \pi_0 = \{\{0\}, \{1\}, ..., \{\bar{N} - 1\}\}$
$M = PTA(S^+)$
for i = 1 **to** $\bar{N} - 1$ **do**
for j = 0 **to** $i - 1$ **do**
$\tilde{\pi} = \pi \backslash \{B(i, \pi), B(j, \pi)\} \cup \{B(i, \pi) \cup B(j, \pi)\}$
$M_{\tilde{\pi}} = derive(M, \tilde{\pi})$
$\hat{\pi} = deterministic\_range(M_{\tilde{\pi}})$
if $compatible(M_{\hat{\pi}}, S^-)$ **then**
$M = M_{\hat{\pi}}$
$\pi = \hat{\pi}$
end if
end for
end for

By looking to the pseudo-code, we can notice that at each step i the algorithm tries to refine the current partition by merging the blocks of the states, so that the quotient automaton corresponding to the refined partition is consistent with the negative samples S^-.
The function $derive(M, \tilde{\pi})$ returns $M_{\tilde{\pi}}$, that is the quotient automaton of M with respect to the partition $\tilde{\pi}$.
There is another function, which is $deterministic\_range(M_{\tilde{\pi}})$ that is used to

avoid non-determinism, this function in fact returns the partition $\hat{\pi}$ obtained by successively merging the blocks in $\tilde{\pi}$ that cause non-determinism. Thanks to these functions, the resulting automaton is guaranteed to be a DFA and as final step it needs to be checked its compatibility with all examples is S^- with the boolean function $compatible(M_{\hat{\pi}})$.

Example

Suppose that we want to learn the DFA shown in Figure 2.12 and that we have $S = S^+ + S^-$, where $S^+ = \{b, aa, aaaa\}$ and $S^- = \{\lambda, a, aaa, baa\}$.

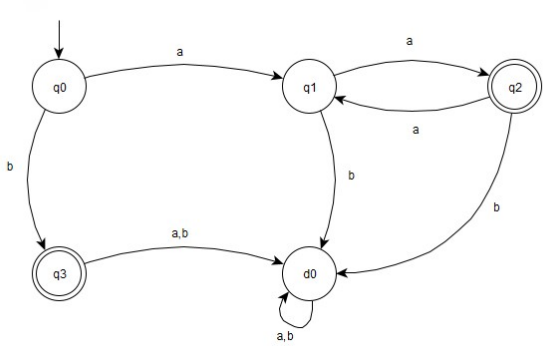


Figure 2.12: DFA

Starting from this knowledge of the samples, the resulting $M = PTA(S^+)$ is the one in Figure 2.13.

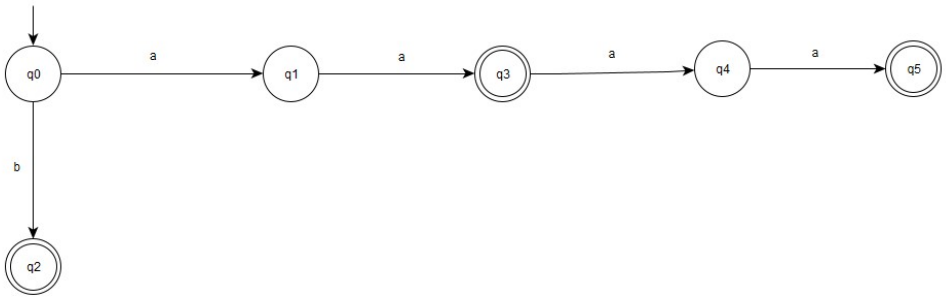


Figure 2.13: $M = PTA(S^+)$

With this configuration, we have an initial partition $\pi = \pi_0 = \{\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$. The next step is to find the partitions $\tilde{\pi}$ and $\hat{\pi}$ such that $\hat{\pi}$ is consistent with all the negative examples in S^-.
As first attempt we can try to merge blocks 1 and 0 of the partition π to obtain $M_{\tilde{\pi}}$ in Figure 2.14. Then, starting from $M_{\tilde{\pi}}$ we get $M_{\hat{\pi}}$ through the function *deterministic_merge* in Figure 2.15.

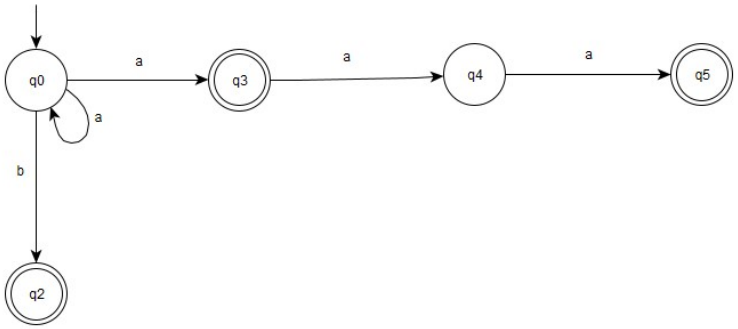


Figure 2.14: $M_{\tilde{\pi}}$ obtained by merging blocks 1 and 0

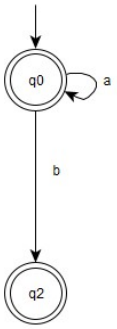


Figure 2.15: $M_{\hat{\pi}}$

By looking at $M_{\hat{\pi}}$ we can notice that the negative example $\lambda \in S^-$ is accepted, so the partition π remains unchanged, and we need to find new partitions $\tilde{\pi}$ and $\hat{\pi}$ to obtain the model. If we iterate this procedure we get the table in Figure 2.16 that shows all the possible combination of merged states and the relative negative examples.

Partition $\tilde{\pi}$	Partition $\hat{\pi}$	Negative Example
$\{\{0,1\},\{2\},\{3\},\{4\},\{5\}\}$	$\{\{0,1,3,4,5\},\{2\}\}$	a
$\{\{0,2\},\{1\},\{3\},\{4\},\{5\}\}$	$\{\{0,2\},\{1\},\{3\},\{4\},\{5\}\}$	λ
$\{\{0\},\{1,2\},\{3\},\{4\},\{5\}\}$	$\{\{0\},\{1,2\},\{3\},\{4\},\{5\}\}$	a
$\{\{0,3\},\{1\},\{2\},\{4\},\{5\}\}$	$\{\{0,3\},\{1,4\},\{2\},\{5\}\}$	λ
$\{\{0\},\{1,3\},\{2\},\{4\},\{5\}\}$	$\{\{0\},\{1,3,4,5\},\{2\}\}$	a
$\{\{0\},\{1\},\{2,3\},\{4\},\{5\}\}$	$\{\{0\},\{1\},\{2,3\},\{4\},\{5\}\}$	baa
$\{\{0,4\},\{1\},\{2\},\{3\},\{5\}\}$	$\{\{0,4\},\{1,5\},\{2\},\{3\}\}$	a
$\{\{0\},\{1,4\},\{2\},\{3\},\{5\}\}$	$\{\{0\},\{1,4\},\{2\},\{3,5\}\}^*$	—
$\{\{0,3,5\},\{1,4\},\{2\}\}$	$\{\{0,3,5\},\{1,4\},\{2\}\}$	λ
$\{\{0\},\{1,3,4,5\},\{2\}\}$	$\{\{0\},\{1,3,4,5\},\{2\}\}$	a
$\{\{0\},\{1,4\},\{2,3,5\}\}$	$\{\{0\},\{1,4\},\{2,3,5\}\}$	baa
$\{\{0\},\{1,4\},\{2\},\{3,5\}\}$	$\{\{0\},\{1,4\},\{2\},\{3,5\}\}^*$	—
$\{\{0\},\{1,3,4,5\},\{2\}\}$	$\{\{0\},\{1,3,4,5\},\{2\}\}$	a

Figure 2.16: Execution of the RPNI algorithm

In the column **Negative Example** we have to focus on the cases denoted by "—", and in particular, by looking at the partition $\pi = \{\{0\},\{1,4\},\{2\},\{3,5\}\}$ it is possible to notice that the corresponding target DFA is exactly the one that we want to learn (Figure 2.12).

RPNI with Mealy Machines. The RPNI algorithm can be also applied when the machine that we want to learn is of type Mealy. We now explain this through an example.
Suppose that our machine is the one shown in Figure 2.17.

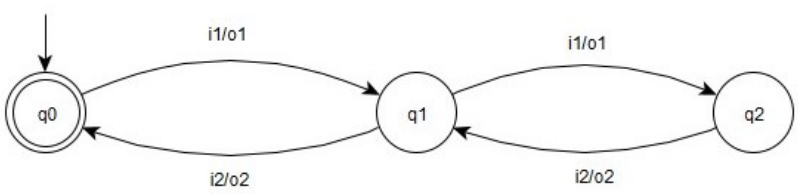


Figure 2.17: Mealy state machine

Now, given the examples $S = \{i_1o_1i_1o_1i_2o_2i_2o_2, i_1o_1i_2o_2\}$ we can build the corresponding $M = PTA(S^+)$ in Figure 2.18.

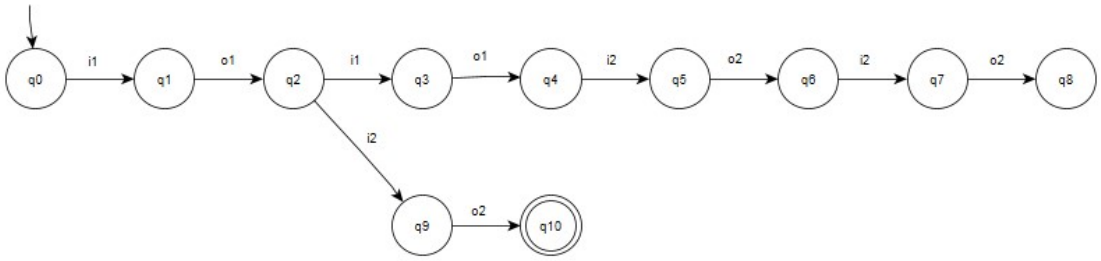


Figure 2.18: $M = PTA(S^+)$

In cases like this, we consider only the positive samples, since from a practical point of view, they can be considered as the traces coming from the execution of a certain automaton. Because of these features, learning this model can lead to over-approximation or under-approximation when merging the states.
To overcome this problem, a solution can be an integration of passive learning together with active learning [40]. The basic idea is to use the model obtained fro the RPNI algorithm as an oracle for the active learning. This means that the new traces coming from the log of the execution, will be given to the oracle to find counterexamples that can improve the obtained model in a dynamic way.

2.3.5 LearnLib

LearnLib [41] is an open-source library for automata learning that is written in Java to enable a high degree of flexibility, guaranteeing at the same time performances that allow large-scale applications.
The development of *LearnLib* began in 2003 with the aim of providing researchers and practitioners with a reusable set of components to facilitate and promote the use of automata learning.
LearnLib is equipped with a set a components to apply automata learning in practical settings, or to develop or analyze automata learning algorithms. Therefore, there are three main classes of features (Figure 2.19) : learning algorithms,

method for finding counterexamples (*Equivalence Queries*), and infrastructure components.

Figure 2.19: *LearnLib* feature overview

Learning Algorithms. *LearnLib* offers the possibility to use both active and passive learning that can be used with DFA or Mealy state machine depending on the algorithm considered (Figure 2.20).
Regarding the active algorithms we have common algorithms such as **L***, or other such as **DHC** [42], **KearnsVazirani** [43], **TTT** [44] and **NL*** [45].
Regarding passive algorithms, we have **RPNI** that can be applied to DFA or Mealy machines.

Algorithm	Target model
Active learning algorithms	
ADT	Mealy
DHC	Mealy
Discrimination Tree	DFA Mealy VPDA
Kearns Vazirani	DFA Mealy
L*	DFA Mealy
NL*	NFA
TTT	DFA Mealy VPDA
Passive learning algorithms	
RPNI	DFA Mealy
RPNI – EDSM	DFA
RPNI – MDL	DFA

Figure 2.20: *LearnLib* learning algorithms

Equivalence Tests and Finding Counterexamples. In active learning, once that the learning algorithm converges to a stable hypothesis, to ensure further progress we need a *counterexample* that can be referred to as an *equivalence query*. Often the most suitable way to approximate equivalence queries is to search for counterexamples directly, and this can be done through a random walk (only on Mealy machines), randomized generation of tests and exhaustive generation of test inputs (up to a certain depth).

Infrastructure. *LearnLib* provides infrastructures to support several functionalities, such as logging facilities, import/export mechanism to store and load hypotheses, or utilities for gathering statistics.
Very useful for practical applications are (optimizing) filters that are used to pre-process the queries posed by the learning algorithm.

2.4 Runtime Verification and Monitoring

Runtime verification (RV) is used in software engineering to guarantee the correct behaviour of a system. The goal of RV is to detect whether the system has encountered a failure or not. This is done through an online monitoring process, which gives a verdict about the correctness in the execution of a program.
In literature runtime verification can be considered as separated from the model checking technique, since they consider a different number of traces [46] (infinite for model checking, finite for RV) and because RV offers a more practical approach [47].

Runtime verification is a very strong tool because it allows the self-managing of a software, as M. G. Hinchey and R. Steritt proposed [48]. In fact the software, using RV can monitor its behaviour and compare it to what is known (e.g. beliefs, current data) being able to recognize a failure. Moreover, runtime verification can also be used for safety properties, as reported by K. Havelund and G. Rosu [49], allowing the detection of errors in a program.
RV has been applied also in robotics to manage the possible failures related to the environment (e.g. presence of humans) and the robot itself. There are three main approaches in runtime verification [50] that are namely *analytical*, *data-driven* and *knowledge-based*.

Analytical approaches are based on mathematical models and on the concept of residual, which means that if there is discrepancy due to a fault, there will be a residual from the comparison with the reference model.
Data-driven approaches, do not rely on mathematical models, but they are based on the input data and on statistical methods. These approaches are less used, but they can be found in some works of industrial robotics, especially in the control part. Knowledge-based approaches, are instead, a sort of trade off between the other two, because they are a combination of analytical and knowledge-based approaches.

From a practical point of view, RV in robotics can be implemented by using the Robot Operating System (ROS) which is a very powerful tool for programming robots [51]. This can be done thanks to a runtime verification framework ROSRV which allows to monitor the robot's behaviour. The RV Master, in fact, can intercept all node requests to the ROS Master and all messages being then able to actuate the monitoring.
In general runtime verification can be applied to different fields in robotics, starting from navigation [52] up to the application on Robonaut2 [53].
An other interesting application is the one provided by A. Desai [54], since it combines the power of model checking to the efficiency of runtime verification, obtaining a patrolling drone which is autonomous and able to detect failures.

Case Study

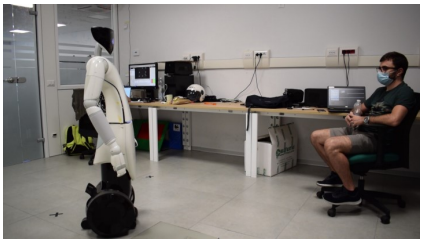
3.1 Description of the Case Study

The application scenario is based on the navigation of the R1 humanoid robot [55] (Figure 3.1) in a known space.

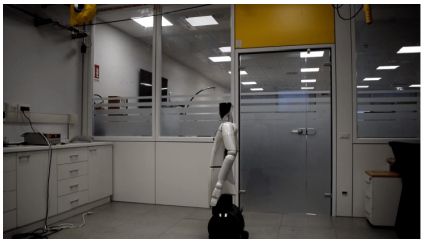
Figure 3.1: The R1 humanoid robot

The robot's goal is to reach a room from a fixed starting position. The robot is powered by a battery module: To ensure continuing operations, when the level of the battery reaches an *Early Warning* threshold of 30% of the total charge the robot stops and goes to the charging station. Once arrived to the charging station, the robot waits for the battery to get fully charged and then resumes the previous navigation task.
In Figure 3.2 there is an example of the behaviour described. In Figure 3.2a the robot is moving towards the destination; during the navigation the battery discharges (under 30%), so the robot aborts the current navigation to the destination (Figure 3.2b) and navigates to the charging station (Figure 3.2c). Once the robot reaches the charging station, it waits for the battery to be recharged

and then resumes the navigation task reaching the destination (Figure 3.2d).



(a) The robot navigates towards the destination



(b) The battery level gets low. The robot aborts the current navigation to the destination.

(c) The robot navigates to the charging station.

(d) The robot resumes the previous navigation task and reaches the destination.

Figure 3.2: Example of navigation of R1

A further safety requirement is that the level of the battery never reaches a value under 20% of the total charge, since the robot should always have reached the charging station before getting such a low battery level. In case of violation of this requirement the user is immediately notified.

3.2 Coordination BT-Skills-Components

All the behaviours described in Section 3.1 are managed by a Behaviour Tree (BT), *Skills* and *Components* that interact with each other to achieve a proper level of autonomy and safety.
The Behaviour Tree is used for the deliberative part and it is a *task plot*, i.e., a sequence of tasks required to achieve certain goals at runtime. The leaves in the BT communicate with skills, i.e., the coordination of functional components made accessible to task-plots; finally, the pieces of software that execute the code

at the functional layer are the components.
In Figure 3.3, there is a general representation of the interaction between the aforementioned elements. From top to bottom, we have the behavior tree (BT) depicted as a red triangle, the skills depicted as gray boxes, the components depicted as blue boxes and a blackboard depicted as black box. The blackboard is a shared memory that can be used for communication or synchronization purposes. Each entity is linked to others through connections that are bidirectional. For example, the BT activates the skills and receives a response by them.

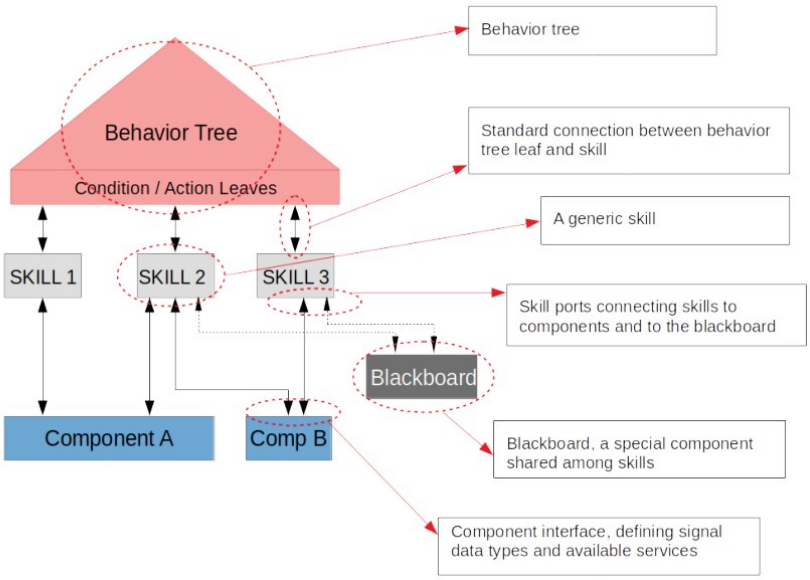


Figure 3.3: Coordination BT-skills-components

3.2.1 BT

We describe the implementation of BT by referring to the case study previously described. Figure 3.4 shows the BT that executes the navigation task together with the battery level management. As described in Subsection 2.1.2 we represent nodes and trees using a graphical syntax, where green rectangles denote *action* nodes, yellow eclipses denote *condition* nodes, while "→" and "?" denote *sequence* and *fallback* nodes, respectively. The execution of the BT begins with the robot receiving a "tick" signal that it propagates to its children.

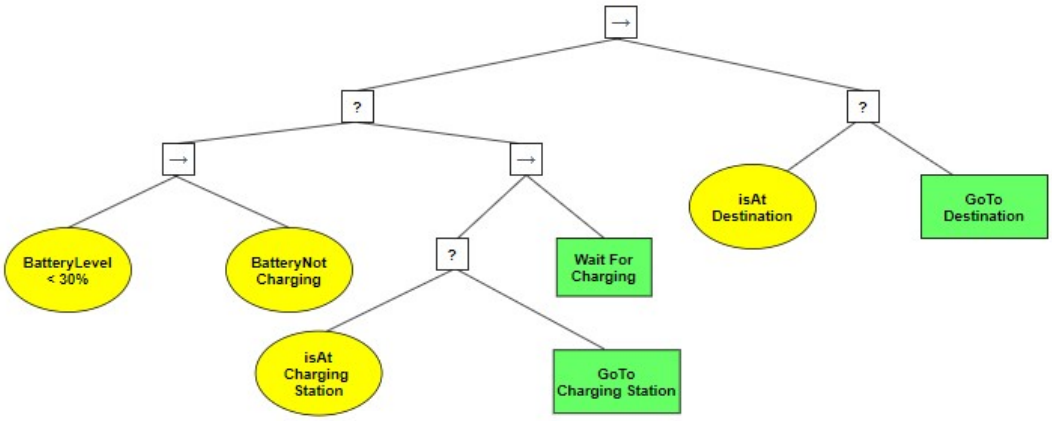


Figure 3.4: BT for the case study

The BT encodes the following logic (from top to bottom and from left to right):

- The robot checks if the battery level is below 30% of its capacity (condition node *BatteryLevel* $<$ *30%*) or if the battery is under charge (condition node *BatteryNotCharging*).
- If the battery level is below 30% and it is not already under charge, the robot goes to the charging station (action node *GoToChargingStation*).
- If the battery is under charge and the robot is at the charging station (condition node *isAtChargingStation*), the robot waits until the battery gets fully charged (action node *WaitForCharging*).
- If the battery level is above 30% and it is not under charge, the robot performs the main task. It checks if it is at the destination (condition node *isAtDestination*) and if not executes the action node *GoToDestination*.

Each action or condition of the BT in Figure 3.4 has a corresponding skill that is activated whenever the action or condition node receives a tick. In turn, the skill will send a request to the relative component to be able to send a response to the BT.

3.2.2 Skills

We now describe the skills developed for the scenario of interest. Each skill is modeled as a Finite State Machine.

GoToDestination Figure 3.5 shows the FSM defined for the action *GoToDestination* of the BT. When the corresponding action in the BT receives a tick, the skill sends a request to the *Navigation* component (that is based on YARP Navigation Stack and will be described in Subsection 3.2.3) and then waits for the outcome of the navigation (destination reached or path not found). The skill goes into a failure state if the Navigation Server cannot find a collision-free path to the destination. It goes into a success state if the robot reaches the destination. Whenever the corresponding action in the BT receives an halt, the skill sends a request to the Navigation Server to stop the robot's mobile base.

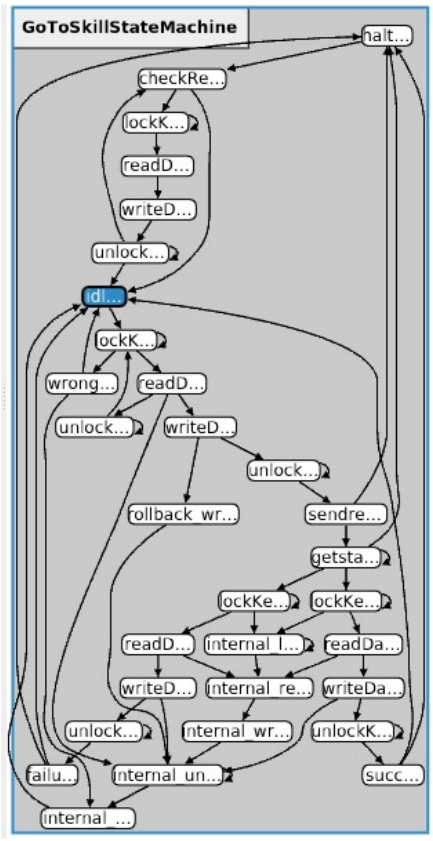


Figure 3.5: FSM implementing the skill *GoToDestination*

isAtDestination Figure 3.6 shows the FSM defined for the action *isAtDestination* of the BT. When the corresponding action in the BT receives a tick, the skill sends a request to a *Navigation* component, which implements an Adaptive Monte Carlo Localization [56]. The condition returns success if the robot is at the prescribed location, and failure otherwise.

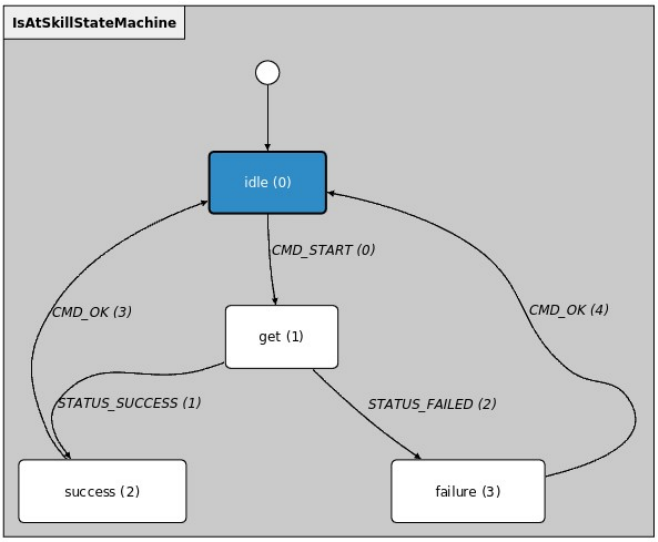


Figure 3.6: FSM implementing the skill *isAtDestination*

BatteryLevel Figure 3.7 shows the FSM for the condition *BatteryLevel*. When the corresponding condition receives a tick, the skill sends a request (state: get(1)) to the *BatteryReader* component (described in Section 3.2.3 below) that provides the battery level. The skill goes into a success state if the battery level is above 30% of its full capacity. It goes to a failure state otherwise.

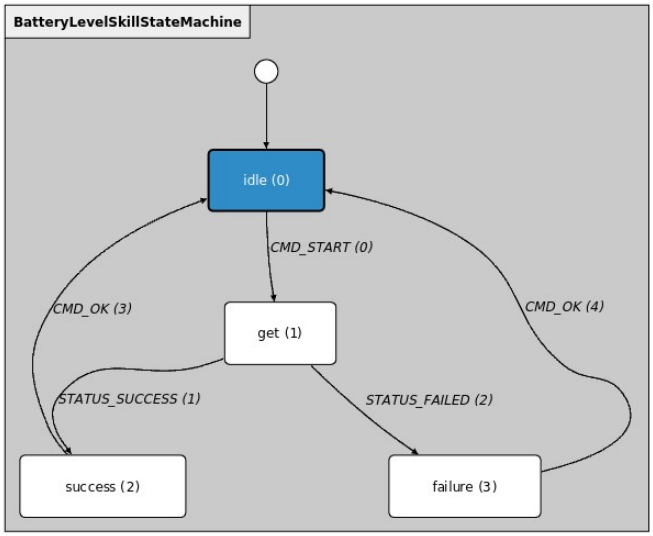


Figure 3.7: FSM implementing the skill *BatteryLevel*

BatteryNotCharging Figure 3.8 shows the FSM for the condition *BatteryNotCharging*. When the corresponding condition receives a tick, the skill sends a request (state: get(1)) to the *BatteryReader* component (described in Section 3.2.3 below) that provides the charging status of the battery. The skill goes into a success state if the battery is not under charge. It goes to a failure state otherwise.

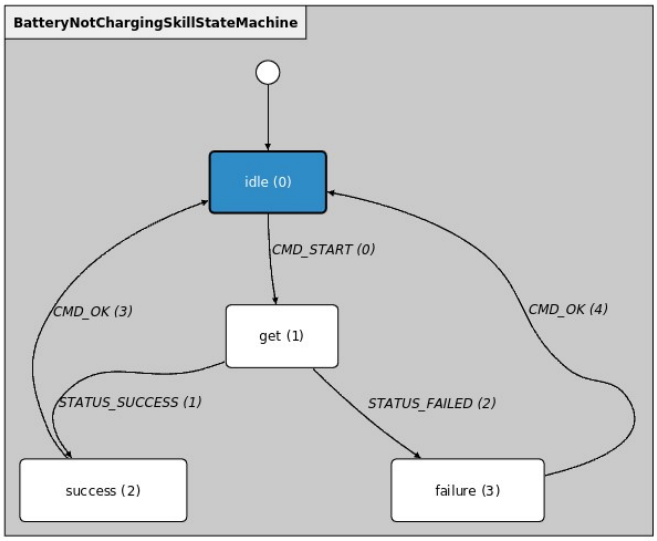


Figure 3.8: FSM implementing the skill *BatteryNotCharging*

GoToChargingStation The FSM for this skill is structured in the same way as the one in Figure 3.5, what is different is the destination that in this case is the Charging Station.

isAtChargingStation The FSM for this skill is structured in the same way as the one in Figure 3.6, what is different is the destination that in this case is the Charging Station.

3.2.3 Components

In Figure 3.9 we present the structure of the entire system. This scheme is analogous to the one in Figure 3.3, but it is detailed for our case study. The triangle stands for the BT process, grey boxes represent skill processes, purple boxes represent component processes and arrows represent channels. As we can notice we have just two components, *BatteryReader* and *Navigation*. The communication between skills and components is implemented via the *YARP Thrift* Remote Procedure Calls (RPCs)[1]. The Interface Definition Language (IDL) is Apache Thrift[2]; the communication protocol is synchronous and follows the RobMoSys Query communication pattern[3] .

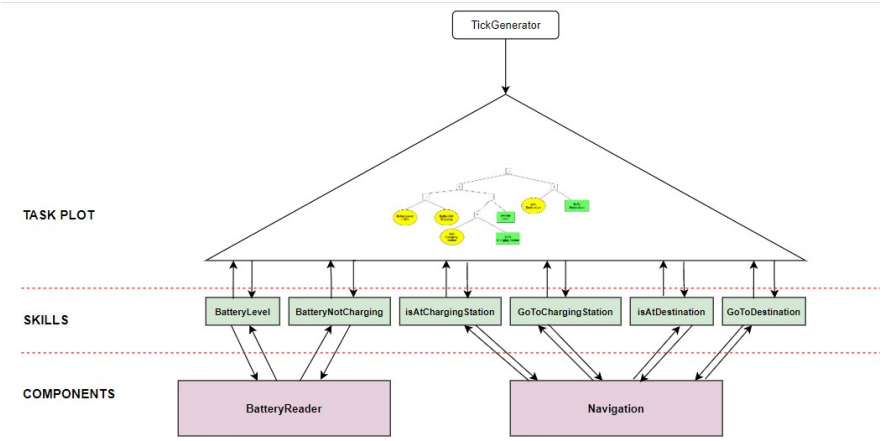


Figure 3.9: Graphical representation of the BT in Figure 3.4 plus its surrounding context of skills and components

[1] https://www.yarp.it/latest/idl_thrift.html
[2] https://thrift.apache.org/
[3] https://robmosys.eu/wiki/modeling:metamodels:commpattern

BatteryReader The *BatteryReader* provides information about the battery level and the charging status (if the battery is under charge). Listing 3.1 shows the Thrift file that specifies the interface of this component implemented via the following RPCs:

- `level` that takes no argument and returns the battery level percentage as a double
- `charging_status` that takes no argument and returns a `ChargingStatus`, defined in the file. It provides the charging status.

```
enum ChargingStatus {BATTERY_NOT_CHARGING, BATTERY_CHARGING}

service BatteryReader {
    double level();
    ChargingStatus charging_status();
}
```

Listing 3.1: Thrift file used to define the interface of the component *BatteryReader*

Navigation The *Navigation* component provides an interface with the Navigation Server [1]. Listing 3.2 shows the Thrift file that specifies the interface of this component implemented via the following RPCs:

- `goTo` that takes as parameters a string and returns void. It sends a request to the Navigation Server to move the robot to the location following a collision-free path to the location. `location` is a label that represents a pose stored in the Navigation Server.
- `getStatus` that takes as parameters a string and returns a `GoToStatus`, defined in the file. It sends a request to the Navigation Server to retrieve the current status of the navigation to the target `location`.
- `halt` that takes as parameters a string and returns void. It sends a request to the Navigation Server to stop the robot if its current target is `location`.
- `isAtLocation` that takes as parameters a string and returns a bool. It sends a request to the Navigation Server and returns `true` if the robot has reached the target `location` or `false` otherwise.

[1] The detail on the Navigation Server are available at https://github.com/robotology/navigation

```
enum GoToStatus
{
    NOT_STARTED,
    RUNNING,
    SUCCESS,
    ABORT
}

service GoTo {
  void goTo(1: string destination);
  GoToStatus getStatus(1: string destination);
  void halt(1: string destination);
  bool isAtLocation (1: string destination);
}
```

Listing 3.2: Thrift file used to define the interface of the component *NavigationHandler*

3.3 On-line Simulator and Monitoring

3.3.1 Runtime Monitoring

Runtime monitoring is implemented by software components that observe signals exchanged by the BT, the skills and the components to verify that they are consistent with the requirements. In our case, this is translated into a control that informs the user whenever the battery level reaches a value under 20% and the robot is not yet at the charging station. This corresponds to the requirement that the robot should reach the charging station while its battery level lies in an interval between [30%,20%] of the total charge.
In Figure 3.10 we show the current toolchain for the runtime monitor application.

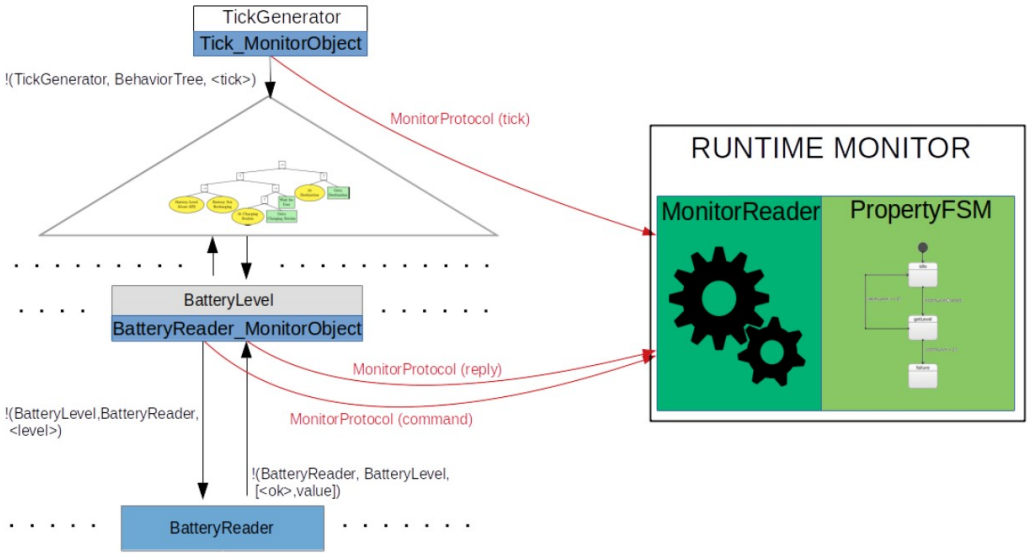


Figure 3.10: Example of runtime monitor execution for the property represented in Figure 3.12a. Red arrows represent the channels used for carrying the monitored signals to the MonitorReader object using the MonitorProtocol.

Runtime monitor application is composed by MonitorObject objects (one for each coordination interfaces defined with IDL and implemented via YARP thrift), a MonitorReader object and the FSM of the property to satisfy. Figure 3.11 shows the current toolchain for runtime monitor generation. A new MonitorObject is created for each channel in the system (from Tick generator to BT, from BT to

skills and from skills to components) starting from coordination interfaces defined with the IDL and implemented via Thrift files. In this way, all channels are setup for monitoring purposes. Each MonitorObject intercepts and forwards a copy of the original message content (command and reply) sent to the original destination to the MonitorReader (red arrows in Figure 3.10). The messages sent to the MonitorReader are generated and serialized using the information contained in the MonitorProtocol (i.e. the message fields). The MonitorReader is constantly listening on each monitored channel and reacts as soon as a signal arrives by raising an event accordingly. Runtime monitor application also takes a property's FSM as input. Intermediate representation of the properties (e.g. as shown in Figure 3.12a) are translated into a one or more FSMs written in SCXML language as shown in Figure 3.12b. Let us consider as example the monitor property described in Figure 3.12a and implemented in Figure 3.12b. The monitored channel is the one that goes from the component *BatteryReader* to the skill *BatteryLevel*. Whenever the value of the battery charge is transmitted from the component *BatteryReader* to the skill *BatteryLevel*, the BatteryReader_MonitorObject sends the same charge value to the MonitorReader that reacts by raising the `batteryLevelChanged` event (see Figure 3.12b). The FSM goes to getLevel state checking whether the charging value it is greater than or equal to 20 or less than 20: in the first case it returns to the initial location waiting for new messages and in the second case it enters the `Failure` state to signal that the property was violated in the current execution. Figure 3.13 shows the monitor GUI: if the property is not violated the semaphore light is green, red otherwise.

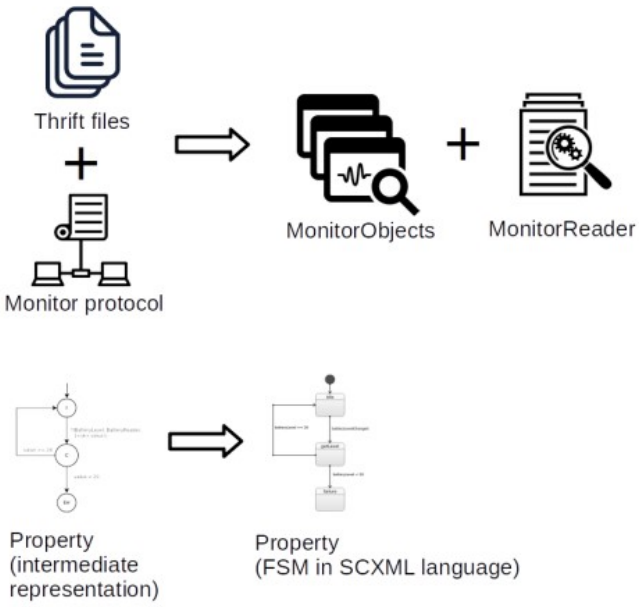
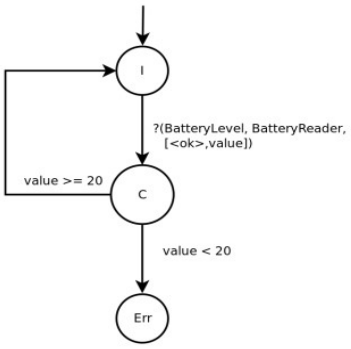
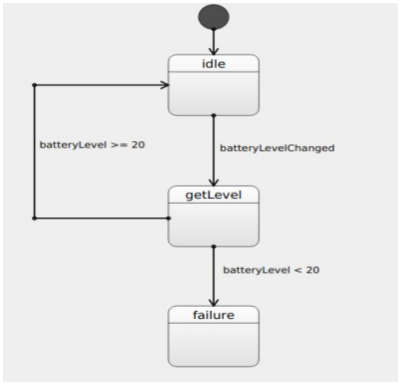


Figure 3.11: Toolchain for runtime monitor generation.



(a) FSM for safe properties



(b) Implemented FSM (in SCXML language) corresponding to the property represented in Figure 3.12a. Each rounded box is a state, arrows represent transitions on event (e.g. *batteryLevelChanged*) or on a condition (e.g. *batteryLevel >= 20*).

Figure 3.12: FSM representation for runtime monitoring

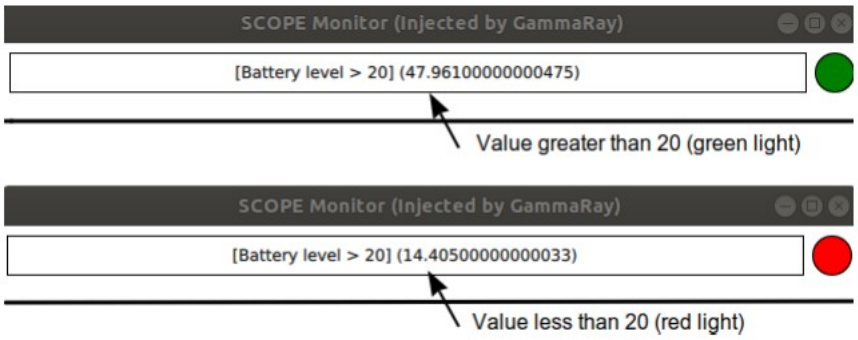


Figure 3.13: Runtime monitor GUI: if the property is not violated the semaphore light is green (on top), red otherwise

3.3.2 Scenarios in the On-line Simulator

We now present two different scenarios of application, a nominal one (with no property violation) and one with a safety property violation with the relative responses from the monitor in both cases.
The simulation is always run using the BT and the skills shown in Figure 3.14.

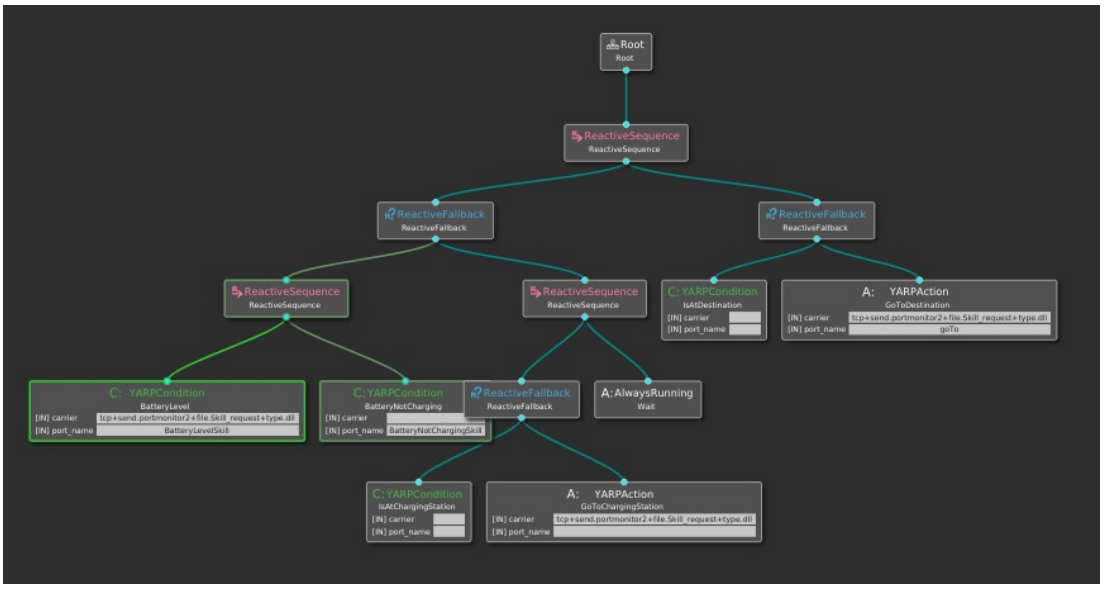
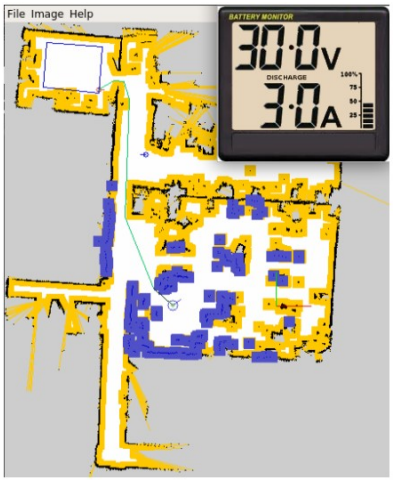
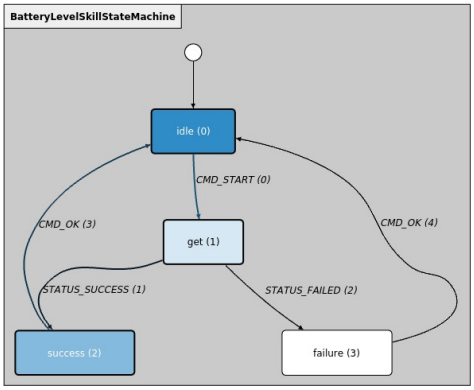


Figure 3.14: BT and Skills scheme

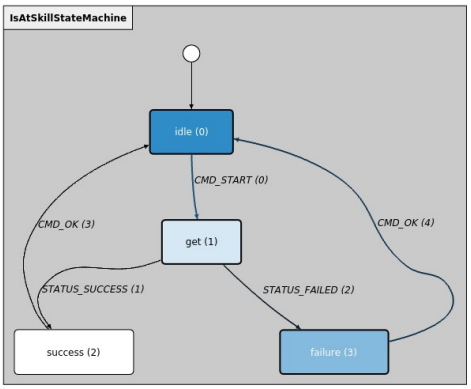
Scenario 1 (Nominal). This scenario corresponds to an execution without property violations. The robot moves towards the destination and a collision-free path is created (see Figure 3.15a). During the navigation the skill *BatteryLevel* keeps returning `SUCCESS` (Figure 3.15b) since the level is higher than 30%, the skill *isAtDestination* keeps returning `FAILURE` (Figure 3.15c) since the robot is still moving and the skill *BatteryNotCharging* keeps returning `SUCCESS` (Figure 3.15d) since the battery is not under charge.
While reaching the destination, the battery level goes below the 30% of its full capacity. Then, the robot stops navigating towards the destination and plans to reach the charging station (Figure 3.16a). The skill *BatteryLevel* keeps returning `FAILURE` during the navigation towards the charging station since the battery level is under the threshold of *Early Warning* (Figure 3.16b), the skill *isAtChargingStation* keeps returning `FAILURE` (Figure 3.16c) until the charging station is not reached and the skill *BatteryNotCharging* is in an `IDLE` state since the robot is waiting to be recharged (Figure 3.16d).
The robot then reaches the charging station and waits there until its battery is fully charged (Figure 3.17a). While charging, the skill *BatteryLevel* keeps returning `SUCCESS` as soon as the level gets higher than 30% (Figure 3.17b), the skill *isAtChargingStation* keeps returning `SUCCESS` (Figure 3.17c) since the robot is waiting to get fully recharged at the charging station and the skill *BatteryNotCharging* now returns `FAILURE` (Figure 3.17d) since the battery is under charge.
Once the battery gets charged, the robot moves again towards the destination. Finally, the robot reaches the destination (Figure 3.18a). The skill *BatteryLevel* returns `SUCCESS` (Figure 3.18b) since the level is higher than 30%, the skill *isAtDestination* returns `SUCCESS` (Figure 3.18c) since the robot has reached the destination and the skill *BatteryNotCharging* returns `SUCCESS` (Figure 3.18d) since the robot is not under charge.
During all these steps, the battery level never goes under 20%, so we will always have the runtime monitor GUI as in Figure 3.19.



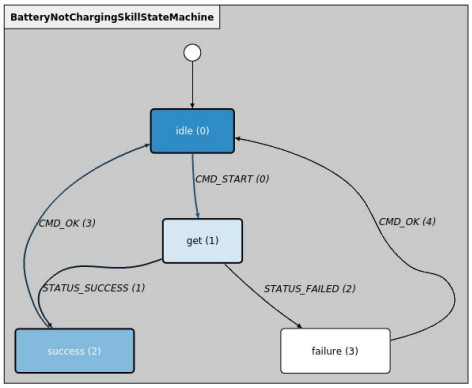
(a) Path creation from the initial position to the destination



(b) FSM of the skill *BatteryLevel*

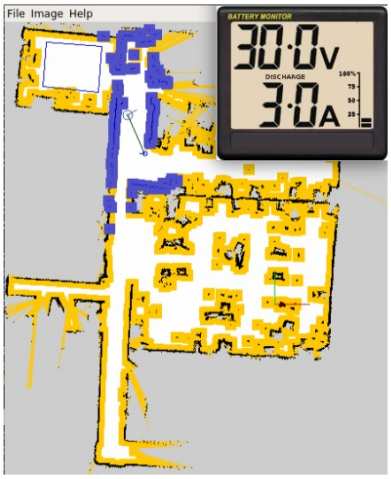


(c) FSM of the skill *isAtDestination*

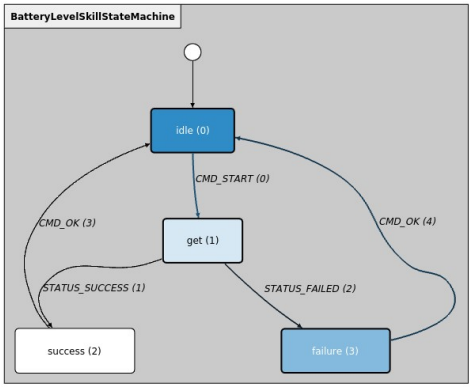


(d) FSM of the skill *BatteryNotCharging*

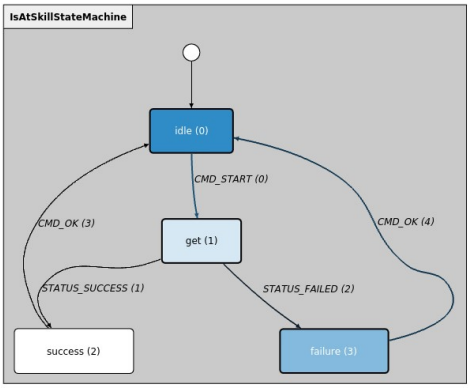
Figure 3.15: Robot going to destination, with battery charge >= 30%



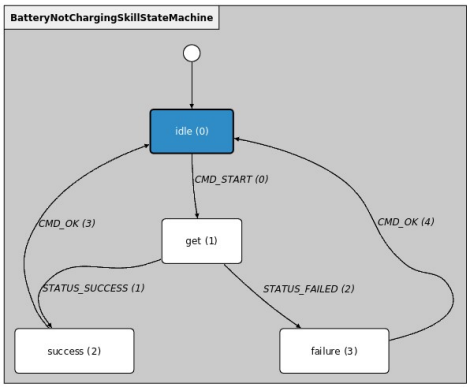
(a) Path creation from the actual position to the charging station



(b) FSM of the skill *BatteryLevel*

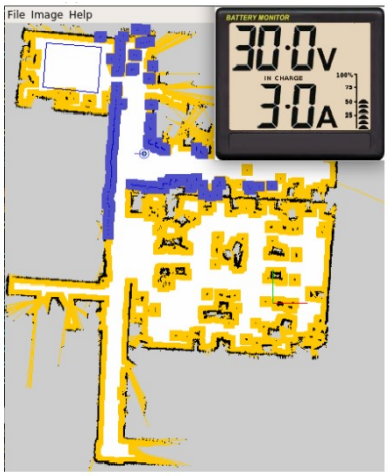


(c) FSM of the skill *isAtChargingStation*

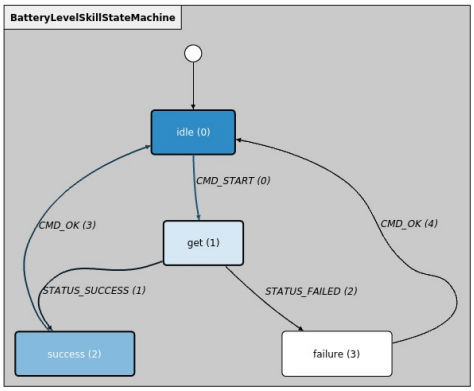


(d) FSM of the skill *BatteryNotCharging*

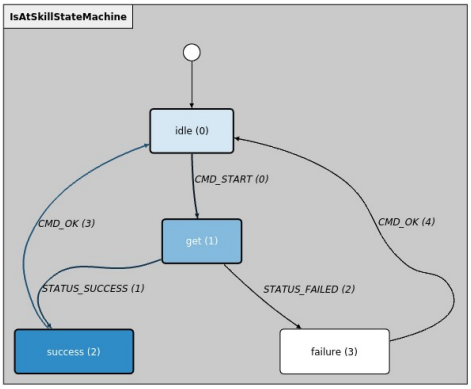
Figure 3.16: Robot going to charging station, with battery charge <= 30%



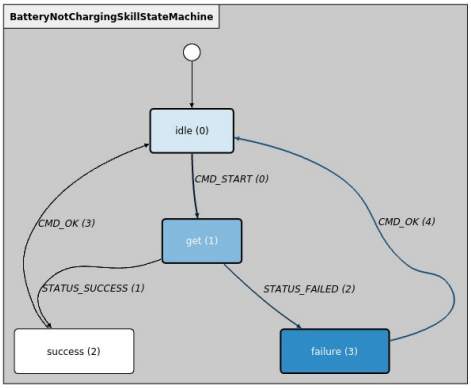
(a) Robot at the charging station



(b) FSM of the skill *BatteryLevel*

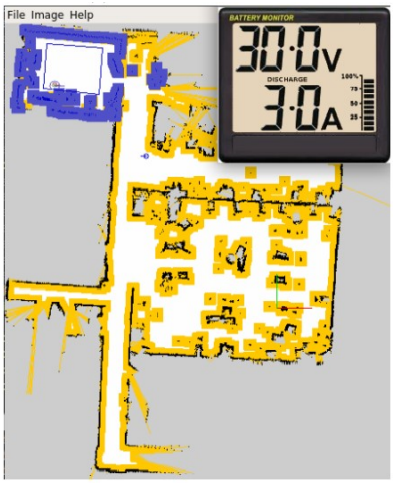


(c) FSM of the skill *isAtChargingStation*

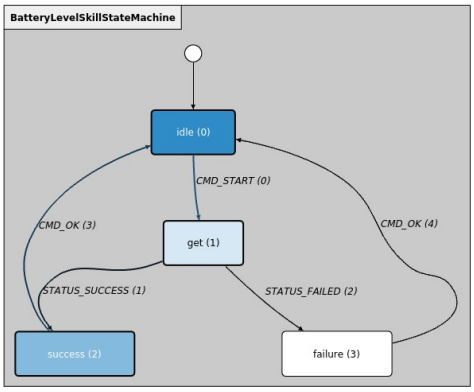


(d) FSM of the skill *BatteryNotCharging*

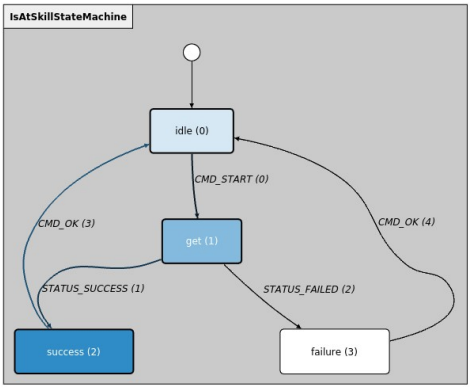
Figure 3.17: Robot at the charging station that is charging the battery



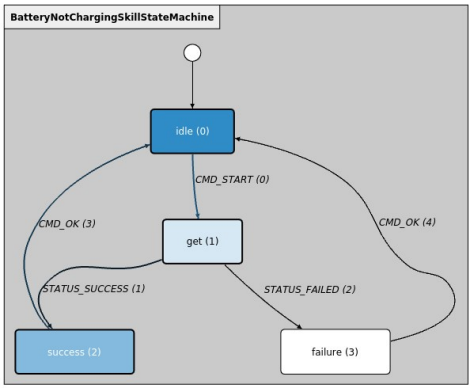
(a) Robot at the destination



(b) FSM of the skill *BatteryLevel*



(c) FSM of the skill *isAtDestination*



(d) FSM of the skill *BatteryNotCharging*

Figure 3.18: Robot at the destination

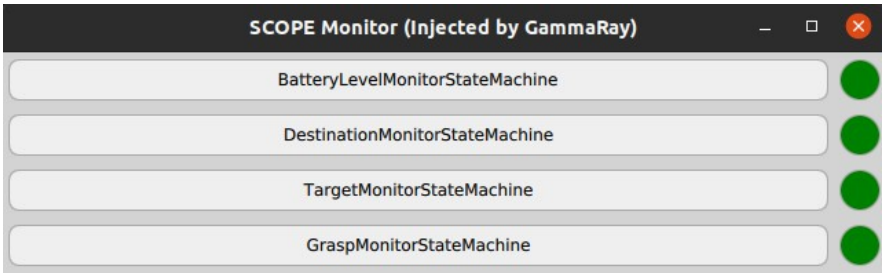
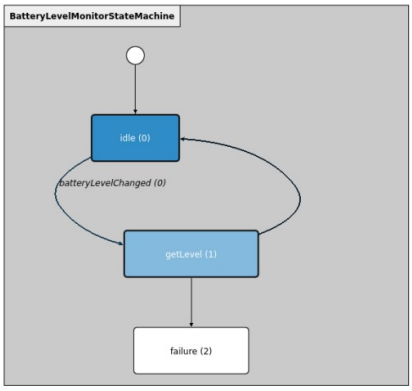
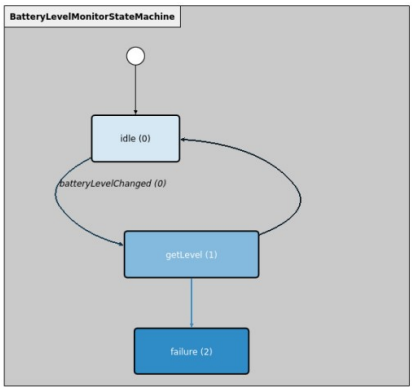


Figure 3.19: Runtime monitor GUI for Scenario 1

Scenario 2 (Safety Property Violation). This scenario shows the execution with a safety property violation. In particular, we show how runtime monitor detects the violation of the property "the battery never reaches below 20%". In this scenario the charging station is positioned in a point which is too far to be reached before the battery charge drops below 20%. The behaviour of the robot is the same of the previous example in Figure 3.15 and then Figure 3.16 with the exception that the charging station is not reached because the battery level gets too low. In Figure 3.20 we show the FSMs of the *BatteryLevelMonitor* before and after having reached 20% of the total charge, while in Figure 3.21 we can see the Monitor GUI in case of response `FAILURE` from the FSM of the monitor.



(a) Battery level 35%. The property is satisfied



(b) Battery level 19%. The property is violated

Figure 3.20: FSM of the runtime monitor for Scenario 2

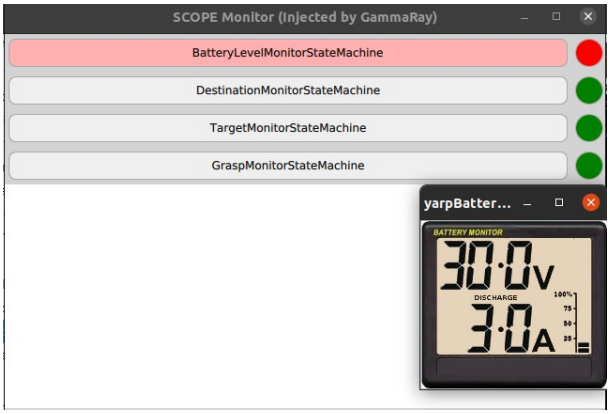
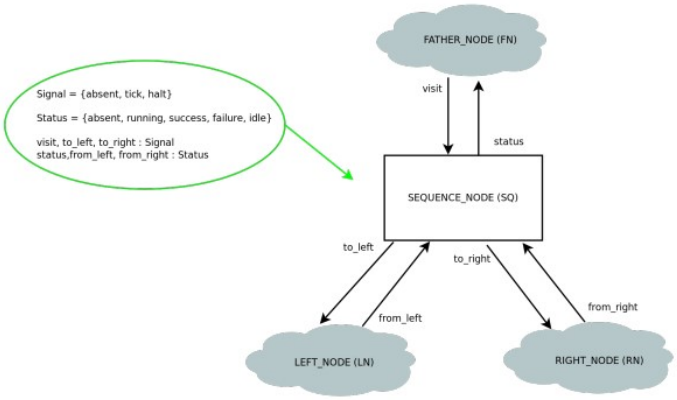


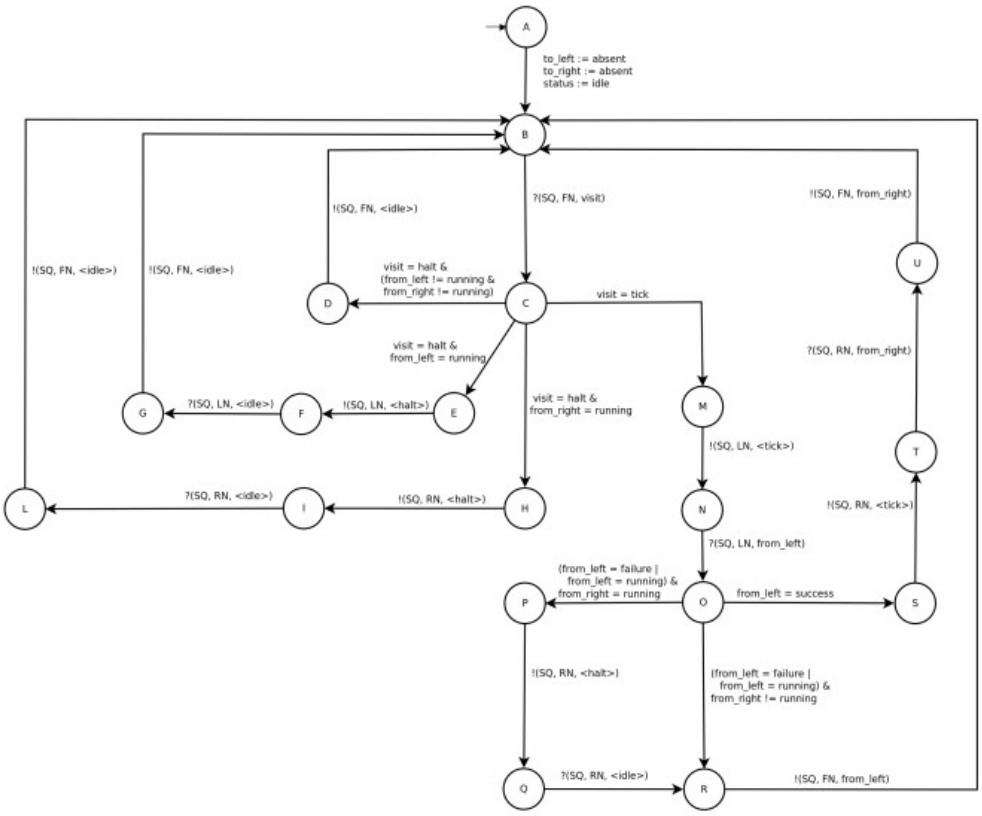
Figure 3.21: Runtime monitor GUI for Scenario 2

3.4 Off-line Simulator

The simulation framework previously described can be translated into C code with Posix Threads. This allows to execute several times the simulation in an off-line mode which is very useful in testing phase to save time. The idea behind the off-line simulator is to execute several possible scenarios (nominal or not) in a randomized way, but by maintaining the same structure (in terms of BT, skills and components) of the on-line simulator described in Section 3.3.
As mentioned, the simulator is based on Posix Threads, every thread implements one or more component of the architecture and it has its own execution function. Intuitively, handshaking communications are managed through function calls within the main thread, while asynchronous communications are managed by mutexes and conditional variables.
For example, if every communication between the components is synchronous, the translation result is a single thread, while if the communications between components are asynchronous, we will have as many threads as components numbers. Listing 3.3 shows the translation result for a sequence node whose intermediate representation is shown in Figure 3.22.
Each BT node is translated to a C function where return the type is $status \in \{\texttt{STATUS\_ABSENT}, \texttt{STATUS\_RUNNING}, \texttt{STATUS\_FAILURE}, \texttt{STATUS\_IDLE}\}$ and the argument is $visit \in \{\texttt{TICK}, \texttt{HALT}\}$. As for skills and to the components, each of them is translated into two different threads, a Skill Communication Manager (SCM) plus a skill implementation in the case of skills, and a Component Communication Manager (CCM) plus an abstract component implementation in the case of components. Given that communications are asynchronous, each element is represented by a single thread, whose the execution function implements a state machine using a *switch...case* statement on the current state value as shown in Listing 3.4 for the *BatteryLevel* condition skill.



(a) Sequence node



(b) Semantics of BT sequence node

Figure 3.22

```
status sequence1(btSignal visit) {
 static status _from_left = STATUS_ABSENT;
 static status _from_right = STATUS_ABSENT;
 // Wait
 if (visit == TICK) {
 // Tick left
  _from_left = fallback1(TICK);
  if ((from_left == STATUS_FAILURE || _from_left ==
   STATUS_RUNNING) &&
  _from_right == STATUS_RUNNING) {
   // Left HALT right
     do {
       _from_right = GoToDestination(HALT);
     } while (_from_right != STATUS_IDLE);
    // Return Left
     return _from_left;
   }
   if ((_from_left == STATUS_FAILURE || _from_left ==
   STATUS_RUNNING) &&
   _from_right != STATUS_RUNNING) {
    // Return Left
    return _from_left;
   }
   if (from_left == STATUS_SUCCESS) {
    // Tick right
    _from_right = GoToDestination(TICK);
   // Return right
    return _from_right;
   }
 }
 if (visit == HALT && _from_left == STATUS_RUNNING) {
 // Halt left
   do {
     _from_left = fallback1(HALT);
   } while (_from_left != STATUS_IDLE);
   // Return halted left
  return STATUS_IDLE;
  }
 if (visit == HALT && _from_right == STATUS_RUNNING) {
 // Halt right
   do {
     _from_right = GoToDestination(HALT);
   } while (_from_right != STATUS_IDLE);
```

```
42  // Return halted right
43  return STATUS_IDLE;
44  }
45 return STATUS_ABSENT;
46 }
```

Listing 3.3: Implementation in C language of a sequence node whose semantics is shown in Figure 3.22

```
1 ...
2  static scmState currentProtocolCond2Skill2State =
     SCM_WAIT_COMMAND;
3  ...
4  void * scmCondition2Skill2Execution (void * threadid) {
5  static skillSignal skillAck = SIG_ABSENT;
6  static commandSignal skillCmd = CMD_ABSENT;
7  static skill_request conditionCmd = absent_command;
8  while(1){
9    switch (currentProtocolCond2Skill2State)
10   {
11     case SCM_WAIT_COMMAND:
12     //Set skill command to absent
13     //Go to SCM_WAIT_COMMAND state
14     //Waiting for a command from condition node
15     //Based on command value, choose the next state and skill
     command
16     if(conditionCmd == send_start){
17     skillCmd = CMD_START;
18     currentProtocolCond2Skill2State = SCM_FWD_START;
19     }
20     else if(conditionCmd == send_ok){
21     skillCmd = CMD_OK;
22     currentProtocolCond2Skill2State = SCM_FWD_OK;
23     }
24     else if(conditionCmd == request_ack){
25     currentProtocolCond2Skill2State = SCM_RETURN_DATA;
26     }
27     pthread_yield();
28     pthread_yield();
29     break;
30
31     case SCM_FWD_START:
32     //Send CMD_START to skill in order to start the skill
```

```
33      //Wait until the skill reads the command and notifies for it
34      //Go back to SCM_WAIT_COMMAND state
35      pthread_yield();
36      pthread_yield();
37      break;
38
39      case SCM_RETURN_DATA:
40      //SCM sends the skill shared variable to condition node
41      //Go back to SCM_WAIT_COMMAND state
42      pthread_yield();
43      pthread_yield();
44      break;
45
46      case SCM_FWD_OK:
47      //Send CMD_OK to skill in order to start the skill
48      //Wait until the skill reads the command and notifies for it
49      //Go back to SCM_WAIT_COMMAND state
50      pthread_yield();
51      pthread_yield();
52      break;
53
54      default:
55      break;
56     }
57    }
58 }
```

Listing 3.4: Basic structure of the execution function for a thread that implements a Skill Communication Manager (SCM) for a condition skill

Methodology

4.1 Proposed Approach

The proposed approach is to learn the behaviours of the frameworks described in Section 3.3 and 3.4 using the tool *LearnLib*. Our aim is to isolate the interface Skills-Components (see the blue box in Figure 4.1) by focusing on the signals exchanged between them. Once intercepted these messages, the idea is to use automata learning to obtain a model describing the interaction between skill and components i.e., the complete behaviour of this interface.

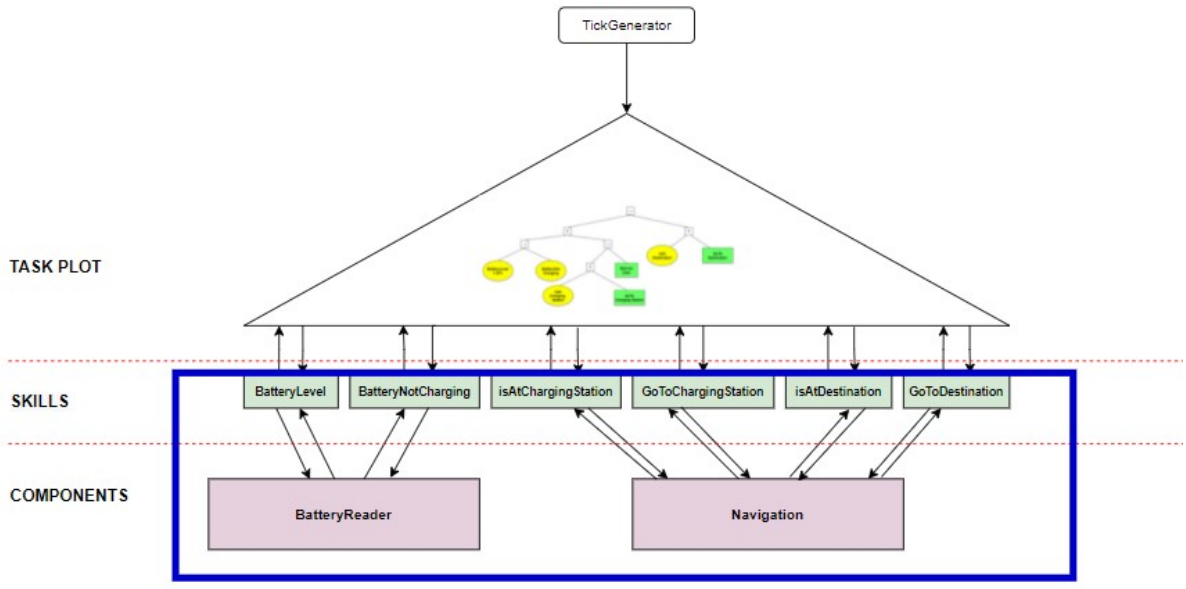


Figure 4.1: Blue box representing the interface Skill-Components in the coordination scheme of the framework

The kind of learning we implement is a passive learning using the algorithm RPNI with Mealy state machines. The choice of passive methods is dictated by the fact that it is relatively easy to collect all the messages exchanged between the components at various levels and store them in log files that can be used for off-line processing. On the other hand, active learning methods would require running the simulator up to a point and then resetting it to support membership queries. A system as the one that we are supposed to learn could be easily represented with I/O Automata, but this format is not supported on *LearnLib*. Because of this, we have chosen to learn Mealy state machines since it is possible to map from an I/O Automaton to a Mealy machine (see Section 2.3.2).
On the basis of these assumptions, we implemented an algorithm to use *LearnLib* outlined in Algorithm 3.

Algorithm 3: *LearnLib* algorithm

Input : A set of samples $L_s \in S^+$, an input alphabet I
Output : A Mealy Machine M

procedure :
  create L_s with $s \in L_s$ and $s = (prefix, suffix, output)$
  $learner = BlueFringeRPNIMealy(I)$
  $learner.addSamples(L_s)$
  $M = learner.computeModel()$
  return M

By looking at the algorithm we can notice that the input required is a set of samples L_s. Each sample is a touple $s = (prefix, suffix, output)$,where with *prefix* we refer to the set of inputs of a state machine before the observation, the *suffix* is the actual input to the machine, and the *output* is the set of resulting outputs. There are three *Learnlib* functions that are used in this algorithm: *BlueFringeRPNIMealy* is used to initialize the *learner*, *addSamples* is used to add all samples in L_s to the learner, and *computeModel* is used to obtain the learned model M. The resulting model is in format *.dot* (a text format to represent graphs used by the Graphiz[1] program) and can be saved by the user.
To see how the learning algorithm works we show two examples about learning simple Mealy state machines, one affected by non-determinism and the other completely deterministic.

[1] https://graphviz.org/

For both machines we have implemented a simple program that works with Algorithm 4 to generate the samples for *Learnlib*.

Algorithm 4: MealyMachine

Input : A Mealy machine M and an input alphabet I
Output : A sequence of outputs L_O over the output alphabet O
procedure :
$L_I = CreateRandomSequence(I)$
$L_O = M.get\_output\_from\_string(L_I)$
return L_O

The principle of this algorithm is to reproduce the behaviour of a Mealy state machine. The function $CreateRandomSequence(I)$ creates a sequence of length between 2 and 8 of elements randomly distributed over the alphabet I. For example if $I = \{a, b\}$ a result may be $L_I = \{bbabbb\}$.
The method $get\_output\_from\_string()$ is instead used to obtain a sequence L_O of outputs corresponding to the sequence L_I of inputs.

Example1 : non-deterministic Mealy machine

We now consider a non-deterministic example. In Figure 4.2 we show the structure of a simple machine, with input alphabet $I = \{a, b\}$ and an output alphabet $O = \{0, 1\}$.

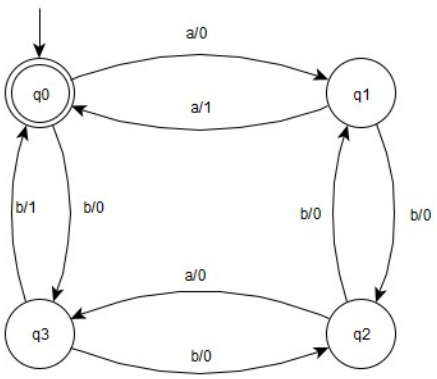
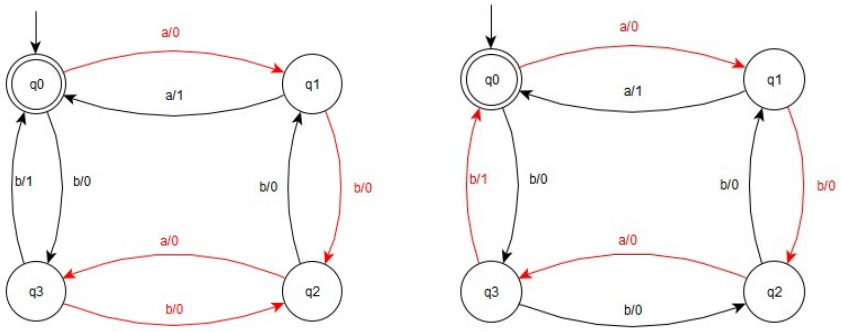


Figure 4.2: A non-deterministic Mealy machine

This machine is clearly non-deterministic, since for an input sequence $L_I = \{abab\}$ there is ambiguity in the output sequence than can be both $L_O = \{0000\}$ and $L_O = \{0001\}$ (see Figure 4.3).



(a) $L_O = \{0000\}$ (b) $L_O = \{0001\}$

Figure 4.3: Ambiguity in output for $L_I = \{abab\}$

Starting from this machine, we have converted it into a format compatible with Algorithm 4 (see Figure 4.4) in order to be able to get all the required samples.

```
mealy = Mealy(
    ['A', 'B', 'C', 'D'],
    ['a', 'b'],
    ['0', '1'],
    {
        'A' : {
            'a' : ('B', '0'),
            'b' : ('D', '0')
        },
        'B' : {
            'a' : ('A', '1'),
            'b' : ('C', '0')
        },
        'C' :{
            'a' : ('D', '0'),
            'b' : ('B', '0')
        },
        'D':{
            'b' : ('C', '0'),
            'b' : ('A', '1')
        }
    },
    'A'
)
```

Figure 4.4: Format of a Mealy state machine where A, B, C, D correspond to q_0, q_1, q_2, q_3 in Figure 4.2

Once the machine M is implemented, it is possible to obtain all the desired samples. These samples will be given as input to the *LearnLib* algorithm (3). In particular, for the example with $L_I = \{abab\}$ and $L_O = \{0000\}$ the sample will be $s = (aba, b, 0000)$, having as *prefix* all the sequence L_I with the exception of the last element, as *suffix* the last element of L_I and as *output* the entire L_O. Then by using Algorithm 4 it is possible to generate all the desired sequences of inputs and outputs to be given to *LearnLib*. Since the machine is non-deterministic, the execution of *LearnLib* will produce an exception, since non-determinism is not supported.

Example2 : deterministic Mealy machine

For this example we consider a deterministic Mealy machine (see Figure 4.5).

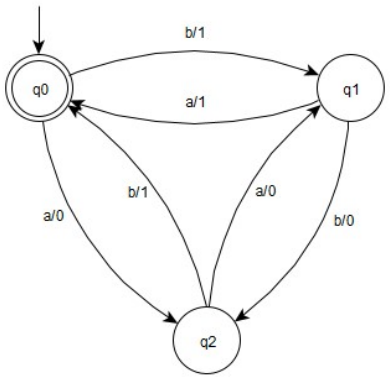


Figure 4.5: A deterministic Mealy machine

The working procedure is the same as the previous example, which includes the conversion into a suitable format and the generation of samples. The samples that have been used in *LearnLib* have been the following :

- $s_0 = \{aabbb, a, 000111\}$
- $s_1 = \{abbbbab, b, 01101011\}$
- $s_2 = \{bbbabab, b, 10101011\}$

With these samples given to the algorithm, the resulting machine M is the one in Figure 4.6. As we can notice, the machine learned is exactly the one in Figure 4.5, with the same states and transitions.

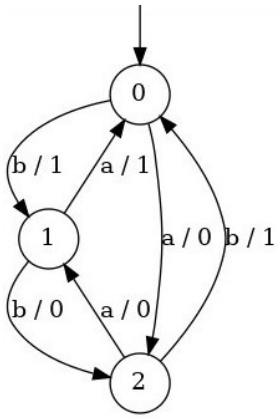


Figure 4.6: Mealy machine learned via *LearnLib*

4.2 Interface Abstraction

Since using *LearnLib* requires the definition of an input and output alphabet, and the complexity of the signals exchanged between skills and components is too high to be used directly, we need to define a suitable abstraction.
As first step we have collected all the messages exchanged between skills and components in one execution of the simulation (for this step it was indifferent which of the two simulators, since the format of messages is exactly the same). An example of a *log* file is shown Figure 4.7, where we have both messages exchanged between BTs and skills and the messages between skills and components.

```
71.818968       DEBUG           Reply intercepted:
From       : /GoToDestination/BT_rpc/server
To         : /GoToDestination/BT_rpc/client
Protocol   : Skill_request
Command    : request_ack
Arguments  :
Reply      : 1
72.125394       DEBUG           Command intercepted:
From       : /GoToGoToClient/kitchen
To         : /GoToComponent
Protocol   : GoTo
Command    : getStatus
Arguments  : kitchen
Reply      :
72.125520       DEBUG           Reply intercepted:
From       : /GoToComponent
To         : /GoToGoToClient/kitchen
Protocol   : GoTo
Command    : getStatus
Arguments  : kitchen
Reply      : 1
72.626740       DEBUG           Command intercepted:
From       : /GoToGoToClient/kitchen
To         : /GoToComponent
Protocol   : GoTo
Command    : getStatus
Arguments  : kitchen
Reply      :
72.637931       DEBUG           Reply intercepted:
From       : /GoToComponent
To         : /GoToGoToClient/kitchen
Protocol   : GoTo
Command    : getStatus
Arguments  : kitchen
Reply      : 1
```

Figure 4.7: Fragment of a communication log

Since we are concerned with the second kind of messages (Skills-Components), we have filtered the log file by keeping only the desired messages. After this preprocessing, we need an encoding to abstract the messages into an input and output alphabet suitable for *LearnLib*.
As input messages I have considered the ones from skills to components, while the ones from components to skills (i.e. the Reply) have been interpreted as output messages. Considering that in the log a single message is composed of 6 rows, we have to simplify it, and translate into a single string, to make the learning process possible. The encoding is shown in Table 4.1 and Table 4.2.

Input messages	
Original message	Encoded message
From : /GoToGoToClient/kitchen To : /GoToComponent Protocol : GoTo Command : goTo Arguments : kitchen Reply :	goTo_ki_i
From : /GoToGoToClient/kitchen To : /GoToComponent Protocol : GoTo Command : getStatus Arguments : kitchen Reply :	getStatus_ki_i
From : /GoToIsAtClient/kitchen To : /GoToComponent Protocol : GoTo Command : isAtLocation Arguments : kitchen Reply :	isAt_ki_i
From : /GoToGoToClient/kitchen To : /GoToComponent Protocol : GoTo Command : halt Arguments : kitchen Reply :	halt_ki_i

From : /BatteryReaderBatteryLevelClient To : /BatteryComponent Protocol : BatteryReader Command : level Arguments : Reply :	level_i
From : /BatteryReaderBatteryNotChargingClient To : /BatteryComponent Protocol : BatteryReader Command : charging_status Arguments : Reply :	batteryStatus_i
From : /GoToGoToClient/charging_station To : /GoToComponent Protocol : GoTo Command : goTo Arguments : charging_station Reply :	goTo_ch_i

From : /GoToGoToClient/charging_station To : /GoToComponent Protocol : GoTo Command : getStatus Arguments : charging_station Reply :	getStatus_ch_i
From : /GoToIsAtClient/charging_station To : /GoToComponent Protocol : GoTo Command : isAtLocation Arguments : charging_station Reply :	isAt_ch_i
From : /GoToGoToClient/charging_station To : /GoToComponent Protocol : GoTo Command : halt Arguments : charging_station Reply :	halt_ch_i

Table 4.1: Encoded input messages

Output messages	
Original message	Encoded message
From : /GoToComponent To : /GoToGoToClient/kitchen Protocol : GoTo Command : goTo Arguments : Reply : kitchen	o_goto_ki
From : /GoToComponent To : /GoToGoToClient/kitchen Protocol : GoTo Command : getStatus Arguments : kitchen Reply : 1	o_getStatus_ki_run
From : /GoToComponent To : /GoToGoToClient/kitchen Protocol : GoTo Command : getStatus Arguments : kitchen Reply : 2	o_getStatus_ki_suc
From : /GoToComponent To : /GoToGoToClient/kitchen Protocol : GoTo Command : getStatus Arguments : kitchen Reply : 3	o_getStatus_ki_ab

From : /GoToComponent To : /GoToIsAtClient/kitchen Protocol : GoTo Command : isAtLocation Arguments : kitchen Reply : 0	o_isAt_ki_false
From : /GoToComponent To : /GoToIsAtClient/kitchen Protocol : GoTo Command : isAtLocation Arguments : kitchen Reply : 1	o_isAt_ki_true
From : /GoToComponent To : /GoToGoToClient/kitchen Protocol : GoTo Command : halt Arguments : kitchen Reply :	o_halt_ki
From : /BatteryComponent To : /BatteryReaderBatteryNotChargingClient Protocol : BatteryReader Command : charging_status Arguments : Reply : 0	o_batteryStatus_false

From : /BatteryComponent To : /BatteryReaderBatteryNotChargingClient Protocol : BatteryReader Command : charging_status Arguments : Reply : 1	o_batteryStatus_true
From : /GoToComponent To : /GoToGoToClient/charging_station Protocol : GoTo Command : goTo Arguments : Reply : charging_station	o_goto_ch
From : /GoToComponent To : /GoToGoToClient/charging_station Protocol : GoTo Command : getStatus Arguments : charging_station Reply : 1	o_getStatus_ch_run
From : /GoToComponent To : /GoToGoToClient/charging_station Protocol : GoTo Command : getStatus Arguments : charging_station Reply : 2	o_getStatus_ch_suc

From : /GoToComponent To : /GoToGoToClient/charging_station Protocol : GoTo Command : getStatus Arguments : charging_station Reply : 3	o_getStatus_ch_ab
From : /GoToComponent To : /GoToIsAtClient/charging_station Protocol : GoTo Command : isAtLocation Arguments : charging_station Reply : 0	o_isAt_ch_false
From : /GoToComponent To : /GoToIsAtClient/charging_station Protocol : GoTo Command : isAtLocation Arguments : charging_station Reply : 1	o_isAt_ch_true
From : /GoToComponent To : /GoToGoToClient/charging_station Protocol : GoTo Command : halt Arguments : charging_station Reply :	o_halt_ch

Table 4.2: Encoded output messages

Regarding the output message coming from the *BatteryComponent* to the related skill, we have a format as the one shown below, where the *Reply* field contains a floating point number indicating the level of the battery during the execution.

From : /BatteryComponent
To : /BatteryReaderBatteryLevelClient
Protocol : BatteryReader
Command : level
Arguments :
Reply : 97.316999999987189085

The encoding for this message has been done by dividing the interval $[0, 100]$ into ranges and obtaining the following output messages :

- $100 < level < 30$: o_level_high
- $30 < level < 20$: o_level_medium
- $20 < level < 0$: o_level_low
- $level = 0$: o_level_zero

During the execution, since each message is repeated several times, the total amount of messages could be too big to be managed by *LearnLib*. To remedy this, we have made another abstraction to reduce the number of messages and consider only the ones needed for the learning purpose.
This further abstraction is implemented as shown in algorithm *Compactor*, whose implementation is shown in Algorithm 5, where we assume to have a set of traces $T = \{t_1, ..., t_n\}$, that are the input and output messages coming from the encoding previously shown, and a series of queries $Q = \{q_1, ...q_m\}$.
Each $q_i \in Q$ is defined as $q_i = (input, outputs, last)$ where *input* is the input message for that query (i.e. *isAt_ki_i*), *outputs* are all the possible output messages for the indicated input (i.e. *o_isAt_ki_false* and *o_isAt_ki_true*), and *last* is the last output message that has been read.

Algorithm 5: Compactor

Input : T and Q
Output : A new set of abstract traces T_{abs}

```
procedure :
    initialize indexes variable
    for j = 0 : n do
        if T[j] == q_i.input then
            for k = (j + 1) : n do
                for r = 0 : m do
                    if T[k] == q_i.outputs[r] and q_i.outputs[r]! = q_i.last then
                        indexes.append = j
                        indexes.append = k
                    end if
                end for
            end for
        end if
    end for
    indexes.sort()
    for l = (0 : len(indexes)) do
        T_abs = T[indexes[l]]
    end for
    return T_abs
```

The main idea of this algorithm is to keep only relevant messages, so if one input message and its related output is repeated several times (i.e. *isAt_ki_i* and the answer *o_isAt_ki_false*), we will write it only once, writing again the input message *isAt_ki_i*, only when the output message will trigger to a different value, which means when we will have *o_isAt_ki_true*.
To better understand this procedure we consider Figure 4.8 : in Figure 4.8a we show a fragment of log already abstracted, and in Figure 4.8b we show the corresponding compact version. As we can see, the repeated messages are not kept, but the original order of how they are produced by the system is maintained, in order to avoid changing the logic that behind the framework.
Notice that in this example, several output messages do not appear. This is the case of *o_isAt_ki_true* since this message would came up towards the end of the execution, indicating that the robot has reached the target position. However when this message shows up in the log it will be kept together with the related

input because the previous output message was *o_isAt_ki_false*.

```
level_i
o_level_high
batteryStatus_i
o_batteryStatus_false
isAt_ki_i
o_isAt_ki_false
goTo_ki_i
level_i
batteryStatus_i
o_batteryStatus_false
isAt_ki_i
o_isAt_ki_false
level_i
batteryStatus_i
o_batteryStatus_false
isAt_ki_i
o_isAt_ki_false
batteryStatus_i
o_batteryStatus_false
isAt_ki_i
o_isAt_ki_false
batteryStatus_i
o_batteryStatus_false
isAt_ki_i
o_isAt_ki_false
level_i
o_goto_ki
```

(a) Set of abstract traces T

```
level_i
o_level_high
batteryStatus_i
o_batteryStatus_false
isAt_ki_i
o_isAt_ki_false
goTo_ki_i
o_goto_ki
```

(b) Set of compact traces T_{abs}

Figure 4.8: Compaction of abstract traces

Once obtained the abstract and compact traces, we have to process them further because they correspond to the traces of an I/O Automaton, while the algorithm in *Learnlib* produces a Mealy state machine.
To overcome this issue, a possible solution is to apply the mapping from I/O automaton to Mealy machine. This, in practice, can be obtained by adding an input message *Delta* and an output message *OK*, that are a sort of dummy input and output messages from the system, related to the real input and output

messages.
Therefore this means that every input message will produce the output *OK*, while every output message will be a response of an input *Delta*.
So, by referring to the traces in Figure 4.8b, the resulting mapping will lead to what shown in Figure 4.9.

```
level_i/OK
Delta/o_level_high
batteryStatus_i/OK
Delta/o_batteryStatus_false
isAt_ki_i/OK
Delta/o_isAt_ki_false
goTo_ki_i/OK
Delta/o_goto_ki
```

Figure 4.9: Example of the mapped traces

The last step to be able to learn a model from these traces, is to split them into two sets (input and output), that will form a sample $s \in L_s$ to be fed to Algorithm 3.
In Figure 4.10a and Figure 4.10b there are the results from splitting the data in Figure 4.9.

```
level_i
Delta
batteryStatus_i
Delta
isAt_ki_i
Delta
goTo_ki_i
Delta
```

(a) Set of inputs

```
OK
o_level_high
OK
o_batteryStatus_false
OK
o_isAt_ki_false
OK
o_goto_ki
```

(b) Set of outputs

Figure 4.10: I/O split

If we consider L_I as the list of all the inputs in Figure 4.10a without the last element (*Delta*), and L_O as the list of all the outputs in Figure 4.10b, we can express $s \in L_s$ as :

$$s = (L_I, "Delta", L_O)$$

Given the definition of s, we are finally able to compute a model for a single sample or for a list of samples of the same system.
If we consider again the reduced log shown in the previous example, the resulting model is depicted in Figure 4.11 where the initial state is indicated with an entering arrow.

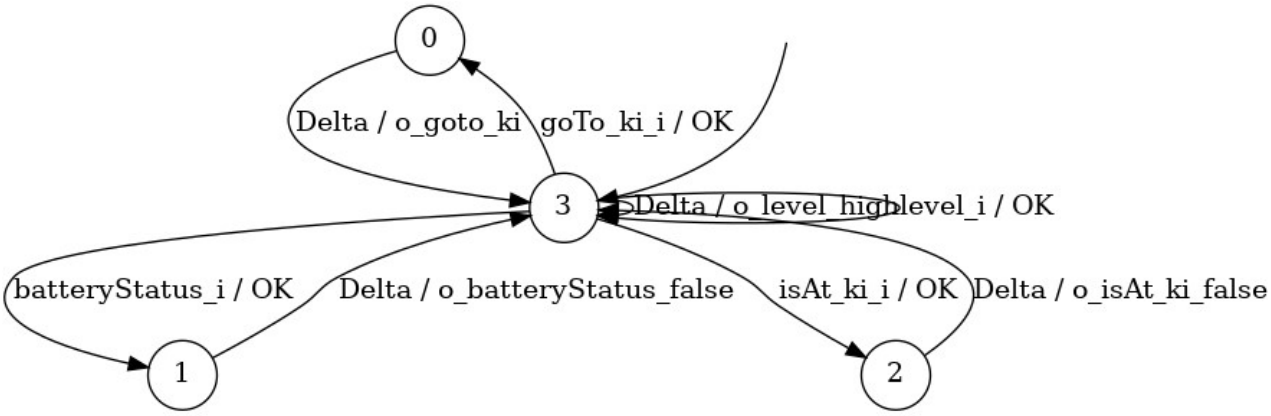


Figure 4.11: Learned model from Figure 4.10

4.3 Verification of the Learned Model

Once obtained a model in *LearnLib*, we want to verify that what has been learned is significant and correct. The format in which I saves the model is not executable, but since the goal is to be able to execute the model, we need to have a different format, something similar to what previously shown in Figure 4.4.
To achieve this we have implemented a *dot_to_Mealy* converter that, starting from a digraph (.dot format) creates a Mealy machine suitable for our purposes (see Figure 4.12).

```
digraph g {

        s0 [shape="circle" label="0"];
        s1 [shape="circle" label="1"];
        s2 [shape="circle" label="2"];
        s3 [shape="circle" label="3"];
        s0 -> s3 [label="Delta / o_goto_ki"];
        s1 -> s3 [label="Delta / o_batteryStatus_false"];
        s2 -> s3 [label="Delta / o_isAt_ki_false"];
        s3 -> s3 [label="Delta / o_level_high"];
        s3 -> s0 [label="goTo_ki_i / OK"];
        s3 -> s3 [label="level_i / OK"];
        s3 -> s1 [label="batteryStatus_i / OK"];
        s3 -> s2 [label="isAt_ki_i / OK"];

__start0 [label="" shape="none" width="0" height="0"];
__start0 -> s3;

}
```

(a) .dot of the example in Figure 4.11

```
Mealy(['0','1','2','3'],
['Delta','goTo_ki_i','level_i','batteryStatus_i','isAt_ki_i'],
['o_goto_ki','o_batteryStatus_false','o_isAt_ki_false','o_level_high','OK'],
{
        '0' : {
                'Delta' : ('3','o_goto_ki')
        },
        '1' : {
                'Delta' : ('3','o_batteryStatus_false')
        },
        '2' : {
                'Delta' : ('3','o_isAt_ki_false')
        },
        '3' : {
                'Delta' : ('3','o_level_high'),
                'goTo_ki_i':('0','OK'),
                'level_i':('3','OK'),
                'batteryStatus_i':('1','OK'),
                'isAt_ki_i':('2','OK')
                }
},
'3')
```

(b) Mealy format conversion

Figure 4.12: *dot_to_mealy*

To verify the correctness of the learned model, we have implemented an algorithm (6) similar to the one described before (4).

Algorithm 6: MealyMachine

Input : A Mealy machine M and a sequence of inputs L_I
Output : A sequence of machine outputs L_{OM} over the output alphabet O

procedure :
$L_{OM} = M.get\_output\_from\_string(L_I)$
return L_{OM}

This algorithm, differently from the previous one, takes in input an already created sequence of inputs computing the corresponding sequence of outputs on the basis of the working principle of the Mealy machine.
Therefore, if our aim is to verify the learned model, we can use as L_I the set of inputs in Figure 4.10a, as Mealy machine the one in Figure 4.12b, and we expect to get a set of machine outputs L_{OM} coincident to the set of outputs in Figure 4.10b.

4.4 Off-line Trace Classification

Our goal is to implement Predictive Monitoring which consists in the detection of a faulty scenario before the occurrence of a dangerous situation. By referring to the case study, we wish to understand if the battery level of the robot will reach a value under 20% of its full charge (faulty scenario), before the intervention of the runtime monitoring module described in the previous chapter.
To do this we need to classify a given trace (coming from the execution *log*) as a the prefix of nominal trace or the prefix of a trace with faults. As first attempt we can do this offline, which means that we record a session of execution, apply to this trace the procedure of abstraction, obtaining L_I and L_O and then perform a *MatchTraces* algorithm to verify in which case study we are (see Algorithm 7). As we can see this algorithm takes as input the aforementioned L_I and L_O and a set of Mealy Machines M_i. This set includes both nominal and faulty machines that have been learned offline in a previous phase by applying the algorithms described in Section 4.1 and 4.2. The aim of this algorithm is to recognize a trace belonging to $F_{T1,..Tm}$ (set of m Mealy machines with faults) in order to notify the user. The idea is that given a sequence of inputs L_I, we want to compute the relative sequence of outputs for each machine M_i obtaining L_{O_i}. This is done with the function $get\_output\_from\_string()$ that has been described in Section

4.1. The computation of L_{O_i} for a given machine can lead to an exception, that is handled by considering the machine as removed from the matching procedure. For the remaining machines, we use a function *ComputeDiscrepancies* to obtain how much L_{O_i} is different from the real sequence of output L_O. This discrepancy rate ($disc_i$) is the number of times that a single element in L_{O_i} in a certain position (i.e. *o_level_high* at index k) is different from the element in the same position in L_O (i.e. *o_level_low* at index k). Then, once all the discrepancy rates are computed, the following step is to select the smallest one (remaining under a certain *threshold*), and the machine with the selected rate will be the matching one. If the machine is of type $F_{T1,..Tm}$, this will trigger a $Warning\_Procedure()$.

Algorithm 7: MatchTraces

Input : A sequence of inputs L_I, sequence of output L_O and a series of Mealy machines M_i, where $i \in \{N_{T1,..Tn}, F_{T1,..Tm}\}$
Output : Warning in case of $F_{T1,..Tm}$

procedure :
  Initialize $deleted_i$ to *False*
  Initialize $disc_i$ to zero
  try:
    $L_{O_i} = M_i.get\_output\_from\_string(L_i)$
  catch *KeyError*:
    // Deleted trace i
    $deleted_i = True$
  end try
  if $deleted_i == False$ **then**
    $disc_i = ComputeDiscrepancies(L_O, L_{O_i})$
  end if
  $i = min(disc_i)|_{disc_i<threshold}$
  if i == $F_{T1,..Tm}$ **then**
    $Warning\_Procedure()$
  end if

4.5 On-line Trace Classification

The procedure described in the previous section can be implemented also online. This enables detection of a trace with faults in advance, before that the fault is detected by the safety monitor. To achieve this, we have implemented an

algorithm (see Algorithm 8) that processes log file while it is being produced.
The algorithm calls all the functions used for the offline part: *TracesCreator* encodes the log following the rules previuosly shown in Table 4.1 and Table 4.2, then we have *Compactor* for the compaction process, then *InputOutput_Split* maps from I/O Automaton to Mealy machines and to obtain L_I and L_O, and finally *MatchTracesRT* whose implementation is very similar to Algorithm 7, but adapted to the runtime application.

Algorithm 8: Online processing

procedure :
  while (1) **do**:
    Load *log* file
    $TracesCreator()$
    $Compactor()$
    $InputOutput_Split()$
    $MatchTracesRT()$

The idea behind *MatchTracesRT* is that we do not want to be informed that a trace is of type $F_{T1,..Tm}$ at the end (when choosing the one with less discrepancies), but we want an immediate action. To achieve this, the algorithm keeps computing the discrepancy rate for each machine (the ones that are still not removed) : if this rate is over a certain *threshold*, the machine with that rate is removed too. At the end, if the remaining trace has a rate under the *threshold*, we can conclude that it is the matched trace, and we can check if it is a nominal machine or a machine with faults.
In conclusion, *MatchTracesRT* is the algorithm that actually performs the predictive monitoring, since it takes as input a trace of execution (couple of input and output sets) and recognizes if it is a trace of type $F_{T1,..Tm}$, which means a faulty one, by progressively deleting the nominal ones that are too far from the given trace.

Implementation and Experimental Results

5.1 Learning and Off-line Traces Classification

Our goal is to learn the interface Skills-Components of the proposed frameworks. From a practical point of view we have first tried the procedure with the off-line simulator (3.4) in order to save time and make the necessary tests. Then, once verified the correctness of the procedure, we have repeated it with the on-line simulator (3.3).
The idea that is behind both approaches consists in learning three different machines, each of them corresponding to a precise scenario. The chosen scenarios are two nominal (i.e., the robot reaches the destination without violating any requirement) and a scenario with faults (i.e., a requirement is violated).
In particular they are :

- **Scenario 1 (nominal)** : the robot reaches the destination and the level of the battery never gets under 30% of its capacity.
- **Scenario 2 (nominal)** : the robot is reaching the destination, but it is too far, so the battery level gets lower than 30%. This makes the robot interrupting its navigation task and reaching the charging station to get the battery fully charged. Once that the battery is charged the robot reaches the destination.
- **Scenario 3 (with faults)** : the robot is reaching the destination, but it is too far, so the battery level gets lower than 30%. This makes the robot interrupting its navigation task. The robot tries to reach the charging

station, but also the charging station is too far. As consequence the battery level gets lower than 20% violating a requirement and notifying this to the user.

5.1.1 Off-line Simulator

This simulator offers the possibility to obtain the desired scenarios. By running it, we can get both nominal scenarios and the one with faults (according to what described before). The simulator immediately provides us the *log* file that will be used to learn a model.
For each scenario we have the following steps to perform :

- **Step 1** : Run the simulator obtaining the *log* file.
- **Step 2** : Process the *log* as described in Section 4.2 obtaining L_I and L_O.
- **Step 3** : Fed L_I and L_O to *LearnLib* as described in Section 4.1 and save the learned model as .dot file.
- **Step 4** : Convert the model into a Mealy format with a *dot_to_Mealy* converter (see Section 4.3)
- **Step 5** : Verify with Algorithm 6 that the set of outputs L_{OM} is the same as L_O.

As next step, we have tested the learned model for other executions belonging to the same scenario in order to stress the strength of the model. In practice, we have repeated Step 1 and Step 2, skipping the steps related to the learning and passing directly to Step 5 by running Algorithm 6. This could lead to three different cases:

- **Case 1 (positive)** : $L_{OM} = L_O$ which means that the model is able to manage the new trace.
- **Case 2 (negative)** : $L_{OM} \neq L_O$ which means that even if the model manages the trace, it is still incomplete and not able to do it in the proper way.
- **Case 3 (negative)** : *Key exception* which means that the model is not able to manage the trace.

In Case 2 and Case 3 (the negative ones) the next step is the same : the model needs to be improved, and this can be done by adding to *LearnLib* a new sample composed of the trace that causes $L_{OM} \neq L_O$ or *Key Exception*, so we have to perform Step 3, Step 4 and Step 5. Once learned the model with the additional sample we repeat again the testing phase until a solid model is found for each scenario. As result we have obtained that for easier scenarios (Scenario 1) i.e., with a lower number of messages, the samples needed to get a proper model have been lower, while with more complex scenarios (Scenario 2), we needed to add more samples to *LearnLib* to get the same result. At the end of this procedure we have obtained three models, one for each scenario. In particular, in Figure 5.1 is shown the resulting model for Scenario 1.

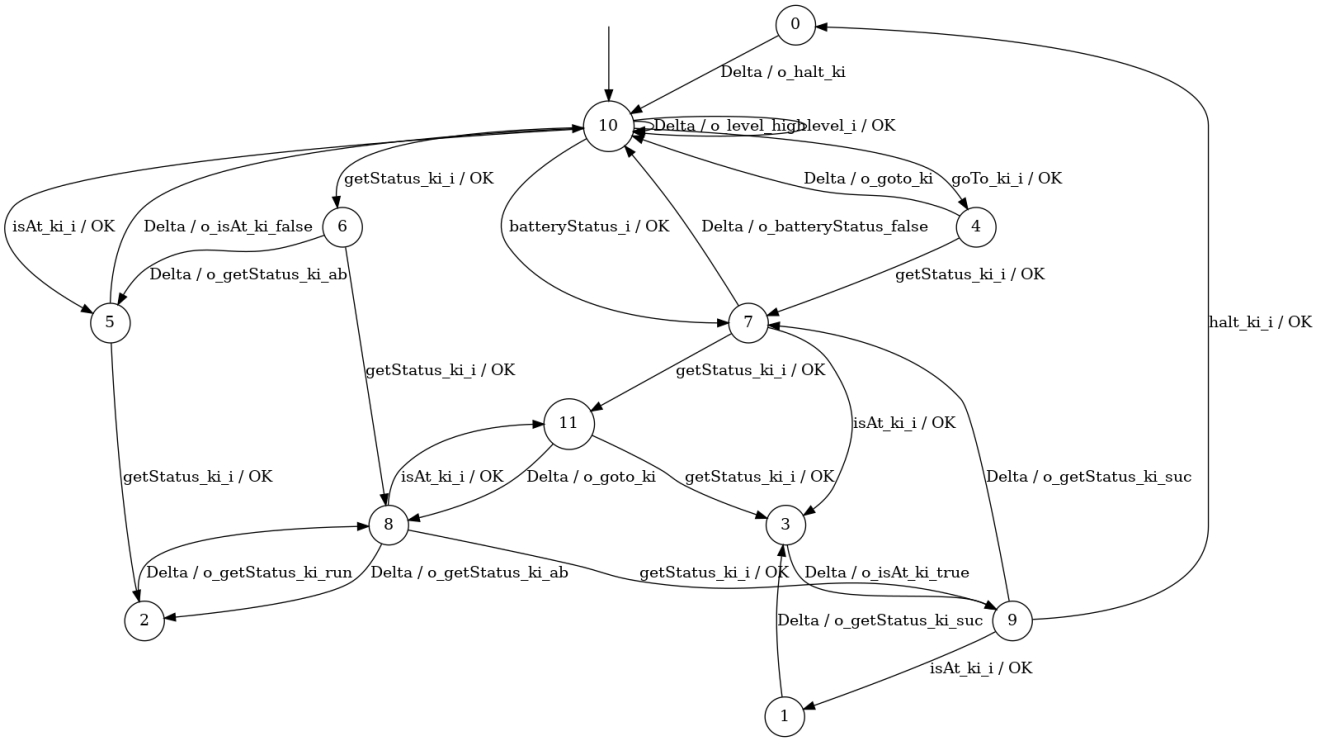


Figure 5.1: Mealy machine for Scenario 1

This model represents the behaviour of the robot If we refer to the example in Section 4.2 and to the machine obtained (Figure 4.11) we can notice that this smaller machine is embodied in the one in Figure 5.1 since it represented the initial execution of the robot, so the first messages exchanged. In Figure 5.2 is highlighted in red how the initial transitions of the machine for Scenario 1 are

the same of the smaller machine.

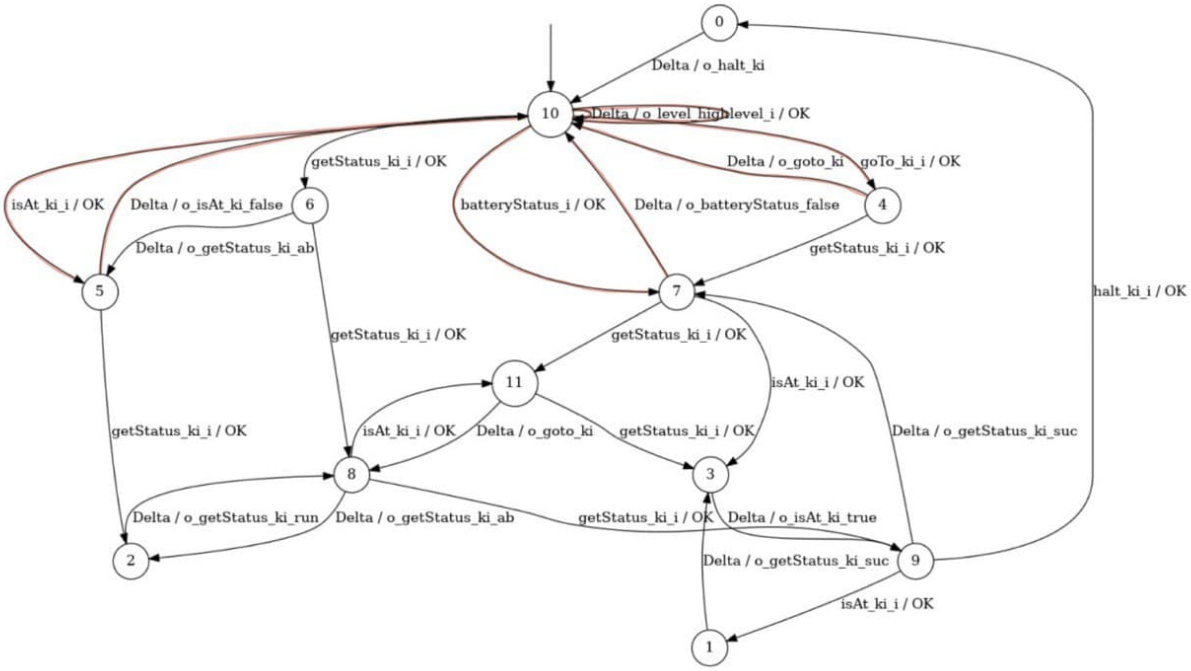


Figure 5.2: Highlight of machine in Figure 4.11 for Scenario 1

We can argue that this model represents exactly our scenario. If we look at the other messages (i.e., the transitions) it is possible to notice that there are no unexpected messages related for example to low levels of battery or to a navigation to the charging station (e.g. *o_level_medium*, *isAt_ch..*). We also have that the execution of the machine ends with transition messages indicating that the robot has reached the destination as it should be in this scenario. In particular if we refer to the transitions highlighted in blue in Figure 5.4 we can see that the last messages are :

```
isAt_ki_i/OK
Delta/o_isAt_ki_true
halt_ki_i/OK
Delta/o_halt_ki
```

Figure 5.3: Last messages exchanged in Scenario 1

These messages indicate that the destination has been reached, since the skill *IsAtDestination* returns `SUCCESS` and so we have the encoded output message *o_isAt_ki_true*. Then since the navigation to the destination is terminated we have the input and output halt messsages (*halt_ki_i* and *o_halt_ki*) confirming again that this representation of the model is coherent.

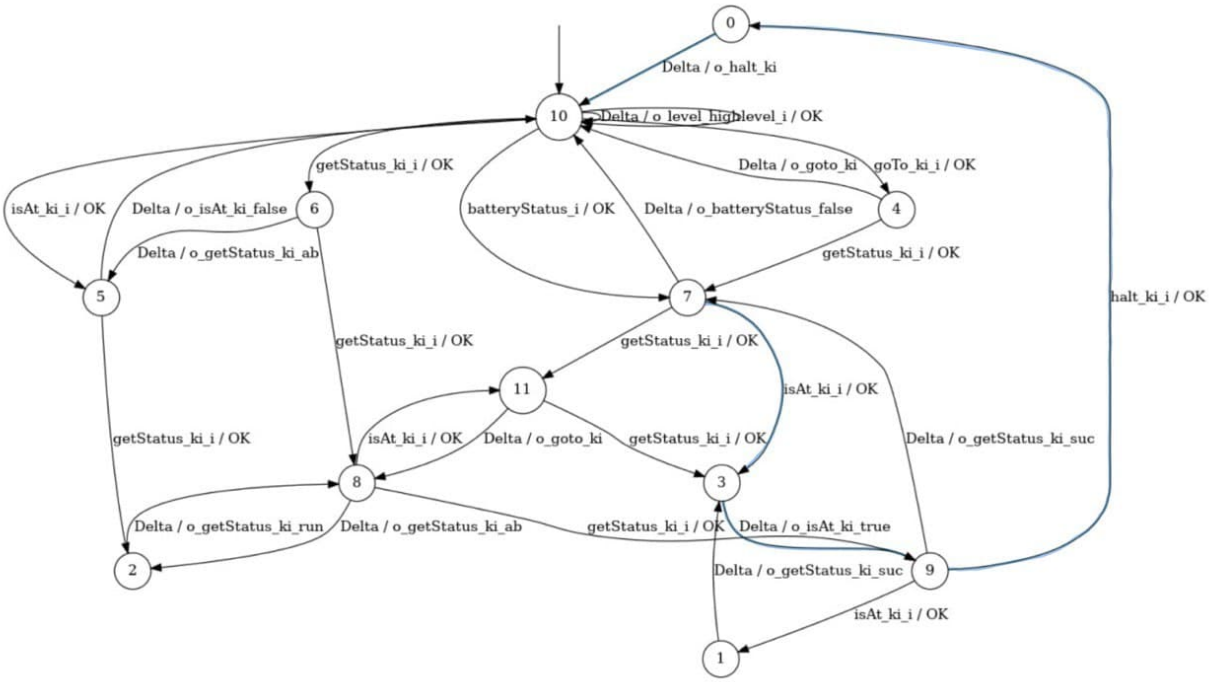


Figure 5.4: Highlight of last messages for Scenario 1

When the models for the three scenarios have been learned, we have proceed with the classification phase. Here we repeated again Step 1 and Step 2 in order to get a new trace, and then we have given this as input to Algorithm 7 to check if the trace belongs to the faulty scenario (Scenario 3). As result we have obtained that whenever we give as input to the algorithm a couple of input and output sets relative to a trace belonging to Scenario 3, the classifier recognizes it.

5.1.2 On-line Simulator

Regarding the on-line simulator, the procedure is exactly the same adopted for the off-line simulator. The only difference is related to how we obtain the *log*

file. In fact for the off-line simulator we have that the file is produced whenever the program is run, while for the on-line simulator we need to record the messages during the session of execution (i.e., while the robot is moving towards the destination or towards the charging station). This can be achieved by exploiting the runtime monitoring and in particular the MonitorObjects that intercepts and forewards messages to the MonitorReader. The tool used to intercept these messages is *yarplogger* [1]. In Figure 5.5 is shown the GUI of *yarplogger*: on the left side is selected from where we want to read (`/log/ubuntu/monitor/..`), while on the right side there are the messages of interest. Once intercepted these messages, they can be redirected to a *log* file and we can repeat Step 1, Step 2 and so on. As first attempt we have adopted an off-line approach, which means that we have executed one session belonging to one of the scenarios proposed in Section 5.1 recording all the messages and redirecting them on a *log* file. At the end of the execution the file will represent the entire communication *log* of the robot, and can be used for learning purposes applying everything described before.

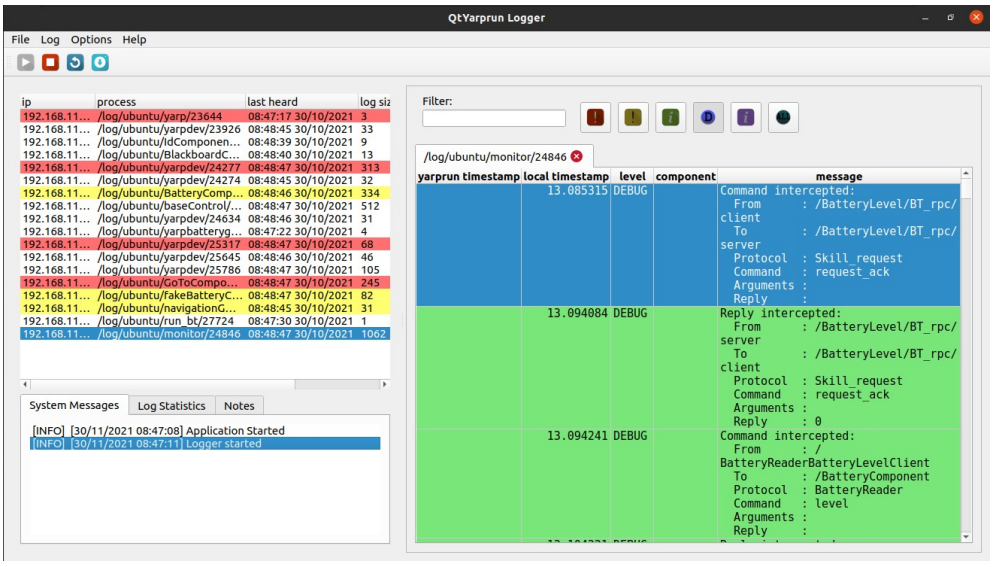


Figure 5.5: *yarplogger* GUI

Also in this case we get the same result when applying Algorithm 7, which means that again we are able to distinguish a trace belonging to Scenario 3 among other nominal scenarios. These results obtained in the off-line mode (for both simulators), have laid the groundwork for the predictive monitoring.

[1] http://www.yarp.it/git-master/yarp_logging.html

5.2 Predictive Monitoring

Once tested everything in off-line mode, we need to repeat everything at runtime in order to implement the predictive monitoring. Our goal is, in fact, to anticipate the already implemented runtime monitoring, so we basically want to understand if a trace contains faults before the Monitor i.e., understand that we are in Scenario 3 before that the battery level gets lower than 20%.
From a practical point of view, with the actual requirements this cannot be done, since in the on-line simulator, with an *Early Warning* of 30% it may happen that the robot reaches the charging station at almost 20% of battery. This means that the eventual predictive monitoring would recognize the faulty trace at the same time of the monitoring or even later. The cause of this is that up to the moment where the skill *isAtChargingStation* triggers to `SUCCESS` in Scenario 2, Scenario 2 and Scenario 3 are the same (in terms of exchanged messages). See in Figure 5.6 a partial abstract trace for Scenario 2 (on the right) and an abstract trace for Scenario 3 (on the left). As we can see the two traces are different only for the last messages, and this could be a problem, since when the output message coming from the *BatteryReader* component is *o_level_low*, the runtime monitoring intervenes because this means that the level is under 20%.

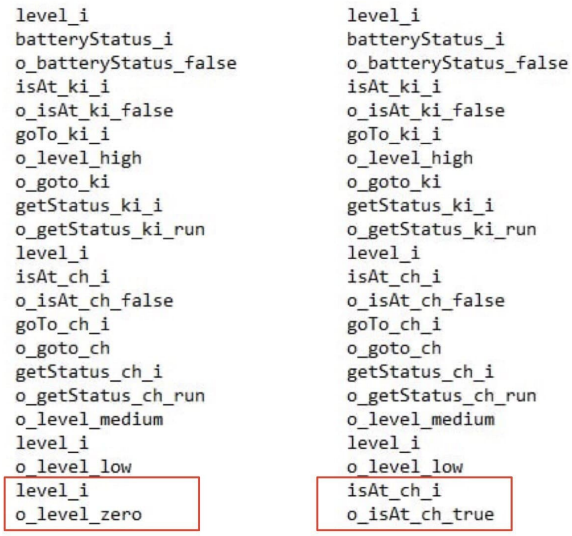


Figure 5.6: Comparison between abstract trace for partial Scenario 2 (right) and abstract trace for Scenario 3 (left).

Because of this issue, we have decided to change the *Early Warning* threshold to 50% and also the abstraction for the message that outputs the battery level. Starting from what described in Section 4.2, the new abstraction is the following:

- $100 < level < 50$: o_level_high
- $50 < level < 45$: o_level_medium_six
- $45 < level < 40$: o_level_medium_five
- $40 < level < 35$: o_level_medium_four
- $35 < level < 30$: o_level_medium_three
- $30 < level < 25$: o_level_medium_two
- $25 < level < 20$: o_level_medium_one
- $20 < level < 0$: o_level_low
- $level = 0$: o_level_zero

In this way, by adding new intermediate levels in the abstraction, it is possible to better differentiate traces from Scenario 2 and from Scenario 3. With this new abstraction we have then tested everything with both simulators.

5.2.1 Off-line Simulator

As first step, for the off-line simulator, we have learned again the machines for Scenario 2 and Scenario 3 with the new abstraction by repeating the steps in Subsection 5.1.1. For Scenario 1 was sufficient the previous model, since the battery level never goes under the *Early Warning* threshold and the output message will always be *o_level_high*. Once learned the machines, to test everything at runtime, we need to make some modifications in order to make the simulator as similar as possible to the on-line execution of the robot. Our aim is to verify at runtime to be notified of being in the case of trace with faults before getting a battery level of 20%. To obtain this, we have implemented a script that starting from the *log* file obtained when running the off-line simulator, writes a new log file that is periodically updated, simulating the communication pattern of skills and components in the runtime scenario. Once that the script is running, we

have used the Algorithm 8 to verify in which scenario we are. In Figure 5.7 is shown the result when we have a trace belonging to Scenario 3.

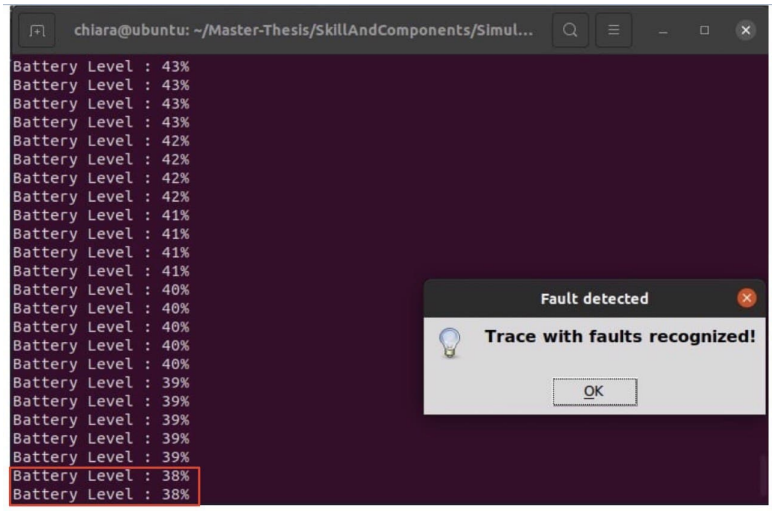


Figure 5.7: Result for a trace belonging to Scenario 3 with the off-line simulator

As we can see, the warning message is produced when the battery level is approximately 38%, which is rather earlier that the Monitor, proving that we have obtained the desired result i.e., the predictive monitoring.

5.2.2 On-line Simulator

The last step consists in trying everything on the on-line simulator, since this is the real application of interest. What we want now, is to verify that the predictive monitoring holds in a real scenario, with the real times of execution of the robot's tasks.
The procedure is the same, so we have to learn again the machines for Scenario 2 and Scenario 3. With the machines learned, it is possible to proceed with the verification of the predictive monitoring at runtime. To perform this, we need to access the *log* file during the execution, so that we can use again Algorithm 8 to classify the type of trace. In Figure 5.8 is shown the result when we have a trace belonging to Scenario 3. As we can notice the robot is trying to reach the charging station, but the battery level is not sufficient to success, the level is approximately 30-35%, so the Monitor GUI shows that the property is still not

violated (green semaphore light). However, with all these preconditions, we have the warning message notifying that we are in a scenario with faults, proving that the predictive monitoring has happened successfully.
Again we have the proof that we have implemented something that is predictive, since the robot in this example is still navigating towards the charging station, nothing apparently has been violated, but by comparing the trace of execution with what learned before (the models of the Mealy state machines), we can understand that this execution will lead to a dangerous situation even before that the dangerous situation has actually occurred.

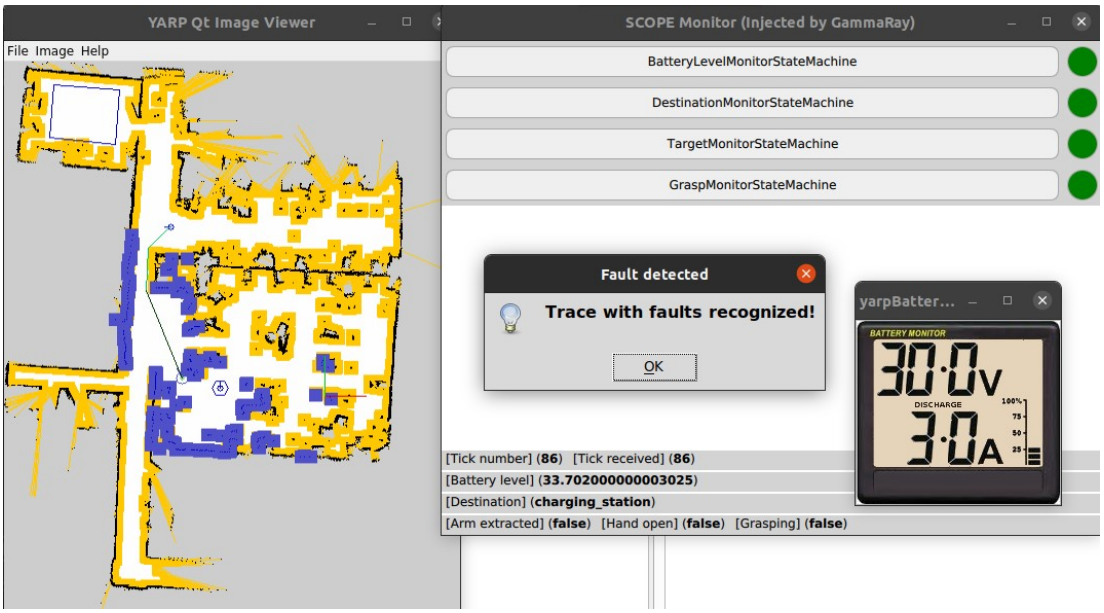


Figure 5.8: Result for a trace belonging to Scenario 3 with the on-line simulator

Conclusions

Assitive robotics is becoming more and more used due to the necessities of the last years. Since SARs operate in a domestic environment and with elder people, safety must be guaranteed. In assitive robotics we can rely on Runtime Monitoring to achieve an appropriate level of safety, but it is not sufficient. The goal of this project has been to guarantee also robot's autonomy. This has been possible thanks to the application of Automata Learning used to obtain usable description of the environment together with the implementation of the Predictive Monitoring i.e., the capability of detecting property violations, due to faulty scenarios, in advance.

The first step of this study has concerned the learning of the environment below the deliberative layer of the assistive robot R1 in a simple but realistic scenario involving navigation in a domestic environment. To achieve this we have defined the alphabet for the abstraction of the environment and then we have chosen three main scenarios (two nominal and a faulty one) building a FSM for each of them. Then, we have tested with a classifier if, starting from those models we were able to recognize if a trace of execution belonged to one of the three models, giving more attention to the discrimination of the faulty scenario i.e., a scenario with property violations.

As final step we tested this classification firstly off-line, and then we brought everything at runtime in order to verify the predictive monitoring. At runtime, we have shown that in scenarios where a safety requirement will be violated, our software allows the robot to understands that something will go wrong before the intervention of the runtime monitoring module.

Besides, the gap between the moment where the dangerous scenario is identified and the moment where the runtime monitoring intervenes, offers to actuate some recovery action to obtain the desired level of autonomy for the assistive robot.

As future work, a recovery plan can be implemented, since until now we only have a warning when a fault is detected, but to make the robot more autonomous, it can be equipped with some module to manage the situation properly, avoiding the intervention of the user. Moreover another future implementation, could be to to test more complex scenarios or to replicate the predictive monitoring but by using Runtime Verification applied in a predictive way and exploiting powerful tools for the verification.

References

[1] C. Getson and G. Nejat, "Socially Assistive Robots Helping Older Adults through the Pandemic and Life after COVID-19," *Robotics*, vol. 10, no. 3, p. 106, Sep. 2021. 1

[2] S. C. Gouvernement du Canada, "Isolement social et mortalité chez les personnes âgées au Canada," https://www150.statcan.gc.ca/n1/pub/82-003-x/2020003/article/00003-fra.htm, Jun. 2020. 1

[3] B. Isabet, M. Pino, M. Lewis, S. Benveniste, and A.-S. Rigaud, "Social Telepresence Robots: A Narrative Review of Experiments Involving Older Adults before and during the COVID-19 Pandemic," *International Journal of Environmental Research and Public Health*, vol. 18, no. 7, p. 3597, Jan. 2021. 1

[4] N. S. Jecker, "You've got a friend in me: Sociable robots for older adults in an age of global pandemics," *Ethics and Information Technology*, vol. 23, no. 1, pp. 35–43, Nov. 2021. 1

[5] ""Social Distancing" Amid a Crisis in Social Isolation and Loneliness," https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7267573/. 1

[6] P. Kellmeyer, O. Mueller, R. Feingold-Polak, and S. Levy-Tzedek, "Social robots in rehabilitation: A question of trust," *Science Robotics*, vol. 3, no. 21, p. eaat1587, Aug. 2018. 2

[7] "Intensive upper limb neurorehabilitation in chronic stroke: Outcomes from the Queen Square programme | Journal of Neurology, Neurosurgery & Psychiatry," https://jnnp.bmj.com/content/90/5/498.abstract. 2

[8] "Current Trends in Robot-Assisted Upper-Limb Stroke Rehabilitation: Promoting Patient Engagement in Therapy | SpringerLink," https://link.springer.com/article/10.1007/s40141-014-0056-z. 2

[9] L. E. Kahn, P. S. Lum, W. Z. Rymer, and D. J. Reinkensmeyer, "Robot-assisted movement training for the stroke-impaired arm: Does it matter what the robot does?" *Journal of rehabilitation research and development*, vol. 43, no. 5, pp. 619–630, Nov. 2014. 2

[10] M. J. Matarić, J. Eriksson, D. J. Feil-Seifer, and C. J. Winstein, "Socially assistive robotics for post-stroke rehabilitation," *Journal of NeuroEngineering and Rehabilitation*, vol. 4, no. 1, p. 5, Feb. 2007. 2

[11] "Smooth leader or sharp follower? playing the mirror game with a robot - IOS Press," https://content.iospress.com/articles/restorative-neurology-and-neuroscience/rnn170756. 2

[12] "Robotic gaming prototype for upper limb exercise: Effects of age and embodiment on user preferences and movement - IOS Press," https://content.iospress.com/articles/restorative-neurology-and-neuroscience/rnn170802. 2

[13] M. Matarić, A. Tapus, C. Winstein, and J. Eriksson, "Socially Assistive Robotics for Stroke and Mild TBI Rehabilitation," *Advanced Technologies in Rehabilitation*, pp. 249–262, 2009. 2

[14] D. Park, H. Kim, and C. C. Kemp, "Multimodal anomaly detection for assistive robots," *Autonomous Robots*, vol. 43, no. 3, pp. 611–629, Mar. 2019. 2

[15] M. Colledanchise, G. Cicala, D. E. Domenichelli, L. Natale, and A. Tacchella, "Formalizing the Execution Context of Behavior Trees for Runtime Verification of Deliberative Policies," *arXiv:2106.12474 [cs]*, Jun. 2021. 2

[16] O. Biggar, M. Zamani, and I. Shames, "A principled analysis of Behavior Trees and their generalisations," *arXiv:2008.11906 [cs]*, May 2021. 6

[17] M. Colledanchise, R. Parasuraman, and P. Ögren, "Learning of Behavior Trees for Autonomous Agents," *IEEE Transactions on Games*, vol. 11, no. 2, pp. 183–189, Jun. 2019. 6

[18] R. Ghzouli, T. Berger, E. B. Johnsen, S. Dragule, and A. Wąsowski, "Behavior trees in action: A study of robotics applications," in *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2020. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 196–209. 6

[19] F. Rovida, B. Grossmann, and V. Krüger, "Extended behavior trees for quick definition of flexible robotic tasks," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sep. 2017, pp. 6793–6800. 7

[20] X. Neufeld, S. Mostaghim, and S. Brand, "A Hybrid Approach to Planning and Execution in Dynamic Environments Through Hierarchical Task Networks and Behavior Trees," *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 14, no. 1, Sep. 2018. 7

[21] C. I. Sprague and P. Ögren, "Adding Neural Network Controllers to Behavior Trees without Destroying Performance Guarantees," *arXiv:1809.10283 [cs]*, Jun. 2019. 7

[22] N. Axelsson and G. Skantze, "Modelling Adaptive Presentations in Human-Robot Interaction using Behaviour Trees," in *Proceedings of the 20th Annual SIGdial Meeting on Discourse and Dialogue*. Stockholm, Sweden: Association for Computational Linguistics, Sep. 2019, pp. 345–352. 7

[23] M. Kim, M. Arduengo, N. Walker, Y. Jiang, J. W. Hart, P. Stone, and L. Sentis, "An Architecture for Person-Following using Active Target Search," *arXiv:1809.08793 [cs]*, Sep. 2018. 7

[24] M. Colledanchise, "Behavior Trees in Robotics," 2017. 7

[25] N. Hili, M. Bagherzadeh, K. Jahed, and J. Dingel, "A model-based architecture for interactive run-time monitoring," *Software and Systems Modeling*, vol. 19, no. 4, pp. 959–981, Jul. 2020. 11

[26] B. Zeigler and S. Chi, "Model-based architecture concepts for autonomous systems," in *Proceedings. 5th IEEE International Symposium on Intelligent Control 1990*, Sep. 1990, pp. 27–32 vol.1. 11

[27] "IEEE 610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology," https://standards.ieee.org/standard/610_12-1990.html. 11

[28] A. Vogelsang, S. Eder, G. Hackenberg, M. Junker, and S. Teufl, "Supporting concurrent development of requirements and architecture: A model-based approach," in *2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, Jan. 2014, pp. 587–595. 11

[29] L. C. Silva, M. Perkusich, F. M. Bublitz, H. O. Almeida, and A. Perkusich, "A model-based architecture for testing medical cyber-physical systems," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ser. SAC '14. New York, NY, USA: Association for Computing Machinery, Mar. 2014, pp. 25–30. 12

[30] D. Garlan and B. Schmerl, "Model-based adaptation for self-healing systems," in *Proceedings of the First Workshop on Self-Healing Systems*, ser. WOSS '02. New York, NY, USA: Association for Computing Machinery, Nov. 2002, pp. 27–32. 12

[31] "Model-based vehicular prognostics framework using Big Data architecture - ScienceDirect," https://www.sciencedirect.com/science/article/pii/S0166361519304439. 12

[32] A. Carbone, A. Finzi, A. Orlandini, and F. Pirri, "Model-based control architecture for attentive robots in rescue scenarios," *Autonomous Robots*, vol. 24, no. 1, pp. 87–120, Jan. 2008. 12

[33] J. Tani, "Model-based learning for mobile robot navigation from the dynamical systems perspective," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 3, pp. 421–436, Jun. 1996. 12

[34] M. Shahbaz, "Reverse engineering enhanced state models of black box software components to support integration testing," *Ph. D. thesis*, 2008. 12

[35] Q. M. Tan and A. Petrenko, "Test Generation for Specifications Modeled by Input/Output Automata," in *Testing of Communicating Systems: Proceedings of the IFIP TC6 11th International Workshop on Testing of Communicating Systems (IWTCS'98) August 31-September 2, 1998, Tomsk, Russia*,

ser. IFIP — The International Federation for Information Processing, A. Petrenko and N. Yevtushenko, Eds. Boston, MA: Springer US, 1998, pp. 83–99. 14

[36] A. Khalili and A. Tacchella, "Learning nondeterministic mealy machines," pp. 109–123, 2014. [Online]. Available: http://proceedings.mlr.press/v34/khalili14a.html 15

[37] B. Steffen, F. Howar, and M. Merten, "Introduction to Active Automata Learning from a Practical Perspective," in *Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*, ser. Lecture Notes in Computer Science, M. Bernardo and V. Issarny, Eds. Berlin, Heidelberg: Springer, 2011, pp. 256–296. 16

[38] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, Nov. 1987. 17

[39] J. Oncina and P. García, "Inferring regular languages in polynomial updated time," in *Pattern Recognition and Image Analysis*, ser. Series in Machine Perception and Artificial Intelligence. WORLD SCIENTIFIC, Jan. 1992, vol. Volume 1, pp. 49–61. 18

[40] N. Yang, K. Aslam, R. Schiffelers, L. Lensink, D. Hendriks, L. Cleophas, and A. Serebrenik, "Improving Model Inference in Industry by Combining Active and Passive Learning," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb. 2019, pp. 253–263. 23

[41] "LearnLib - An open framework for automata learning," Sep. 2017. [Online]. Available: https://learnlib.de/ 23

[42] M. Merten, F. Howar, B. Steffen, and T. Margaria, "Automata Learning with On-the-Fly Direct Hypothesis Construction," in *Leveraging Applications of Formal Methods, Verification, and Validation*, ser. Communications in Computer and Information Science, R. Hähnle, J. Knoop, T. Margaria, D. Schreiner, and B. Steffen, Eds. Berlin, Heidelberg: Springer, 2012, pp. 248–260. 24

[43] M. J. Kearns, U. V. Vazirani, and U. Vazirani, *An Introduction to Computational Learning Theory*. MIT Press, 1994. 24

[44] M. Isberner, F. Howar, and B. Steffen, "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning," in *Runtime Verification*, ser. Lecture Notes in Computer Science, B. Bonakdarpour and S. A. Smolka, Eds. Cham: Springer International Publishing, 2014, pp. 307–322. 24

[45] B. Bollig, P. Habermehl, C. Kern, and M. Leucker, "Angluin-Style Learning of NFA," in *IJCAI*, 2009. 24

[46] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, May 2009. 25

[47] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, "Introduction to Runtime Verification," in *Lectures on Runtime Verification: Introductory and Advanced Topics*, ser. Lecture Notes in Computer Science, E. Bartocci and Y. Falcone, Eds. Cham: Springer International Publishing, 2018, pp. 1–33. 25

[48] M. Hinchey and R. Sterritt, "Self-managing software," *Computer*, vol. 39, no. 2, pp. 107–109, Feb. 2006. 25

[49] K. Havelund and G. Roşu, "Synthesizing Monitors for Safety Properties," in *Tools and Algorithms for the Construction and Analysis of Systems*, G. Goos, J. Hartmanis, J. van Leeuwen, J.-P. Katoen, and P. Stevens, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, vol. 2280, pp. 342–356. 25

[50] O. Pettersson, "Execution monitoring in robotics: A survey," *Robotics and Autonomous Systems*, vol. 53, no. 2, pp. 73–88, Nov. 2005. 25

[51] J. Huang, C. Erdogan, Y. Zhang, B. Moore, Q. Luo, A. Sundaresan, and G. Rosu, "ROSRV: Runtime Verification for Robots," in *Runtime Verification*, ser. Lecture Notes in Computer Science, B. Bonakdarpour and S. A. Smolka, Eds. Cham: Springer International Publishing, 2014, pp. 247–254. 26

[52] D. Ulus and C. Belta, "Reactive Control Meets Runtime Verification: A Case Study of Navigation," in *Runtime Verification*, ser. Lecture Notes in Computer Science, B. Finkbeiner and L. Mariani, Eds. Cham: Springer International Publishing, 2019, pp. 368–374. 26

[53] B. Kempa, P. Zhang, P. H. Jones, J. Zambreno, and K. Y. Rozier, "Embedding Online Runtime Verification for Fault Disambiguation on Robonaut2," in *Formal Modeling and Analysis of Timed Systems*, ser. Lecture Notes in Computer Science, N. Bertrand and N. Jansen, Eds. Cham: Springer International Publishing, 2020, pp. 196–214. 26

[54] A. Desai, T. Dreossi, and S. A. Seshia, "Combining Model Checking and Runtime Verification for Safe Robotics," in *Runtime Verification*, ser. Lecture Notes in Computer Science, S. Lahiri and G. Reger, Eds. Cham: Springer International Publishing, 2017, pp. 172–189. 26

[55] A. Parmiggiani, L. Fiorio, A. Scalzo, A. V. Sureshbabu, M. Randazzo, M. Maggiali, U. Pattacini, H. Lehmann, V. Tikhanoff, D. Domenichelli, A. Cardellino, P. Congiu, A. Pagnin, R. Cingolani, L. Natale, and G. Metta, "The design and validation of the R1 personal humanoid," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sep. 2017, pp. 674–680. 27

[56] M. Randazzo, A. Ruzzenenti, and L. Natale, "YARP-ROS Inter-Operation in a 2D Navigation Task," *Frontiers in Robotics and AI*, vol. 5, p. 5, 2018. 32