IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received July 26, 2021, accepted September 23, 2021, date of publication October 4, 2021, date of current version October 12, 2021.
Digital Object Identifier 10.1109/ACCESS.2021.3117852

A Deployment Framework for Formally Verified Human-Robot Interactions

LIVIA LESTINGI [1], MEHRNOOSH ASKARPOUR [2], MARCELLO M. BERSANI [1], AND MATTEO ROSSI [3]
[1]Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, 20133 Milan, Italy
[2]Department of Computing and Software, McMaster University, Hamilton, ON L8S 4L8, Canada
[3]Dipartimento di Meccanica, Politecnico di Milano, 20156 Milan, Italy
Corresponding author: Livia Lestingi (livia.lestingi@polimi.it)

ABSTRACT In the future, assistive robots will spread to everyday settings and regularly interact with humans. This paper introduces a deployment approach for assistive robotic applications where human-robot interaction is the main element. The deployment infrastructure hinges on a model-to-code transformation technique and a ROS-based middleware layer and enables deployment in real life or simulation in a virtual environment. The approach fits into a model-driven framework for the formal verification of interactive scenarios. At design-time, the application analyst estimates the most likely outcome of the robotic mission through Statistical Model Checking of a Stochastic Hybrid Automata network modeling the scenario. We introduce an innovative approach to convert a specific subset of Stochastic Hybrid Automata into executable code to control the robot and respond to human actions. Deploying or simulating the application allows analysts to validate the results obtained at design time or to refine the formal model based on runs in the real or the virtual scene. The methodology's effectiveness is tested via simulation of use cases from the healthcare setting, which can significantly benefit from this kind of approach thanks to its innovative features related to human physiology and autonomous behavior.

INDEX TERMS Human-robot interaction, service robots, model-driven approach, robot deployment.

I. INTRODUCTION

In the future, robots will no longer be confined in factories, but they will spread to everyday life settings. Currently, robots are either autonomous components of manufacturing lines or support humans in collaborative tasks. Factory workers are usually thoroughly trained to work side-by-side with robots, and, throughout the interaction, they perform a predetermined sequence of actions. Service robots, instead, will operate in less controlled environments and interact with a more extensive range of people [1], with different features and demands, and likely not used to *interact* with a machine.

The robotic community has established strategic objectives for robotics development over the upcoming years [2]. For the assistive robotics field, there is a specific emphasis on automating the most time-consuming technical tasks (such as code generation) so that practitioners can focus on high-level mission design and configuration. Furthermore, a recently conducted survey on the current state of software engineering for service robotics [3] has highlighted the most pressing challenges for this domain. This paper targets the demand for a *code generation* mechanism, *deployment framework*, and *validation* environment that can cope with the *uncertainty* due to the presence of humans within the environment in which robots operate [4].

We have developed a model-driven framework to analyze human-robot interaction scenarios [5], [6]. The methodology covers scenarios that feature: a battery-powered mobile robot, one or multiple humans that need to interact with the robot, and a closed environment. The agents'—the robot and the human—behavior is formally modeled as a network of Stochastic Hybrid Automata. The formal model is put through Statistical Model Checking to verify the probability of completing the mission with success. These elements constitute the design-time phase of the analysis that is performed before the application is conclusively defined.

This paper focuses on an innovative mechanism to generate and deploy executable code for assistive robotic applications starting from verified formal models. More precisely, the main contributions introduced by this paper are:

The associate editor coordinating the review of this manuscript and approving it for publication was Yang Liu.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

1) the **formal model** enhanced with deployment-related features;
2) an approach to **convert** a subset of Stochastic Hybrid Automata into executable **code** to deploy the application in a real or a simulated environment.

The approach mentioned above envisages a *mapping* function associating each feature of the automata with an element of the deployment framework. The procedure ensures that the behavior observable with the deployed version of the scenario is comparable with the one defined at design time. Although we describe its application to the assistive robotics domain, the model-to-code mapping principle is applicable to a broader range of cyber-physical systems, as long as they can be formally modeled through the same subset of Stochastic Hybrid Automata and employ a middleware technology based on the *publish-subscribe* pattern.

The run-time phase allows for the application's deployment in a real-life setting and simulation in a virtual environment. This possibility stems from the middleware layer of the module, entirely developed using ROS, thus making it portable both to 3D simulators and real mobile robots.

The run-time analysis phase serves a double validation purpose. The deployment in a physical setting with data provided by real people that interact with the robot potentially highlights weaknesses in design-time results. The reason is that the humans' formal model is an *underapproximation* of human behavior in real life. On the other hand, in some cases, the set-up of an actual testing environment might be unfeasible at an early stage of the application's development. Therefore, simulation with a realistic physics engine is a valuable alternative to highlight potential weaknesses.

The new module of the framework is tested on experimental scenarios inspired by case studies from the healthcare setting. Experiments are carried out in a simulated environment with realistic parameter values compatible with existing mobile robots, such as the TurtleBot3 Waffle Pi.[1] Human-robot interaction occurs by having real human users interact with the simulated robot in the scene.

The paper is structured as follows: Section II presents the background for the work; Section III presents the run-time phase of the framework; Section IV introduces the formal model of the middleware; Section V introduces the deployment approach and the formal model-to-code mapping principle; Section VI reports on experimental validation results; Section VII compares our work with related ones; Section VIII concludes.

II. PRELIMINARIES

This section reports on the prerequisites for the work presented in this paper and is further subdivided into three sub-topics. Firstly, we recall the theoretical and technical foundations of the work, which encompass the chosen formalism and verification technique, plus the primary pre-existing technologies underlying the work. Secondly, we report on the high-level model-driven robotic development framework that this work aims at expanding. Finally, we explain the modeling approach and the previously developed formal model, before the modifications presented in this paper.

A. TECHNICAL BACKGROUND

Stochastic Hybrid Automata (SHA) is the modeling formalism underlying the approach, whose definition—as an extension of Hybrid Automata (HA)—is reported in the following.

Definition 1: A Hybrid Automaton $\mathcal{A}$ is a tuple $(L, W, F, I, C, E, l_{ini})$ [7], where:

1) *L is the set of locations, and $l_{ini} \in L$ is the initial location;*
2) *W is the set of real-valued variables of which clocks ($X \subset W$), dense-counter variables ($V_{\mathrm{dc}} \subset W$), and constants ($K \subset W$) are special cases;*
3) *$F : L \to (\mathbb{R}_+ \to \mathbb{R}^W)$ is the labeling function assigning a set of flow-conditions to each location, where $\mathbb{R}^W$ is the set of real-valued assignments (i.e., valuations) to variables in W;*
4) *$I : L \to \wp(\mathbb{R}^W)$ is the labeling function assigning a set of invariants to each location;*
5) *C is the set of channels, including the internal action τ;*
6) *$E \subset L \times C_{!?} \times \Gamma(W) \times \Xi(W) \times L$ is the set of edges, where $C_{!?}$ is the set of complementary labels involving channels in C, $\Gamma(W)$ is the set of guard conditions and $\Xi(W)$ is the set of updates [8].*

Unlike ordinary Timed Automata [9], HA locations can be endowed with *flow conditions* —i.e., differential equations— thus supporting generic expressions for the derivatives of *real-valued* variables [7]. Therefore, through HA it is possible to model systems with complex non-linear dynamics. Complex systems consisting of multiple entities can be modeled as a combination of HA, thus forming a *network*. Different automata in a network can synchronize with each other through *channels* [10]. Given two edges belonging to different automata of the network and labelled as $\mathtt{c}!$ (the *sender*) and $\mathtt{c}?$ (the *receiver*), where $\mathtt{c} \in C$ is a generic channel, triggering an event through channel $\mathtt{c}$ causes the two transitions to fire at the same time.

Definition 2: A Stochastic Hybrid Automaton $\mathcal{A}_s$ is a tuple $(\mathcal{A}, \mu, \mathcal{P}_{\gamma,\mathrm{c}})$, where:

1) *$\mathcal{A}$ is a Hybrid Automaton defined according to Def.1;*
2) *$\mu : L \times \mathbb{R}^W \to [0, 1]^{\mathbb{R}_+}$ is a labeling function assigning probability measures over time delays for each state of $\mathcal{A}_s$, where states are (l, ν) pairs constituted by a location $l \in L$ and a valuation $\nu \in \mathbb{R}^W$;*
3) *$\mathcal{P}_{\gamma,\mathrm{c}}(l, l') \in [0, 1]$ is the probability of switching from location l to l' when guard condition γ is satisfied and a message is triggered through channel c.*

In our model, non-deterministic choices are refined through stochastic features. As per Def.2, automata can have probability distributions over time delays $(\mu(l, \nu)(d))$ and

[1] Full documentation available at: https://www.turtlebot.com/

over transition outcomes ($\mathcal{P}_{\gamma,c}(l, l')$). A transition may fire with a *bounded* (e.g., within [2, 4] time units after entering the location) or *unbounded* time-delay (e.g., within range $[2, +\infty)$ time units after entering the location). We implement our SHA using the Uppaal tool. In Uppaal, probability distributions by default are either uniform over a bounded delay range or, in case of unbounded delay, exponential [11]. Nevertheless, it is possible to encode custom distributions (e.g., a normal one) by combining different uniformly distributed random variables [12]. In addition, transitions can be labelled by *weights* (captured by $\mathcal{P}_{\gamma,c}(l, l')$), which describe the probability of landing in a different location when the transition is taken.

SHA are eligible for Statistical Model Checking (SMC), which we run through the Uppaal tool and Uppaal SMC extension [12]. The extension of SMC to hybrid systems is enabled by ODE solvers that implement discretization methods. Therefore, it is possible to obtain reliable approximations of ODE solutions, although some experimental parameters such as the discretization step size require particular care [12]. The inputs to SMC are a stochastic system (e.g., the network of SHA) and a property (in our case, expressed through the PCTL logic) [13]. SMC applies statistical techniques to a sample set of system runs to check whether the specified property is verified or not. The experiment yields a range of values for the probability that such property holds within a certain time-bound. Unlike with traditional model-checking, the state-space is not exhaustively explored, thus mitigating the state-space explosion issue.

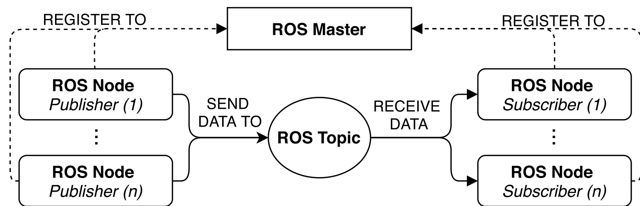


FIGURE 1. Scheme representing ROS processes interactions. All nodes register to the master. When a message is published over a topic, it is received by every node subscribed to such topic.

For the deployment approach, the chosen middleware is ROS (Robot Operating System) [14].[2] ROS is a framework for the development of robotic applications. Specifically, it provides a set of tools and libraries tailored to robot programming, developed to prioritize modularity, cross-platform compatibility, and code re-usability. Using ROS nomenclature, each process in charge of processing and exchanging data represents a *node*. All nodes communicate with each other over named buses, called *topics*. This exchange of information can occur through messages shared over topics, like in the *publish-subscribe* pattern (depicted in Fig.1), or through requests (and provision) of services (*service-client* pattern). The first type of procedure is pivotal to our approach since the robot does not ask the controller for an instruction, but the controller (i.e., the *publisher*) transmits instructions to the robot (i.e., the *subscriber*) which is constantly listening for new messages. Since ROS is built to be language-agnostic, it provides a basic Interface Definition Language (IDL) to define the messages' structure, specifically the fields that compose them and their type. The whole framework builds upon the *peer-to-peer* pattern. A process called ROS *master* serves as the conveyor unit that initially deals with nodes' registration and sets up the peer-to-peer communication. Sending messages through topics is, then, anonymous and independent from the master.

Although our approach is open to deployment on a real test bench, validation in a virtual environment is a major feature of the framework. Given the nature of the present work, the two major requirements that must be satisfied by a simulation environment are:

1) the capability to integrate with ROS;
2) the possibility to model humans and their behavior.

The following were considered as preferable features:

3) prominence of user community support;
4) openness to general-purpose scenarios.

Askarpour *et al.* report a critical comparison of existing robotic simulation tools [15]. Among the surveyed open-source tools, CoppeliaSim has been selected. CoppeliaSim (formerly V-REP)[3] is a general-purpose simulation tool that supports a plethora of robots, ranging from industrial manipulators to wheeled mobile platforms [16]. The tool provides a fully programmable model of a human, which is fundamental for developing interactive applications. This degree of flexibility is enabled by its inner architecture, which is open to different programming techniques. Developers can, thus, choose how to distribute the computational power required to control the robot and render the simulation. For example, the robot controller and the 3D simulation could run on two different machines (thus, splitting the computational load) and communicate with each other over ROS nodes.

B. MODEL-DRIVEN ANALYSIS OF ROBOTIC MISSIONS

The contribution presented in this paper is the runtime deployment infrastructure of a model-driven framework that formally verifies scenarios involving interactions between humans and robots. The approach is tailored, though not exclusively, to use cases from the healthcare setting. The motivation is that people in healthcare-related situations are often in critical conditions. For example, patients are usually in pain or discomfort, whereas employees can be mentally stressed due to harsh work habits. The markedly human-oriented nature of the tool-supported framework makes it particularly beneficial to this area.

Potential target users also come from the healthcare environment. As stated by Payne *et al.* [17], in the future, there will be professional figures entrusted with the optimization of hospital logistics to make services safer and more efficient.

[2]ROS complete documentation available at: http://wiki.ros.org

[3]Full documentation available at: https://www.coppeliarobotics.com/resources

Given the future evolution perspective described in Section I, this will likely also include analyzing possible contingencies involving interaction between humans and service robots.

The framework meets different requirements, mainly:

1) the results of the analysis ought to be robust and reliable as required by the critical setting;
2) it has to be strongly focused on human needs;
3) it has to be user-friendly and accessible to target users.

The overall workflow of the approach is represented in Fig.3. As shown in the diagram, we have identified two macro phases to analyze interaction scenarios: the design-time phase and the run-time phase. The first one is briefly summarised in the following (interested readers can refer to [5] and [6] for a thorough presentation).

The toolchain's entry point is the configuration of the scenario by the application designer, referred to as the *analyst* from this point forward. This occurs through a configuration file with the main parameters of the scenario—i.e., the floor layout, how many humans need to be served, the service they are requesting, and their physiological features.

In our scenarios, the interaction between a human and a robot conforms to pre-defined **patterns**. There is a one-to-one correspondence between humans in the scenario and interaction patterns since patterns semantically match the service that each human is requesting. A pattern determines how the agents will *behave* while that specific human is being served and what condition needs to be verified to establish that a service has been provided. Two notable examples of patterns are **Human-Follower** and **Human-Leader**. In the first case, the human needs to follow the robot until they are both sufficiently close to the destination. In the dual case, the robot follows the human and, when the human decides to start or stop, the robot follows accordingly.

As for the physiological features, the analyst can choose a **fatigue profile** for each human that determines how quickly the subject will reach full exhaustion. To determine the set of fatigue profiles, subjects are aggregated based on their *age* (young/elderly) and *health condition* (healthy/sick) [18].

The framework automatically processes the input file, it generates the formal model (the SHA network) and the PCTL property and runs the SMC experiment. By doing so, the analyst saves the effort of manually drafting the formal model, which is likely a distant skill from their technical background. The main PCTL property ψ that we are interested in verifying is: $\psi = \diamond\,\mathsf{scs}$, that is to say that the mission will *eventually* ($\diamond$) end with *success*, where variable scs formally models the mission goal achievement. As explained in Section II-A, the SMC experiment yields a range for the probability of ψ holding within a time-bound $\tau : P_{\leq\tau}(\psi) \in [\mathrm{p_{min}}, \mathrm{p_{max}}]$. The analyst assesses these values. If the results are unsatisfactory, the scenario needs to be refined and newly undergo formal verification. Possible refinements of the scenario include scheduling the provision of different services or changing the order in which humans will be served. On the other hand, if the configuration of the scenario passes the evaluation at design-time, the analyst can switch to the second phase, i.e., the *run-time* analysis which is the main contribution of this paper.

C. HRI SCENARIO FORMAL MODEL

As explained in Section II-B, the design-time phase of the framework relies on a formal model of the system under analysis, specifically a SHA network. The SHA network is made up of the automata modeling:

(a) humans (one for each interaction pattern);
(b) the mobile robot;
(c) the robot battery;
(d) the robot controller, i.e., the *orchestrator*.

The general guidelines to draft the SHA are summed up by Table 1 and are explained in the following through the running example of the robot automaton, which is fully shown in Fig.2. The mathematical notation is based on Def.2.

TABLE 1. SHA modeling guidelines.

Agent Feature	$\longrightarrow$	Automaton $\mathcal{A}_s$ Feature
State	$\longrightarrow$	Location: $l \in L$
Physical Variable	$\longrightarrow$	Real-valued Variable: $w \in W \setminus (X \cup V_{\mathrm{dc}} \cup K)$
Sensor Output	$\longrightarrow$	Dense-Counter Variable: $v \in V_{\mathrm{dc}}$
Design Parameter	$\longrightarrow$	Numerical Constant: $k \in K$
Time-Dynamics	$\longrightarrow$	Flow Condition: $f \in F(l)$
Uncontrollable Switch	$\longrightarrow$	Invariant-Edge pair: $(i \in I(l), e \in E)$ s.t. $e = (l, \perp, \gamma, \xi, l')$
Controllable Switch	$\longrightarrow$	Edge: $e \in E$ s.t. $e = (l, \mathsf{c}, \top, \xi, l')$ and $\mathsf{c} \in C \setminus \{\perp\}$
Non-deterministic Choice	$\longrightarrow$	Probability Weight: $\mathcal{P}_{\gamma,\mathsf{c}}(l, l') \neq 1$, Time-Delay Probability: $\mu(l, \nu) \neq 1$

Agents' automata have two types of locations: *ordinary* locations and *operating conditions*. Locations r_{idle}, r_{rec}, $r_{\mathrm{turn_l}}$, and $r_{\mathrm{turn_r}}$ (also in Fig.2) are *ordinary* locations. An agent stays in an ordinary location for a non-zero time, and, while in these locations, the agent does not publish sensor measurements. In Fig.2, ordinary locations respectively model the cases in which the robot is idle, recharging, or turning left or right. From this point forward, we will refer to automata modeling the robot, the battery and humans (i.e., the *agents* of the system) as generic automaton x. Operating conditions $\langle\mathsf{op}\rangle$ correspond to SHA locations, and will be hereinafter referred to with notation $x_{\langle\mathsf{op}\rangle} \in L$. While in $x_{\langle\mathsf{op}\rangle}$ locations, agent x periodically shares a new sensor reading. This modeling pattern is presented in detail in Section IV. As for the robot, there are three $x_{\langle\mathsf{op}\rangle}$ locations: r_{start}, r_{mov}, and r_{stop}. These are enclosed in as many $\langle\mathsf{op}\rangle_\mathsf{pub}_{\langle\mathsf{id}\rangle}$ instances (also presented in Section IV) and represented as dashed boxes in Fig.2 for ease of visualization. Constraints within the dashed boxes in

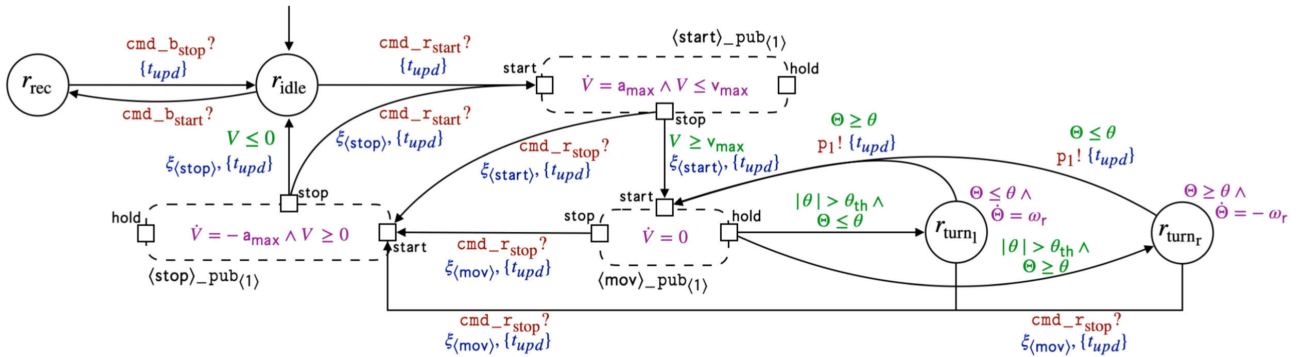


FIGURE 2. SHA modeling the robot. Invariants and flow conditions are represented in purple, guard conditions in green, channels in red, and updates in blue. Real-valued variables V and Θ model the robot's velocity and orientation. The three $\langle$op$\rangle$ locations (r_{start}, r_{mov}, and r_{stop}) are embedded into corresponding $\langle$op$\rangle$_pub$_{\langle id\rangle}$ instances (pattern $\langle$op$\rangle$_pub$_{\langle id\rangle}$ is presented in Section IV and shown in Fig.6). Locations r_{turn_l} and r_{turn_r} model the case in which the robot is turning *left* or *right* to follow the trajectory.

Fig.2 represent flow condition and invariant associated with the corresponding $x_{\langle op\rangle}$ location.

Each physical variable with non-trivial time dynamics is modeled by a real-valued variable in $W \setminus (X \cup V_{dc} \cup K)$. The robot automaton features real-valued variables V to model the robot's velocity, and Θ for the robot's orientation.

Flow conditions on each location constrain the time dynamics of real-valued variables in the model through differential equations. The robot's velocity V evolves according to a trapezoidal profile [19], and it is, thus, constrained by the three flow conditions in Eq.1. Each flow condition in Eq.1 models a phase of the velocity profile: acceleration, travel, and deceleration, where $\mathsf{a}_{max} \in K$ is a constant parameter for the robot's maximum acceleration.

$$\dot{V} = \begin{cases} \mathsf{a}_{max} & \text{if } x_{\langle op\rangle} = \mathrm{r}_{start} \\ 0 & \text{if } x_{\langle op\rangle} = \mathrm{r}_{mov} \\ -\mathsf{a}_{max} & \text{if } x_{\langle op\rangle} = \mathrm{r}_{stop} \end{cases} \tag{1}$$

Real-valued variable Θ models the orientation of the robot with respect to the x-axis, and the two locations r_{turn_l} and r_{turn_r} model the case in which the robot is rotating *left* or *right* with constant speed $\omega_r \in K$. Variable Θ varies according to flow conditions $\dot{\Theta} = \omega_r$ (in r_{turn_l}) or $\dot{\Theta} = -\omega_r$ (in r_{turn_r}) until the desired orientation is reached.

Clocks are a specific case of real-valued variables whose value grows uniformly with time ($\dot{t}_x = 1$ holds for all $t_x \in X \subset W$). A clock $t_x \in X$ can only be *reset* by transitions and this is indicated by label $\{t_x\}$ (e.g., $\{t_{upd}\}$ in Fig.2).

Both the robot and the human are equipped with sensors that monitor data required by the orchestrator (e.g., position and orientation for the robot). Sensors' readings are modeled through *dense-counter* variables, a special case of real-valued variables: specifically a variable in $V_{dc} \subset W$ for each sensor. Their value is not explicitly dependent on time ($\dot{v} = 0$ holds for any $v \in V_{dc}$) and it only varies through update instructions. Each sensor detects a new value with frequency $1/\mathsf{T}_{poll}$, where $\mathsf{T}_{poll} \in K$ is a constant parameter. Update instructions $\xi_{\langle start\rangle}$, $\xi_{\langle mov\rangle}$, and $\xi_{\langle stop\rangle} \in \Xi(W)$ (also in Fig.2) specific to the robot are made explicit in Eq.2. The Cartesian coordinates of the robot inside the building are measured and periodically updated through the dense-counter variables r_{pos_x} and r_{pos_y}.

$$\xi_{\langle start\rangle,\langle mov\rangle,\langle stop\rangle} : \begin{cases} r'_{pos_x} = r_{pos_x} + V\mathsf{T}_{poll}cos(\Theta) \\ r'_{pos_y} = r_{pos_y} + V\mathsf{T}_{poll}sin(\Theta) \end{cases} \tag{2}$$

Dense counters do not exclusively model sensor readings but, more generally, non-constant variables without an explicit time-dependency. For example, while in r_{mov} (within $\langle$mov$\rangle$_pub$_{\langle 1\rangle}$), every T_{poll} seconds the automaton updates dense-counter variable $\theta \in V_{dc}$, which represents the new set-point for orientation Θ. To make the path as smooth as possible, the robot starts turning only if the set-point θ is greater than a threshold $\theta_{th} \in K$. As in Fig.2, if $|\theta| > \theta_{th}$ holds, the robot switches from $r_{\langle mov\rangle}$ to r_{turn_l} or r_{turn_r} depending on whether $\Theta \lesseqgtr \theta$ holds. Finally, the robot switches back to r_{mov} and publishes the updated position values by sending an event through channel $\mathtt{p}_1$ (label $\mathtt{p}_1!$ in Fig.2): the details of this mechanism will be presented in Section IV.

The switch between two locations is modeled as an edge $e = (l, \mathtt{c}, \gamma, \xi, l') \in E$ that connects the starting location l to the destination l'.

A *controllable* switch occurs if a command is explicitly issued by the orchestrator by triggering an event through a *channel* $\mathtt{c} \in C$. Channels specifically related to the robot, the battery, and the human belong to subsets C_r, C_b, and C_h respectively. Examples of controllable switches are the robot starting to move or stopping, corresponding to channels $\mathtt{cmd_r}_{start} \in C_r \subset C$ and $\mathtt{cmd_r}_{stop} \in C_r \subset C$ (see Fig.2).

An *uncontrollable* switch occurs when physical variables meet specific constraints (and no command has been issued by the orchestrator): the combination of invariant $i \in I$ on location l and a guard condition γ on the outgoing edge e ensures that the transition fires only in the intersection point between i and γ. In this case, no channel is necessary for the transition and $\mathtt{c}$ is the null action. An example of uncontrollable switch is the one between r_{start} and r_{mov} that captures the end of the acceleration phase and the start of

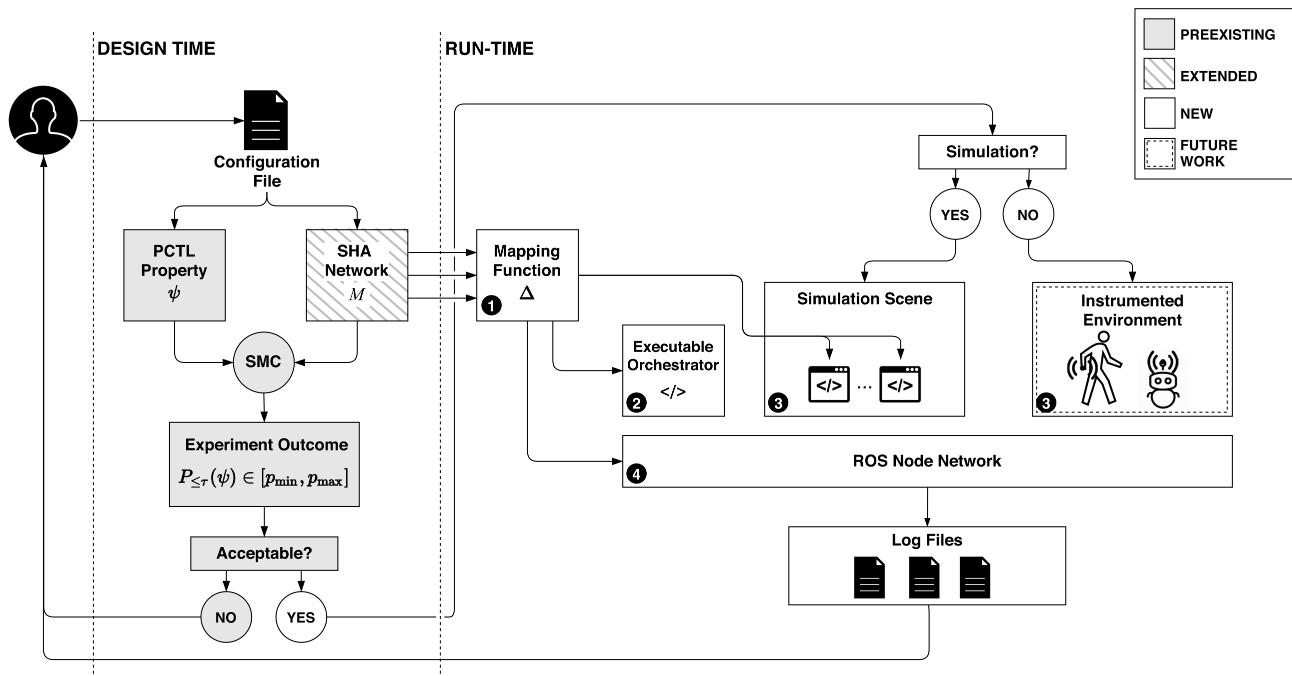


FIGURE 3. Diagram representing the framework's workflow. The two macro-phases are: design time and run-time analysis. The framework automatically generates the formal model M and the PCTL property ψ starting from a configuration file, and, then, runs the SMC experiment. If verification results are satisfactory, the analysis switches to the run-time phase. The application can either be deployed in a real setting or simulated in a virtual environment. Through mapping function Δ (presented in Section V), the automata network is translated into an *executable* form. The orchestrator communicates with agents through a ROS-based middleware layer. Deployment in a real environment is set for further investigation as future work.

the travel phase of the velocity profile. The invariant $V \leq \mathsf{v}_{\mathsf{max}}$ on r_{start} (visible within $\langle \mathsf{start} \rangle_\mathtt{pub}_{\langle 1 \rangle}$) and the guard condition $V \geq \mathsf{v}_{\mathsf{max}}$ on the edge to r_{mov} ensure that the switch occurs exactly when velocity V equals the maximum value, corresponding to constant $\mathsf{v}_{\mathsf{max}} \in K$.

III. EXTENDING THE FRAMEWORK TO RUN-TIME

As mentioned in Section I, this work targets the demand for well-engineered approaches to service robotics application development. The framework presented in this paper is an adaptation to the assistive robotics domain of the Multi-Paradigm Modeling (MPM) framework [20], which previous studies have suggested as a solution to tackle the complexity of cyber-physical systems [21]. Similarly to MPM, we propose a validation approach for robotic applications based on the comparison between the results obtained at design-time as described in Section II-B with those obtained by *deploying* the application in a realistic environment. This paper contributes with a rigorous methodology to translate the formalism from Section II-C into deployable code, which is fundamental to carry out the validation process.

In this section, we present the high-level architecture of the run-time phase and its relation to the model-driven framework recalled in Section II-B. As a further remark, the formal model presented in Section II-C has been enriched to match more accurately the deployment architecture. Therefore, it is marked as "extended" in Fig.3. The three main elements of the run-time phase (numbered accordingly in Fig.3) are:

(1) the model-to-code **mapping function** Δ responsible for converting automata into deployment units;
(2) a deployable form of the robot controller, i.e., the **executable orchestrator**;
(3) the **deployment environment** (real or virtual);
(4) a **middleware** layer that allows the orchestrator and the agents in the environment to communicate via a ROS node network.

Function Δ is presented in detail in Section V. The executable orchestrator should be a replica of the orchestrator from the formal model. This entails that each automaton component (e.g., locations, edges, and flow conditions) needs to be translated into a suitable deployment unit element. The two versions should check the system's state against the same set of policies and perform operations with the same timing. Section V describes in detail the protocol to obtain a deployable version of the automata.

The middleware layer, as mentioned in Section II-A, is based on a network of ROS publisher and subscriber nodes. Fig.4 represents how data flows from the deployment environment to the orchestrator and vice versa. The orchestrator receives sensor updates as messages published on dedicated topics and sends commands to the agents via the same mechanism. Commands cause agents to react to a specific observable situation.

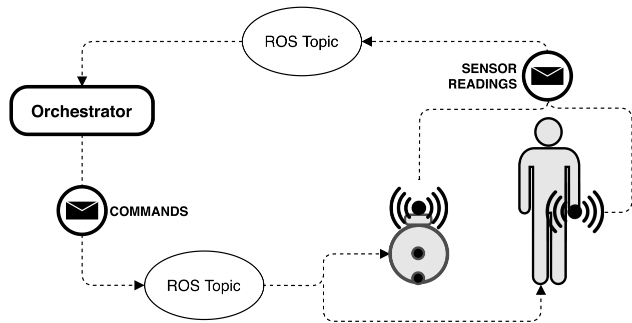


FIGURE 4. Diagram of the data flow between agents in the deployment environment and the orchestrator. Publisher nodes in the scene send the latest sensor readings to the orchestrator. The orchestrator sends new commands through a dedicated topic, and the agents in the scene receive them by subscribing to the same topic.

The usefulness of the run-time phase is twofold. In the first place, it is possible to deploy the application in a real setting so that the interaction between the human and the robot can actually occur. As Fig.3 shows, the approach is open to deployment in a real environment, as long as it is properly instrumented. Specifically, the necessary equipment includes a wheeled mobile robot compatible with ROS and a set of sensors: an Indoor Positioning System (IPS) to locate both the robot and the human inside the building, and a wearable device to measure health-related variables. The robot should also be powered by a lithium battery, typical of electronic devices, with a state of charge sensor.

As per Fig.3, it is also possible to simulate the application in a virtual environment. This operation aims to critically compare the results obtained at design-time with those obtained with the simulations. Indeed, the formal model represents a high-level abstraction of the system, especially its human-related component. While developing the formal model, our efforts focused on identifying the subpart of the whole system that represents the best tradeoff between the degree of complexity of the model and its reliability. For example, not all physical aspects, such as friction between the robot wheels and the floor, are covered as this would require an excessive degree of complexity. The residual model-to-reality gap needs to be dealt with. Increasing the model's complexity is a possible solution, but it may cause verification times to rise to a point where the framework is no longer usable in practice. The second possibility is to simulate the specific scenario in a virtual environment, like a three-dimensional simulator. In more detail, this results in a simulation relying on an executable version of the same orchestrator as in the formal model, a scenario with the same features as the one analyzed at design-time, but with a model of the agents driven by a realistic physics engine. Furthermore, the human agent in the simulation moves as a result of user input. Therefore, we can say that the robot in the simulation scene interacts with a human who provides the inputs even if the deployment environment is virtual.

The simulator's advanced physics engine raises the analysis's level of accuracy. Nevertheless, it is inexact to state that the results obtained with SMC experiments and those obtained with a single simulation are directly comparable. As a matter of fact, simulation cannot issue an exhaustive verdict like a model-checking experiment. For example, if a scenario has a 90% probability of success at the end of the design-time phase, a single successful run in the simulator would not be sufficient to say that the verification results are indisputable. On the other hand, a failed run might indicate a critical oversight of the formal model. It is possible to identify different degrees of the criticality of these flaws, depending on the level of expertise required to tackle them. Concrete examples are reported in Section VI with the presentation of experimental results.

IV. MIDDLEWARE FORMAL MODEL

As discussed in Section III, it is necessary for the verification and the deployment results to be comparable. To this end, the formal model presented in Section II-C has been enriched with features related to the software and middleware layers.

Publisher nodes are recurrent within the system since, for each sensor, a ROS node periodically shares the latest readings with the orchestrator. More specifically, the agents in the network—the robot and the humans—are responsible for *publishing* new data, whereas the orchestrator serves as a *subscriber* to such data.

ROS handles all messages received by subscribers and shared by publishers through independent queues. The rate at which subscriber queues are emptied is easier to control and anticipate than for publisher queues. As a matter of fact, the first one depends either on the time it takes to execute a single iteration of the callback function, or on the rate explicitly set for the execution of callbacks through the `ros :: spinOnce()` function. On the other hand, the time necessary to process a publisher queue depends on how quickly the message can reach the subscribers, which is not fully controllable [22].

As argued by Halder *et al.* [23], a formal model of this mechanism should include the following parameters: the publisher's publishing rate ($\mathsf{T_{pub}}$), the subscriber's *spin* rate ($\mathsf{T_{sub}}$), the time required to trasmit messages over channels ($\mathsf{T_{min}}$ and $\mathsf{T_{max}}$) and the time required to process callbacks ($\mathsf{CB_{min}}$ and $\mathsf{CB_{max}}$). Sensor readings are shared with frequency $1/\mathsf{T_{poll}}$ with $\mathsf{T_{poll}} \in K$, therefore $\mathsf{T_{pub}} = \mathsf{T_{poll}}$ holds.

In our framework, all subscriber nodes perform no other action besides data subscription, thus we can assume that $\mathsf{T_{sub}} = 0$ holds and the only delay attributable to the subscriber is the callback execution time. Given that callbacks consist of update instruction sets (see, for example, Eq.2), we can assume that $\mathsf{CB_{min}}$ and $\mathsf{CB_{max}}$ are negligible compared to the timeline of a mission (usually in the order of *minutes*). We also choose not to include the tuning of queue sizes or formally verify the *overflow* problem avoidance. Given the sizing of the system's parameters (for example, $\mathsf{T_{poll}}$ is always approximately $1s$), the message publishing rate is such that the overflow issue is not pressing enough to justify an increase in model complexity.

In the following, we present the developed SHA conforming to Def.2 and modeling:

1) ROS **publisher queues**, each referred to as `ros_pub`$_{\langle id \rangle}$;
2) agents' modeling pattern responsible for periodically **sharing** the latest **reading**, referred to as $\langle$op$\rangle$_pub$_{\langle id \rangle}$.

1) ROS PUBLISHER QUEUE MODEL

The `ros_pub`$_{\langle id \rangle}$ automaton, also shown in Fig.5, models independent publisher queues. This element of the SHA network models the previously mentioned delays to transmit messages stored in queues over ROS topics. To capture the uncertain nature of the time required to publish the message, we envisage a probability distribution that approximates this delay. Instead of having a defined interval T_{min} and T_{max}, we model the delay through variable $\lambda \in V_{dc}$, whose values are randomly generated from a Normal distribution $\mathcal{N}(\mu_\lambda, \sigma_\lambda^2)$. Previous studies have shown that such distribution is a suitable approximation for message transmission delays [24]. In this way, communication delays due to ROS latency are embedded in the model and potential impacts are accounted for by the verification process.

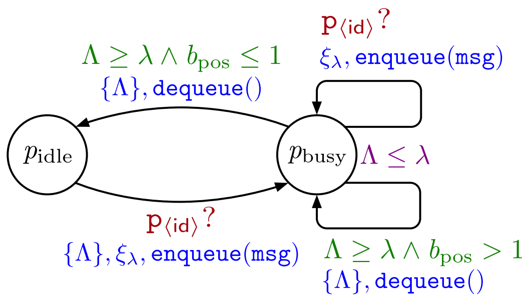


FIGURE 5. `ros_pub`$_{\langle id \rangle}$ **automaton modeling ROS publisher queue identified by parameter** $\langle$id$\rangle$**. Color-coding is the same as in Fig.2.**

As in Fig.5, a publisher queue is *empty* (location p_{idle}) until an agent (e.g., the robot) requests the publication of a message through channel $\mathtt{p}_{\langle id \rangle} \in C$. Parameter $\langle$id$\rangle$ identifies the selected queue and semantically corresponds to a ROS topic. The publication request is captured by edge label $\mathtt{p}_{\langle id \rangle}!$, either in the component presented in Section IV-2 (and shown in Fig.6) or as seen in Fig.2 (label $\mathtt{p}_1!$). As explained in Section II-A, when the edge with label $\mathtt{p}_{\langle id \rangle}!$ fires, the edge from location p_{idle} to p_{busy} with label $\mathtt{p}_{\langle id \rangle}?$ (see Fig.5) fires simultaneously.

Queues are modeled as fixed-length arrays, and variable $b_{pos} \in V_{dc}$ keeps track of the first available position's index inside the queue. As the publication request is issued (), the message is added to the buffer through the update instruction `enqueue(msg)`, a new value of λ is generated through update ξ_λ, and the automaton switches to location p_{busy}. The automaton stays in p_{busy} as long as $\Lambda \leq \lambda$ holds, where clock $\Lambda \in X$ models the message publication latency. Once time λ has elapsed, the first element of the queue is published to the orchestrator through instruction `dequeue()` (see Fig.5).

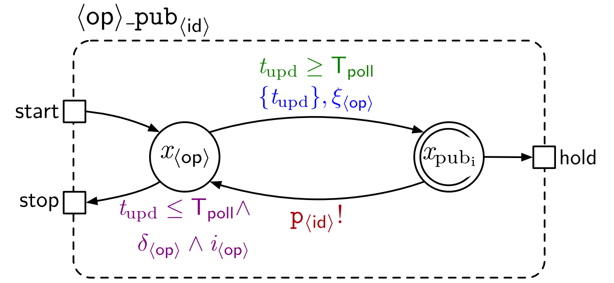


FIGURE 6. $\langle$op$\rangle$_pub$_{\langle id \rangle}$ **pattern, applied to all** $x_{\langle op \rangle}$ **locations in the network to model the message publishing mechanism on queue** $\langle$id$\rangle$**. Color-coding is the same as in Fig.2.**

The latter subsumes the *subscriber* (e.g., the orchestrator) behavior, which, as already argued, is not explicitly modeled to limit complexity.

2) SENSOR DATA PUBLISHING PATTERN

While `ros_pub`$_{\langle id \rangle}$ instances deal with message queue management, $\langle$op$\rangle$_pub$_{\langle id \rangle}$ patterns are responsible for periodically publishing the latest readings. This pattern is applied to all agents' operating conditions (see Section II-C). Location $x_{\langle op \rangle}$ in Fig.6 corresponds to a generic operating condition of agent x, and *ports* start, stop, and hold mark the transitions that enter and leave $x_{\langle op \rangle}$, whose characteristics change from case to case. Ports are not officially part of the formalism, but merely a visual expedient to represent the transitions entering and leaving the component.

While in $x_{\langle op \rangle}$, the dynamics of the system are constrained by differential equations marked by the generic symbol $\delta_{\langle op \rangle} \in F$. Symbol $i_{\langle op \rangle} \in I$ corresponds to the set of location-dependent invariants, in addition to constraint $t_{upd} \leq \mathsf{T}_{poll}$, which is common to all instances of $\langle$op$\rangle$_pub$_{\langle id \rangle}$. Clock $t_{upd} \in X$ measures time between consecutive readings. Every T_{poll} time instants, the sensor reading, modeled by a dense counter variable $v \in V_{dc}$, is updated by instruction $\xi_{\langle op \rangle} \in \Xi(W)$. It is possible for one $\langle$op$\rangle$_pub$_{\langle id \rangle}$ instance to be in charge of publishing *multiple* sensor readings (e.g., the human shares data about fatigue and position), thus, multiple dense-counter variables are simultaneously updated by $\xi_{\langle op \rangle}$.

The automaton then switches to a *committed* location x_{pub_i}. In Uppaal parlance, the automaton must leave a committed location without any delay [10] and this ensures that the latest sensor readings have precedence over the other transitions. Therefore, upon entering x_{pub_i}, if the execution of the pattern does not need to be put on hold, the automaton immediately sends an event through channel $p_{\langle id \rangle}$, triggering the queueing routine on the corresponding `ros_pub`$_{\langle id \rangle}$ instance. If, for a specific $x_{\langle op \rangle}$, execution can be put on hold, it is represented via an edge connecting location x_{pub_i} to port hold. This is useful when the operating condition has to be suspended—on an exceptional basis—to apply some reconfiguration measure. For example, concerning the robot SHA in Fig.2, this occurs when the robot has to momentarily stop moving forward to adjust its orientation Θ.

There are three instances of the $\langle op \rangle_pub_{\langle id \rangle}$ pattern in the robot automaton, one for each $\langle op \rangle$ location, also highlighted in Fig.2 and identified by a different $\langle id \rangle$ for each separate queue. As per Eq.2, depending on the values of the design parameters (i.e., if $\frac{v_{max}}{a_{max}} \geq T_{poll}$ holds), the robot might keep sharing sensors readings also while it decelerates to a full stop.

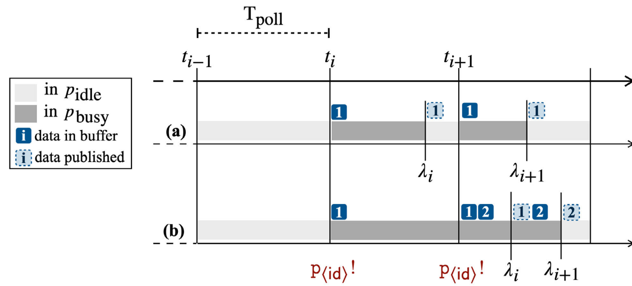


FIGURE 7. Diagram of the synchronization contingencies between an $\langle op \rangle_pub_{\langle id \rangle}$ pattern and the corresponding ros_pub$_{\langle id \rangle}$ automaton. Case (a) occurs when $\forall i, \lambda_i < T_{poll}$ holds, so the buffer never contains more than one element. If it is true that $\exists i$ s.t. $\lambda_i \geq T_{poll}$ (case (b)), the publisher will send two consecutive messages and switch back to p_{idle} only after the second one has been published.

Fig.7 displays possible contingencies (cases **(a)** and **(b)**) resulting from the combination of a $\langle op \rangle_pub_{\langle id \rangle}$ pattern with the corresponding `ros_pub`$_{\langle id \rangle}$ automaton. Case **(a)** occurs when $\forall i, \lambda_i \leq T_{poll}$ holds, therefore a message is always successfully published before the new reading and the queue never holds more than one element. If, on the other hand, $\exists i$, s.t.$\lambda_i > T_{poll}$ holds, which corresponds to case **(b)** in Fig.7, more than one position in the queue will be simultaneously occupied and two messages will be published back-to-back without switching back to p_{idle}. Case **(b)** is enabled by the two self-loops on p_{busy} (Fig.5) and guard conditions on variable b_{pos}.

V. DEPLOYMENT FRAMEWORK

The model presented in Section IV is put through Statistical Model Checking, as described in Section II-B. Once the results at design-time are deemed adequate, the scenario can be either deployed in a real environment or simulated, as per Fig.3. This section focuses on the deployment infrastructure (summarized by Fig.8) and the model-to-code mapping principle. The main elements are the executable orchestrator, the middleware layer, and the environment. The high-level role they play in the framework is explained in Section III. In the following, we explain in detail how each formal model feature is mapped to an equivalent deployable form, and how we guarantee that the deployed system and the formal model behave correspondingly.

The high-level mapping between the formal model and the deployment framework is depicted in Fig.8. Each automaton of the SHA network, excluding instances of `ros_pub`$_{\langle id \rangle}$, corresponds to an atomic entity of the deployment model. The robot, the battery, and the human, which are real entities if the application is deployed in a real environment, are

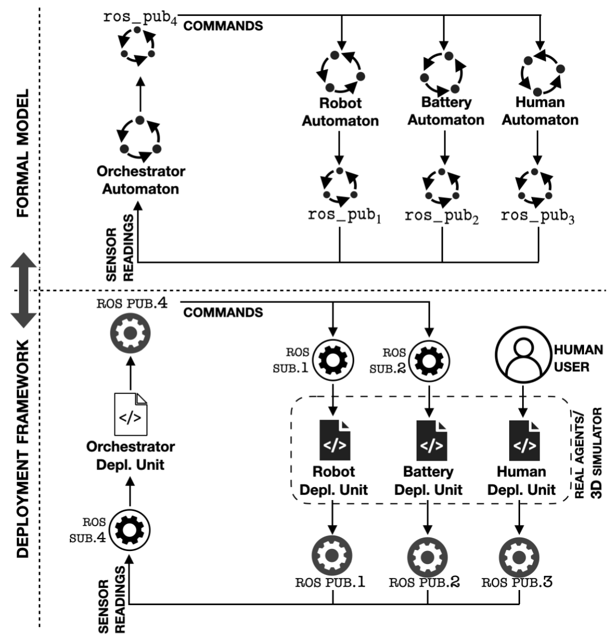


FIGURE 8. Diagram of the mapping between the formal model and the deployment infrastructure. The automata for the robot, the battery and the human are each mapped to a deployment unit. The orchestrator is ported to a standalone script, that communicates with the agents through ROS nodes. Sensor readings are shared with the orchestrator by a publisher node for each agent. The orchestrator sends its commands through a fourth publisher to all agents but the human, that is directly controlled by the human user.

actuated through scripts internal to the simulator in case of a virtual environment. The orchestrator is implemented through a standalone script in both cases.

This architecture allows the analyst to choose between deploying the application in a real or virtual environment without changing the code. All `ros_pub`$_{\langle id \rangle}$ instances correspond to a real ROS publisher node (and queue). Nodes internal to the simulator are implemented using the ROS interface provided by the CoppeliaSim API framework [16], while the nodes related to the orchestrator are implemented using the `rospy` library.[4]

Note that assistive robotics applications are our running example, and we test the effectiveness of this approach on use cases from this field, but it is not the only application domain of the model-to-code mapping principle presented in the following. A cyber-physical system is eligible for this technique if it consists of distributed agents that periodically share sensor readings through a publish-subscribe middleware technology and a centralized controller that monitors the system's state and sends instructions accordingly. In the future, we plan on deepening the analysis of the generality of the approach by testing it on other areas, such as smart homes or automated warehouses management systems.

A. MAPPING MODEL TO CODE

We exploit the stochastic features of SHA presented in Section II-A to capture the uncertain aspects of reality which

[4] `rospy` documentation available at: http://wiki.ros.org/rospy

TABLE 2. Mapping relations between features of a HA $\mathcal{A}$ and a deployment unit $\mathcal{D}$, as defined by function Δ.

Automaton Feature ($\mathbf{X}$)	$\longrightarrow$	Deployment Unit Feature (*simulated/*real/*both) ($\mathbf{\Delta(X)}$)
Location ($l \in L$)	$\longrightarrow$	Agent State ($\sigma \in \Sigma$)
Real-valued Variable ($v \in W \setminus (X \cup V_{\mathrm{dc}} \cup K)$)	$\longrightarrow$	Physical Variable ($v \in \Upsilon \setminus (X \cup \Upsilon_{\mathrm{s}} \cup K)$)
Clock ($t_x \in X \subset W$)	$\longrightarrow$	Variable Uniform to Time ($\chi \in X \subset \Upsilon$)
Dense-Counter Variable ($v \in V_{\mathrm{dc}} \subset W$)	$\longrightarrow$	Sensor Reading ($v \in \Upsilon_{\mathrm{s}} \subset \Upsilon$)
Numerical Constant ($k \in K \subset W$)	$\longrightarrow$	Numerical Constant/Design Parameter ($\kappa \in K \subset \Upsilon$)
Flow-Condition ($f \in F$)	$\longrightarrow$	Physical Law Implementation ($\phi \in \Phi$)
Invariant ($i \in I$)	$\longrightarrow$	Conditional Expression ($s \in S(\Upsilon)$)
Robot/Battery-related Channel ($\mathtt{c} \in (C_{\mathrm{r}} \cup C_{\mathrm{b}}) \subset C$)	$\longrightarrow$	ROS Topic ($\mathtt{t} \in (T_{\mathrm{r}} \cup T_{\mathrm{b}}) \subset T$)
Human-related Channel ($\mathtt{c} \in C_{\mathrm{h}} \subset C$)	$\longrightarrow$	Keyboard Input/Human Behavior ($\mathtt{t} \in T_{\mathrm{h}} \subset T$)
Edge ($e \in E$)	$\longrightarrow$	Conditional Statement ($\beta \in B$)

are relevant to our scenarios, i.e., human behavior and unpredictable network delays. Other elements of the SHA network represent either electronic devices or software modules whose behavior is fully deterministic. Therefore, although all automata in the network are defined as SHA, only those modeling humans and the $\mathtt{ros_pub}_{\langle \mathrm{id} \rangle}$ instances fully exploit the stochastic features of SHA. All other automata are designed to have a deterministic behavior, that is:

1) there are no edges that may fire with unbounded delay, thus described by an exponential distribution;
2) for all states (l, ν), with $l \in L$ and $\nu \in \mathbb{R}^W$, there is only one delay $d \in \mathbb{R}_+$ such that $\mu(l, \nu)(d) = 1$ holds, whereas, for all other $d' \in \mathbb{R}_+$ such that $d' \neq d$, $\mu(l, \nu)(d')$ equals 0. This means that, for each state, there is always only one delay which allows an uncontrollable switch (see Section II-C) to fire: thus, it is necessarily assigned probability 1;
3) all edges have probability weight 1 ($\mathcal{P}_{\gamma,\mathrm{c}}(l, l') = 1$ holds in any case).

Under these premises, we can conclude that automata representing electronic devices or software modules in the network *reduce* to pure Hybrid Automata (HA). The aspects modeled by the stochastic features do not require deployment since they are naturally present in the deployment environment. Therefore, they are not considered by the deployment approach presented in this paper. Hence, from this point forward, the discussion will focus exclusively on the automata that behave like purely HA, as they are the only ones relevant to the present work.

To guarantee that the deployed system is a sound transposition of the original network of automata, we define a mapping function Δ that maps an HA automaton $\mathcal{A}$ to $\mathcal{D}$, a generic atomic entity of the deployment framework. An atomic deployment unit $\mathcal{D} = (\Sigma, \Upsilon, \Phi, S, T, B, \sigma_{ini})$ consists of the following artefacts:

1) the set Σ of agent *states*, including the initial one σ_{ini};
2) the set Υ of *variables*, including sensor readings ($\Upsilon_s \subset \Upsilon$), constant parameters ($K \subset \Upsilon$), clocks ($X \subset \Upsilon$), and *physical* variables (real or simulated) ($\Upsilon \setminus (X \cup \Upsilon_s \cup K)$);
3) the set Φ, which contains *physical laws* in case of deployment in the real world, or the *implementation* of such laws in the case of simulation;
4) the set $S(\Upsilon)$ of *conditional expressions* on variables in Υ;
5) the set T of ROS topics over which *messages* (or *commands*) that trigger a change of state are published, with subsets T_{r}, T_{b}, and T_{h} related respectively to the robot, the battery, and the human;
6) the set $B \subset \Sigma \times T \times S(\Upsilon) \times A(\Upsilon) \times \Sigma$ of *conditional statements* governing the control flow, where $A(\Upsilon)$ is the set of assignment instructions executed on variables in Υ.

The definition of $\mathcal{D}$ is to be strictly followed while drafting the orchestrator script (both in simulated and real environments) and the agents' scripts for the simulator. As for the agents' deployment units in a real environment, since this involves actual robotic systems and humans, this definition must be intended as a high-level guideline to identify the correspondence between the formal and real systems.

Two reasons underlie this discrepancy in the interpretation. Firstly, the robotic system's code might vary significantly depending on the specific manufacturer and model and might not be fully accessible to the public. Secondly, since one of the agents is an actual person, artefacts composing the human deployment unit should not be interpreted in a software engineering-specific sense but as abstract elements constituting the human decision-making process. An explanatory example is the set S that cannot contain classic Boolean expressions for the human, but notionally corresponds to the set of questions that someone (consciously or not) ponders to make a decision [25].

Table 2 displays how function Δ maps each element of $\mathcal{A}$ to an element of $\mathcal{D}$ and is presented in detail in the following.

Each *non-committed* location in L corresponds to a state in Σ, that is to say a block of code that defines the *behavior* of an

agent under certain circumstances (e.g., the human walking or standing still).

Dense-counter variables in $V_{dc} \subset W$ have an equivalent variable in set $\Upsilon_s \subset \Upsilon$. In particular, variables representing sensor readings are periodically updated with frequency $1/\mathsf{T}_{poll}$ like dense-counter variables in the formal model.

Constants in $K \subset W$ are mapped to constant parameters in $K \subset \Upsilon$, which match design parameters of the physical equipment.

Each real-valued variable in $W \setminus (X \cup V_{dc} \cup K)$ matches a physical variable in $\Upsilon \setminus (X \cup \Upsilon_s \cup K)$, whereas flow conditions in F correspond to physical laws (or their implementations) in Φ.

Each clock in $X \subset W$ is mapped to a special case of variable in $X \subset \Upsilon$ that evolves uniformly with time. As for the orchestrator, staying in a certain location until clock $t_x \in X \subset W$ reaches threshold $k \in K \subset W$ is implemented as a $\texttt{sleep}(\kappa)$ instruction, where $\kappa \in K \subset \Upsilon$ is expressed in *seconds*. This binds the behavior of the orchestrator deployment unit to the *system* time, which can be considered an element of set $X \subset \Upsilon$. Time in the simulated environment is discrete with a time-step Δt that has a minimum value of $10ms$. Given the system's time variables sizing (e.g., T_{poll} equals $1s$, which is two orders of magnitude greater than Δt), the error caused by the discretization interval has a negligible impact on the system's behavior and, therefore, on the results of the analysis. Let us consider, for example, the model of fatigue F while the human is walking, which evolves according to equation $F(t) = 1 - e^{-\nu t}$ [26]. To prevent the human from reaching full exhaustion, the orchestrator instructs them to stop when it detects that $F = 0.7$ holds. If the reaching of this threshold is detected at time t_{stop} with a continuous-time model, in simulation it is detected at time $t_{stop} + \Delta t$ at the latest. With the most critical fatigue profile, that is with the highest rate $\nu = 0.025$, the value of F when the reaching of the threshold is detected is $F(t_{stop} + \Delta t) \approx 0.700075$, which approximately corresponds to a 0.11% error. Similar conclusions can be drawn about the other physical variables. Considering that the nature of the system does not require a sharp real-time synchronization among the various components, we can reasonably conclude that the order of magnitude of the errors does not critically threaten the model-to-code transposition soundness.

As for commands issued by the orchestrator, it is necessary to make a distinction between the ones destined to the robot or the battery and the ones destined to humans. In the first case, triggering an event through channel $\mathtt{c} \in (C_r \cup C_b) \subset C$ corresponds to publishing a message over a ROS topic $\mathtt{t} \in (T_r \cup T_b) \subset T$. The format of these messages is fixed and defined a-priori. On the other hand, the way commands are sent to the human constitutes a slight discrepancy between the formal and the deployment model. Within the HA network, the orchestrator shares its commands with the human through channels identically to what it does with the robot. The unpredictability of human behavior is embedded in the formal model in terms of stochastic features [5]. Specifically, once the command has been received, edges are also labelled by probability weights that determine whether the human will *obey* or *disobey*. Furthermore, the human can also make *autonomous* decisions, approximated by a *Bernoulli* probability distribution. At deployment-time, these stochastic approximations are lifted by having a real human either providing keyboard input (in simulation) or directly operating in the environment. In the latter case, human "inputs" are not explicit (e.g., when a human stops walking) and need to be inferred from sensors data. Therefore, human-related channels in $C_h \subset C$ match user inputs $\mathtt{t} \in T_h \subset T$.

Edges in E are translated to scripts' conditional statements and callback functions in B, representing the switch of an agent from state $\sigma \in \Sigma$ to a new state $\sigma' \in \Sigma$. As discussed in Section II-C, edges can model controllable and uncontrollable switches. Controllable switches are triggered by ROS messages, i.e., commands sent by the orchestrator. Uncontrollable switches occur in correspondence of the intersection between values that satisfy $i \in I$ and values that satisfy condition $\gamma \in \Gamma(V)$ on the outgoing edge. An edge $e = (l, \perp, \gamma, \xi, l')$ modeling an uncontrollable switch is mapped to conditional statement (i.e., *if-then* construct) $\beta = (\sigma, \perp, s, a, \sigma')$, where states σ and σ' map locations l and l', respectively. Condition γ is mapped to the conditional expression $s \in S(\Upsilon)$ guarding β. Update instructions ξ are mapped to $a \in A(\Upsilon)$, representing the (set of) assignment instruction(s) performed as soon as β is executed and s evaluates to true.

B. DEPLOYABLE CODE PATTERNS

Applying this mapping principle to recurrent modeling patterns in the HA model leads to recurrent code patterns, presented in the following. Lines in Table 3, 4, 5, and 6 are color-coded to highlight differences between deployment in a real and simulated environment.

TABLE 3. Controllable Switch pattern model (on the left) to code (on the right) transformation. Color-coding for the automaton is the same as in Fig.2, except for channel labels which, for visualization purposes, are black instead of red. The code pattern highlights mutually exclusive lines that are present if the application is simulated (in dark blue) or deployed in a real environment (in red).

Automaton Pattern	Code Pattern (*simulated/*real/*both)

$x_{\langle op \rangle}$
$\mathtt{c}_1?$ $\xi_{\langle op, op_1 \rangle}$
$\mathtt{c}_n?$ $\xi_{\langle op, op_n \rangle}$
$x_{\langle op_1 \rangle}$ ... $x_{\langle op_n \rangle}$
	for $\mathtt{t}_j \in T$ **do**
	  $\mathtt{ros.subscribe}(\mathtt{t}_j, \mathtt{t_cb}_j)$
	end
	
	function $\mathtt{t_cb_j}(\mathtt{m})$:
	  $x.l \leftarrow x_{\langle op_{j(m)} \rangle}$
	  $\Upsilon'_s \leftarrow a_{\langle op, op_{j(m)} \rangle}(\Upsilon_s)$
	  /* low-level proprietary functions */
	end

Hereinafter, while presenting code patterns, we use notation a ($\rightarrow b \in B$) to indicate how a formal model element a is mapped to a deployment unit element $b \in B$: for example, the fact that guard condition γ_j is mapped to conditional

TABLE 4. Uncontrollable Switch pattern model-to-code transformation. Color-coding is the same as in Table 3 for both columns.

Automaton Pattern	Code Pattern (*simulated/*real/*both)
$\bigwedge_{j=1}^{n} i_{\langle op_j \rangle}$; $x_{\langle op \rangle}$; γ_1, $\xi_{\langle op,op_1 \rangle}$; γ_n, $\xi_{\langle op,op_n \rangle}$; $x_{\langle op_1 \rangle}$ … $x_{\langle op_n \rangle}$	**if** $s_1 \wedge x.l = x_{\langle op \rangle}$ **then** $\Upsilon'_s \leftarrow a_{\langle op,op_1 \rangle}(\Upsilon_s)$ /* low-level proprietary functions */ $x.l \leftarrow x_{\langle op_1 \rangle}$ ... **else if** $s_n \wedge x.l = x_{\langle op \rangle}$ **then** $\Upsilon'_s \leftarrow a_{\langle op,op_n \rangle}(\Upsilon_s)$ /* low-level proprietary functions */ $x.l \leftarrow x_{\langle op_n \rangle}$ **end**

TABLE 5. Sensor reading pattern (⟨op⟩_pub⟨id⟩) model-to-code transformation. Color-coding is the same as in Table 3 for both columns.

Automaton Pattern	Code Pattern (*simulated/*real/*both)
$\bigwedge_{j=1}^{n} i_{\langle op_j \rangle} \wedge$ $t_{upd} \leq T_{poll} \wedge \delta_{\langle op \rangle}$; $x_{\langle op \rangle}$; $t_{upd} \geq T_{poll}$, $\{t_{upd}\}, \xi_{\langle op \rangle}$; $p_{\langle id \rangle}!$; x_{pub_i}	$t \leftarrow$ getSysTime() **if** $x.l = x_{\langle op \rangle} \wedge$ $\bigwedge_{j=1}^{n} s_{\langle op_j \rangle} \wedge$ $t - t_{last} \leq T_{poll}$ **then** $\Upsilon \leftarrow \phi_{\langle op \rangle}(t, K)$ /* real physical evolution */ **end** **if** $x.l = x_{\langle op \rangle} \wedge$ $t - t_{last} \geq T_{poll}$ **then** $t_{last} \leftarrow t$ $\Upsilon'_s \leftarrow a_{\langle op \rangle}(\Upsilon_s)$ /* sensor reading */ **for** $\upsilon \in \Upsilon_s$ **do** ros.publish($t_{\langle id \rangle}, \upsilon$) **end** **end**

expression s_j is expressed as γ_j ($\rightarrow s_j \in S(\Upsilon)$). We recall that, as in Section II-C, the automata that model the robot, the battery, and the humans are generically labeled as x, whereas the location capturing a generic operating condition ⟨op⟩ of agent x is labelled as $x_{\langle op \rangle}$. The pattern featuring locations labelled as o is instead a subcomponent of the orchestrator automaton and is, thus, realized by a standalone script.

As for the patterns for simulation scripts (described in Section V-B1, Section V-B2, and Section V-B3), a further remark is necessary about their apparently non-cyclical nature. The scripts implement a standard interface provided by the simulator. All the code blocks shown in Table 3, 4, and 5 belong to a function of the interface that deals with agents' *actuation* in the scene and is, by default, re-run at each time step ($\Delta t = 10ms$) throughout the whole simulation.

1) CONTROLLABLE SWITCH PATTERN

The first pattern, shown in Table 3, is the controllable switch. The resulting code pattern consists of:

TABLE 6. System monitoring pattern model-to-code transformation. The generic automaton and code patterns are shown on the left, whereas the right column features the specific instance of this pattern in our framework. Color-coding is the same as in Table 3 for both columns.

Generic Automaton/Code Pattern	Instantiated Automaton/Code Pattern
$t_{act} \leq T_{int}$; $o_{\langle op \rangle}$; $\bigwedge_{j=1}^{n} \neg\gamma_j \wedge \gamma_{proc}$, $\{t_{act}\}$; $t_{act} \geq T_{int}$, $\xi_O, \{t_{act}\}$; $o_{chk_{\langle x \rangle}}$; $t_{act} \leq T_{proc}$; $\gamma_1 \wedge \gamma_{proc}$; $\gamma_n \wedge \gamma_{proc}$; p_1 … p_n	$t_{act} \leq T_{int}$; $o_{\langle op \rangle}$; $\neg\gamma_{stop} \wedge \neg\gamma_{fail} \wedge \neg\gamma_{scs} \wedge \gamma_{proc}$, $\{t_{act}\}$; $t_{act} \geq T_{int}$, $\xi_O, \{t_{act}\}$; $o_{chk_{\langle x \rangle}}$; $t_{act} \leq T_{proc}$; $\gamma_{scs} \wedge \gamma_{proc}$; $\gamma_{stop} \wedge \gamma_{proc}$; $\gamma_{fail} \wedge \gamma_{proc}$; scs; stop; fail
while $\bigwedge_{j=1}^{n} !s_j$ **do** $o.l \leftarrow o_{\langle op \rangle}$ sleep(T_{int}) $\Upsilon_s \leftarrow a_O(\Upsilon_s)$ $o.l \leftarrow o_{chk_{\langle x \rangle}}$ sleep(T_{proc}) **end**	**while** $!s_{stop} \wedge !s_{scs} \wedge !s_{fail}$ **do** $o.l \leftarrow o_{\langle op \rangle}$ sleep(T_{int}) $\Upsilon_s \leftarrow a_O(\Upsilon_s)$ $o.l \leftarrow o_{chk_{\langle x \rangle}}$ sleep(T_{proc}) **end**

- a **for** loop so that, during initizialization, agent x subscribes to ROS topics $t_j \in T$;
- a **callback function** t_cb$_j$ (one for each subscribed topic) which is invoked every time a new message m is received through topic t_j: within the function, operating condition $x.l$ is updated, then assignment instructions $a_{\langle op,op_{j(m)} \rangle}$ are executed.

A controllable switch occurs when a change of operating condition from ⟨op⟩ to ⟨op$_j$⟩ is appropriate according to the orchestrator policies. In the formal model, this causes the related channel c_j to fire; the orchestrator triggers an event (c_j!) and an agent reacts to it (c_j?). In the deployment model, commands are shared via ROS. Therefore, the orchestrator script publishes the command through a dedicated ROS publisher node. The script internal to the simulator corresponding to the destination agent, having subscribed to ROS topic c_j ($\rightarrow t_j \in T$) at the beginning of the simulation/execution, receives such command over the same topic.

While in the corresponding status ⟨op⟩, the agent is constantly listening on topic c_j ($\rightarrow t_j \in T$). As soon as a new message is published, the script executes the callback function, generically called t_cb$_j$.

In the real script, the whole ROS message m is passed as input parameter to the callback function. The target operating condition is a function of message m and is, thus, with a slight abuse of notation, referred to in Table 3 as ⟨op$_{j(m)}$⟩.

The content of function t_cb$_j$ replicates the corresponding edge of the automaton: firstly the variable $x.l \in \Upsilon_s \subset \Upsilon$

that keeps track of agent x's current location is updated to $x_{\langle \mathsf{op}_{j(m)} \rangle}$. Subsequently, the actions performed (or simulated) by agent x in reaction to the command are captured by update instructions $\xi_{\langle \mathsf{op}, \mathsf{op}_{j(m)} \rangle}$ ($\rightarrow a_{\langle \mathsf{op}, \mathsf{op}_{j(m)} \rangle} \in A(\Upsilon)$). As per Table 3, in case of deployment in a real setting, this would correspond to the execution of proprietary functions dealing with lower-level tasks (e.g., trajectory planning).

2) UNCONTROLLABLE SWITCH PATTERN

The uncontrollable switch pattern, shown in Table 4, consists of:

- n **if-then-else** statements $\beta_j \in B$ guarded by as many conditional expressions $s_j \wedge x.l = x_{\langle \mathsf{op} \rangle}$, where $s_j \in S(\Upsilon)$ holds for all $j \in [1, n]$: when one of conditions s_j evaluates to true, assignment instructions $a_{\langle \mathsf{op}, \mathsf{op}_j \rangle}$ are executed.

Note that combining the modeling patterns that formally model the scenario in our framework leads to uncontrollable switches which are guaranteed to be *well-formed*, that is, that have the following features:

1) given the invariant i and guard γ associated with the switch:
   a) $|N_i \cap N_\gamma| > 0$ holds for all uncontrollable switches, where $N_i \subset \mathbb{R}^W$ and $N_\gamma \subset \mathbb{R}^W$ are the sets of valuations that satisfy i and γ respectively;
   b) there exists at least one real-valued variable or clock $v \in W \setminus V_{\mathrm{dc}} \cup K$ such that $\nu_{\mathrm{i,var}}(v) = \nu_{\mathrm{j,var}}(v) = \bar{v}$ holds for some $\bar{v} \in \mathbb{R}$ for all $\nu_{\mathrm{i,var}} \in N_i$ and all $\nu_{\mathrm{j,var}} \in N_j$;
2) given n uncontrollable edges outgoing from the same location and guarded by as many γ_j conditions, such conditions must be *disjoint*: $\cap_{j=1}^{n} N_{\gamma_j} = \emptyset$ holds, where N_{γ_j} is the set of valuations satisfying γ_j.

In case of an uncontrollable switch, as in Table 4, the edge from location $x_{\langle \mathsf{op} \rangle}$ to $x_{\langle \mathsf{op}_j \rangle}$ is guarded by condition γ_j and $i_{\langle \mathsf{op}_j \rangle}(\rightarrow s_j \in S(\Upsilon))$ is a member of the invariant of $x_{\langle \mathsf{op} \rangle}$. The way in which invariants $i_{\langle \mathsf{op}_j \rangle}(\rightarrow s_j \in S(\Upsilon))$ are enforced in scripts is explained in Section V-B3, here we focus only on how outgoing edges are translated into code.

As explained in Section IV, the automaton is forced to switch to $x_{\langle \mathsf{op}_j \rangle}$ when variable and clock values simultaneously satisfy both $i_{\langle \mathsf{op}_j \rangle}$ and γ_j. Note that, as previously mentioned, in our model guard conditions on uncontrollable edges are guaranteed to be disjoint. For this reason, in the corresponding code pattern reported in Table 4, it is correct that, when one of the $\gamma_j \wedge x.l = x_{\langle \mathsf{op} \rangle}$ conditions is verified, no other *if* branch is visited. Constraint $x.l = x_{\langle \mathsf{op} \rangle}$ is necessary because the location might also be updated by a callback function, as explained in Section V-B1. As soon as a γ_j ($\rightarrow s_j \in S(\Upsilon)$) condition becomes true, update $\xi_{\langle \mathsf{op}, \mathsf{op}_j \rangle}$ ($\rightarrow a_{\langle \mathsf{op}, \mathsf{op}_j \rangle} \in A(\Upsilon)$) is executed (or proprietary functions are invoked). Finally, $x.l$ is updated to $x_{\langle \mathsf{op}_j \rangle}$.

3) SENSOR READING PATTERN

The third pattern is the one presented as ⟨op⟩_pub$_{\langle \mathsf{id} \rangle}$ in Section IV, which consists of:

- an **update** of variable $\mathsf{t} \in \Upsilon$ through simulator-specific function `getSysTime()`;
- an **if-then** statement $\beta_1 \in B$ which is executed if: agent x is in operating condition $x_{\langle \mathsf{op} \rangle}$, expression $s_{\langle \mathsf{op}_j \rangle} \in S(\Upsilon)$ is true for all $j \in [1, n]$, and $\mathsf{t} - \mathsf{t}_{\mathrm{last}} \leq \mathsf{T}_{\mathrm{poll}}$ holds. If all of these conditions hold, physical variables evolve according to laws $\phi_{\langle \mathsf{op} \rangle}$;
- an **if-then** statement $\beta_2 \in B$ which is executed if: agent x is in operating condition $x_{\langle \mathsf{op} \rangle}$, and $\mathsf{t} - \mathsf{t}_{\mathrm{last}} \geq \mathsf{T}_{\mathrm{poll}}$ holds. Assignment instructions $a_{\langle \mathsf{op} \rangle}$ are subsequently executed and each sensor reading $\upsilon \in \Upsilon_s$ is published on topic $\mathtt{t}_{\langle id \rangle} \in T$ through a **for** loop.

As per Table 5, every time this block of code is reached, variable $\mathsf{t} \in X \subset \Upsilon$ storing time is updated using proprietary functions provided by the system [16]. In the automaton, the value of clock t_{upd} is compared against $\mathsf{T}_{\mathrm{poll}}$ to check whether a new sensor reading is available. As for the code, a support variable $\mathsf{t}_{\mathrm{last}} \in \Upsilon_{\mathrm{s}} \subset \Upsilon$ keeps track of the time at which the previous sensor measurement was published. It follows that the expression $\mathsf{t} - \mathsf{t}_{\mathrm{last}}$ is uniform to clock t_{upd}.

If all invariants $i_{\langle \mathsf{op}_j \rangle}(\rightarrow s_{\langle \mathsf{op}_j \rangle} \in S(\Upsilon))$ hold, the simulated physical variables are then updated according to laws $\delta_{\langle \mathsf{op} \rangle}$ ($\rightarrow \phi_{\langle \mathsf{op} \rangle} \in \Phi$). If the condition guarding the end of a sensor's refresh period (i.e., $\mathsf{t} - \mathsf{t}_{\mathrm{last}} \geq \mathsf{T}_{\mathrm{poll}}$) is satisfied, $\mathsf{t}_{\mathrm{last}}$ is updated and variables in $\Upsilon_{\mathrm{s}} \subset \Upsilon$ corresponding to sensor readings are updated as required by $\xi_{\langle \mathsf{op} \rangle}$ ($\rightarrow a_{\langle \mathsf{op} \rangle} \in A(\Upsilon)$).

Since the committed location prescribes that no time elapses before the following instruction is executed (i.e., triggering channel $\mathtt{p}_{\langle \mathsf{id} \rangle}$), the script immediately instructs a dedicated ROS node to publish the updated sensor readings on topic ⟨id⟩ through the ROS interface provided by the simulator. At this point, message publication occurs asynchronously with respect to the agent's script. As a matter of fact, ROS handles the publisher's queue independently of the script, which in the formal model is captured by a specific instance of template `ros_pub`$_{\langle \mathsf{id} \rangle}$ presented in Section IV.

4) SYSTEM MONITORING PATTERN

The final pattern in Table 6 is the subcomponent ⟨op⟩_chk$_{\langle \mathsf{x} \rangle}$ [5], which is applied to all ⟨op⟩ locations of the *orchestrator*. The purpose of this pattern is to periodically check the state of the system every $\mathsf{T}_{\mathrm{int}}$ time instants against a set of policies and, if necessary, send commands to the agents (e.g., stop the robot if it has reached the destination). Note that the pattern is also applicable to other systems with a likewise behavior (i.e., sampling-based system monitoring with custom policy enforcement). While Table 6 displays the generic pattern in the left column, in the following, to explain how the pattern works, we exploit the specific instance from our framework, which is shown on the right in Table 6.

We recall that the orchestrator controls the execution from a high abstraction level. Using as reference the abstraction levels proposed by Lutz *et al.* [27], the orchestrator in our framework operates at the *task* level, meaning that it manages when and how the robot does something, irrespective of the underlying implementation. All the lower-level details (e.g., the trajectory-planning algorithm) should be proprietary to the robot manufacturer and dependent on the specific robot model involved in the application. An implementation of these low-level algorithms has been provided to test the model-driven framework, though we do not claim it is the optimal one, since, for the reasons listed above, it is not the core of this research.

The resulting code pattern consists of:

- a **while** loop that runs as long as all expressions $s_j \in S(\Upsilon)$ that map the automaton's γ_j conditions, with $j \in [1, n]$, are false: in our specific instance, these are s_{scs}, s_{fail}, and s_{stop}. Within the loop, the script: updates the state to $o_{\langle \mathsf{op} \rangle} \in \Sigma$; pauses for time $\mathsf{T}_{\mathsf{int}} \in K$ (`sleep`($\mathsf{T}_{\mathsf{int}}$)); executes assignments $a_O \in A(\Upsilon)$; switches to $o_{chk_{\langle \mathsf{x} \rangle}} \in \Sigma$; and pauses for $\mathsf{T}_{\mathsf{proc}} \in K$ seconds (`sleep`($\mathsf{T}_{\mathsf{proc}}$)).

Upon switching to checking location $o_{\mathsf{chk}_{\langle \mathsf{x} \rangle}}$, the automaton applies the orchestrator's set of policies, referred to as ξ_{O} ($\rightarrow a_{\mathsf{O}} \in A(\Upsilon)$) [5], to the agents. After executing all the instructions in ξ_{O}, one of the following guard conditions might become true:

1) the condition that determines whether the mission has ended with *success* γ_{scs} ($\rightarrow s_{\mathsf{scs}} \in S(\Upsilon)$);
2) the condition that determines whether the mission has *failed* γ_{fail} ($\rightarrow s_{\mathsf{fail}} \in S(\Upsilon)$);
3) the condition that determines, for every controlled agent in the system, whether the current agent's action has to *stop* γ_{stop} ($\rightarrow s_{\mathsf{stop}} \in S(\Upsilon)$).

In the formal model, the time required by the orchestrator to make a decision based on the current system state is modeled by parameter $\mathsf{T}_{\mathsf{proc}} \in K \subset W$. Once time $\mathsf{T}_{\mathsf{proc}}$ has elapsed ($\gamma_{\mathsf{proc}} = t_{\mathsf{act}} \geq \mathsf{T}_{\mathsf{proc}}$ holds), if one of the outgoing edges is enabled ($\gamma_{\mathsf{scs}} \vee \gamma_{\mathsf{stop}} \vee \gamma_{\mathsf{fail}}$ holds), the subcomponent $\langle \mathsf{op} \rangle$_chk$_{\langle \mathsf{x} \rangle}$ is left, otherwise the orchestrator switches back to $o_{\langle \mathsf{op} \rangle}$. The corresponding code block, shown in Table 6, captures the cyclical succession of $o_{\langle \mathsf{op} \rangle}$ and $o_{\mathsf{chk}_{\langle \mathsf{x} \rangle}}$. The orchestrator script is put on hold for $\mathsf{T}_{\mathsf{int}}$ seconds, while the system evolves, through the programmatic instruction `sleep` (the same happens afterwards for $\mathsf{T}_{\mathsf{proc}}$ seconds in $o_{\mathsf{chk}_{\langle \mathsf{x} \rangle}}$). A set of policies equivalent to $\xi_{\mathsf{O}}(\rightarrow a_O \in A(\Upsilon))$ is enforced afterwards.

The variable $o.l \in \Upsilon_s \subset \Upsilon$ that keeps track of the location is updated to $o_{\mathsf{chk}_{\langle \mathsf{x} \rangle}}$ and a second `sleep` instruction is issued to pause the execution for $\mathsf{T}_{\mathsf{proc}}$ seconds (thus, γ_{proc} is not part of the condition for the *while* cycle). Identically to the $\langle \mathsf{op} \rangle$_chk$_{\langle \mathsf{x} \rangle}$ pattern, at this point the loop condition is re-evaluated and, if $\gamma_{\mathsf{scs}} \vee \gamma_{\mathsf{stop}} \vee \gamma_{\mathsf{fail}}$ holds, the loop ends along with the execution of this pattern.

As per Table 6, for the last pattern the cycle is explicitly defined since it is part of an ordinary script external to the simulator or the real agent, whose execution flow requires explicit programming.

VI. EXPERIMENTAL VALIDATION

As discussed in Section II-B, Statistical Model-Checking is a valuable tool to analyze cyber-physical systems' settings at an early design stage. However, it cannot provide all-inclusive results on its own [28]. The deployment module presented in this paper strengthens the model-driven framework by providing users with additional tools to test and validate interactive scenarios. These features are enabled by the fact that the deployment framework elements are a rigorous transposition of the HA network. Therefore, the approach ensures that the system will display corresponding behavior at design-time and at deployment time.

In the following, we discuss the deployment module's validation process and present relevant case studies that serve the following two purposes:

P1: provide evidence that the formally modeled and the deployed agents behave correspondingly;

P2: showcase the relevance of the deployment module to the overall assessment process.

Through simulation, analysts can test applications in physically accurate environments. Moreover, they can focus their assessment on manually induced situations theoretically unlikely to occur, but critical for the mission's outcome. As explained in Section I, the mission ends with success if the robot successfully serves all humans. Failure, instead, can occur in two ways: 1) the robot's battery gets fully discharged, which makes the robot unable to move autonomously; 2) one of the humans reaches full-exhaustion; therefore, they can no longer move nor interact with the robot. To test the effectiveness of the approach in all its possible use cases, our experiments have focused on *charge-critical* or *fatigue-critical* configurations that are fitting sources of stress to the system.

EXPERIMENTAL SETTING

Though the approach is general, the performed experiments are based on healthcare-related settings. The chosen experimental setting is represented in Fig.9. The scenario features a T-shaped hospital corridor with doors leading to offices and two cupboards with medical equipment. Specifically, the main corridor of the floor layout has an area of 22m × 4m, whereas the shorter aisle leading to the cupboards is 14m × 2m. We assume that a mobile platform is deployed in this environment to assist patients and employees. In this specific example, two humans are requesting the robot's assistance. The first person needs to fetch an item whose location is unknown to them but known to the robot. Therefore, the robot has to *lead* the human to their destination (**DEST 1** in Fig.9), and the suitable interaction pattern is Human-Follower. The second human is a doctor who needs the robot to carry some tools. The doctor has to *lead* the robot

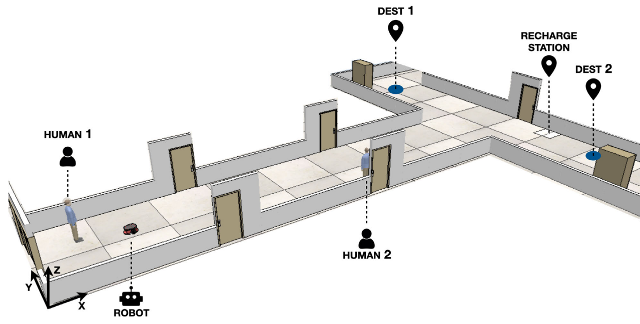


FIGURE 9. Experimental setting used for both experiments. The picture captures the layout from the simulator, which is identically replicated in the formal model. The two humans and the robot are represented in their starting positions. The picture also highlights the two destination points and the location of the recharge station.

TABLE 7. Summary of experimental parameters set. Parameters that make the configuration critical are marked in red, whereas the ones the decrease the degree of criticality are in green.

	Exp.1a	Exp.1b	Exp.2
v_{max}	$100cm/s$	$100cm/s$	$100cm/s$
a_{max}	$50cm/s^2$	$50cm/s^2$	$50cm/s^2$
C_{start}	30%	90%	50%
T_{poll}	$1s$	$1s$	$1s$
μ_λ	$0.5s$	$0.5s$	$0.5s$
σ_λ	$0.1s$	$0.1s$	$0.1s$
Pattern 1	Human-Follower	Human-Follower	Human-Follower
v_1	$100cm/s$	$100cm/s$	$80cm/s$
Ftg.Prof. 1	Young-Healthy	Young-Healthy	Elderly-Healthy
MET_1	$9210.34s$	$9210.34s$	$575.65s$
dest_1	$(2300.0, 1150.0)$	$(2300.0, 1150.0)$	$(2300.0, 1150.0)$
Pattern 2	Human-Leader	Human-Leader	Human-Leader
v_2	$100cm/s$	$100cm/s$	$80cm/s$
Ftg.Prof. 2	Young-Healthy	Young-Healthy	Young-Sick
MET_2	$9210.34s$	$9210.34s$	$460.51s$
dest_2	$(2300.0, 250.0)$	$(2300.0, 250.0)$	$(2300.0, 250.0)$

to the tools' location (point **DEST 2** in Fig.9), conforming to the Human-Leader interaction pattern. Both patterns are described in more detail in Section II-B.

Results obtained by applying our model-driven approach to this scenario significantly vary depending on parameter values. As already discussed, the two critical factors are the robot battery charge (C_{start}) and the *fatigue profiles* of the two humans ($\mathsf{p}_{f1}, \mathsf{p}_{f2}$), presented in Section II-B. The fatigue profile determines the Maximum Endurable Time (MET) value—i.e., how long a subject can walk non-stop before reaching full exhaustion. The MET is an indicator of how critical a fatigue profile is. When the human is walking, fatigue grows exponentially ($F(t) = 1 - e^{-\nu t}$) [26]. Since full exhaustion occurs when F equals $1-\epsilon$, it follows that the MET value is $\frac{-\ln(\epsilon)}{\nu}$, where parameter ϵ represents an uncertainty factor due to sensors' inaccuracy and human physiology variability. If we assume that ϵ equals 0.01, full exhaustion is reached when F equals 0.99. The Young/Healthy fatigue profile corresponds to a MET of $9210.34s$ [29]. With an Elderly/Healthy profile, the MET equals $575.65s$, whereas with the Young/Sick profile the MET equals $460.51s$ (see Table 7).

We present two significant use cases, hereinafter referred to as Experiment 1 and Experiment 2. Experiment 1 presents a *charge-critical* configuration and consists of two iterations of the approach (labelled as Experiment 1a and 1b). Experiment 2 starts with a *fatigue-critical* configuration. Table 7 summarizes the parameters for each experimental configuration. As explained in Section V, the listed design parameters and the low-level algorithms are specified for a generic mobile platform fit for testing purposes. However, they would need to be tuned (or come pre-packaged with the simulation model) based on the specific robot model to be deployed in a real environment.

EXPERIMENTAL VALIDATION PROCESS

The validation process we followed to assess if the deployment framework is an accurate translation of the formal model (i.e., goal **P1**) consists of the following steps (also conforming to the framework presented in Section II-B):

1) automatically generate the **formal model** configured according to parameters in Table 7;
2) run the **SMC experiment** to estimate the probability of success by verifying property ψ presented in Section II-B. The upper part of Table 8 displays the chosen time-bound values and performance data (duration of the experiment and number of states explored);
3) **deploy** the application in the simulated environment, with a real human user giving instructions to the human avatar in the scene;
4) collect the simulations' log files and compute metrics to **compare the system's behavior** at design-time and at runtime and draw conclusions about the soundness of the deployment approach. The bottom part of Table 8 displays the resulting values of such metrics for each experiment, which will be analyzed in more detail later in this section.

Further remarks are necessary about the data shown in Table 8 and how we have calculated it. Firstly we have empirically analyzed single traces for all experiments to estimate the average duration of the mission and choose the time-bound τ accordingly. Hence, we reasonably rule out the possibility of obtaining a low probability of success due to an insufficient time-bound.

In the second place, it is necessary to explain why the number of runs differs between design-time and runtime. Before running the SMC experiment, Uppaal automatically computes the number of traces (i.e., runs) necessary to reach the required level of confidence 0.95. Since the set of statistical

TABLE 8. Metrics to evaluate the correspondence between the formal model (FM) and the simulated system behavior at deployment time (SIM). The upper part of the table reports performance data about the Statistical Model-Checking experiments and the number of runs for each experiment. In the bottom part, the above-mentioned metrics are listed. These metrics concern the average mission duration in simulation and the time-bound (τ) for the verification, the resulting probability of success within the specified time-bound, and the expected values for the most critical physical variables: the two humans' fatigue (F_{h1} and F_{h2}, respectively) and robot battery charge (C).

	Experiment 1a		Experiment 1b		Experiment 2	
	FM	SIM	FM	SIM	FM	SIM
Runs	389	102	389	109	389	100
States	260961	–	520962	–	651867	–
Verification Time [min]	1.05	–	2.38	–	2.63	–
Avg. Completion Time [s]	–	65.32	–	66.10	–	70.95
Time-bound (τ) [s]	70	–	70	–	80	–
$\mathcal{P}_{\leq\tau}(\diamond\, \mathsf{scs})$ [%]	≥ 90.186	96.1	≥ 90.186	94.5	≥ 90.186	100
$\mathrm{E}_{\leq\tau}[\max(F_{h1})]$ [%]	1.37 ± 0.3	1.36	1.39 ± 0.2	1.43	25.54 ± 4.9	25.57
$\mathrm{E}_{\leq\tau}[\max(F_{h2})]$ [%]	0.92 ± 0.3	0.96	0.93 ± 0.4	1.06	23.83 ± 3.7	21.43
$\mathrm{E}_{\leq\tau}[\min(C)]$ [%]	24.82	24.82	89.36	89.26	54.25	54.10

parameters (including the confidence) of Uppaal is the same for all experiments, this computation yields the same result, that is, 389 runs of the system. The deployment phase is not subject to such statistical requirements. Therefore, we choose to produce a number of simulation runs that is meaningful for practical reasons. If we assume that about 40 people might be served by the robot every day for 5 workdays a week and 2 humans are served in each run, simulating the mission about 100 times realistically corresponds to testing the deployment framework over the span of a week.

To fulfill goal **P2**, the validation process also envisages the following step:

5) while simulating the application, **force** situations that can lead to **mission failure** which are not covered by the formal model and, thus, not accounted for by verification results. Some examples and possible countermeasures for each experiment are presented in the corresponding subsections.

The deployment framework used for the experiments is available at [30]. Recording of sample simulation runs is also provided for interested readers [31]. The formal model is created with Uppaal v.4.1.24 [12].[5] The virtual environment is created using CoppeliaSim v.3.6.2. The deployment units' scripts are implemented in Python v.3.6.9 and the LUA scripting language.[6] Finally, the middleware layer is built using the ROS Melodic distribution v.1.14.7.

A. EXPERIMENT 1: CHARGE-CRITICAL CONFIGURATION

Experiment 1a involves two young healthy humans and a robot with a low charge value ($\mathsf{C}_{\mathsf{start}} = 30\%$). With this configuration, when the mission starts, the robot approximately has 350*s* of battery life remaining. If both humans start walking as soon as it is their turn to be served and they perform flawlessly, 350*s* are sufficient to complete the mission. As Table 8 shows, the average mission completion time observed in the simulations is 65.32*s*. This result is compatible with the time-bound empirically chosen for the formal verification (70*s*), which leads to a success probability interval—estimated with SMC—of [0.90186, 1]. This success rate is confirmed while deploying the application since processing the collected log files reveals that in 96.1% of the simulations the mission was indeed successfully completed within 70*s*. Note that, with a slight abuse of notation, both in Table 8 and below the success rate of the deployed application is still indicated as $\mathcal{P}_{\leq\tau}(\diamond\, \mathsf{scs})$, as in the formal model; however, it is not calculated through SMC, but as the ratio between how many runs feature the mission successfully ending within τ and the total number of runs, as per Eq.3.

$$\mathcal{P}_{\leq\tau}(\diamond\, \mathsf{scs}) = \frac{\#\text{ successful runs within } \tau}{\#\text{ runs}} \tag{3}$$

Besides the success rate, it is crucial to also verify that physical variables are accurately simulated. To this end, we have calculated through Uppaal the expected value for the maximum value of fatigue reached by the two humans and the residual battery charge at the end of the mission (thus, the minimum value since, in these scenarios, the robot never recharges). The same three metrics are extracted from the simulation logs. By comparing these indicators, we can conclude that both fatigue and charge evolutions in time are consistent with the results obtained with the formal model. In both phases, the two humans reach a negligible value of fatigue ($\sim$1%) as they both conform to the "best" fatigue profile and only walk for about 30*s* each, whereas the residual level of charge is approximately 24%.

After a thorough evaluation of the system's behavior in "regular" circumstances, the analyst can forcedly induce the situation in which the second human does not start as soon as it is their turn, and the whole execution of the mission is delayed (see Fig.10). The cause of the delay in a real setting could be the doctor being unexpectedly held up or failing to

[5]SMC experiments are performed on a Linux machine with 128 cores, 515GB of RAM, and Debian Linux version 10.

[6]Simulations are executed on a Ubuntu 18.04 virtual machine with 2 cores and 4GB of RAM.

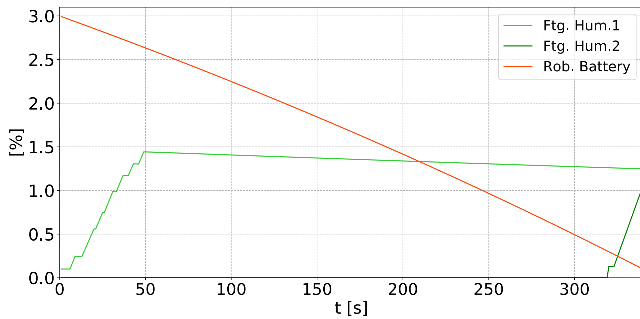


FIGURE 10. Plot of the unsuccessful simulation run from Experiment 1a. Green lines correspond to human fatigue ([0 – 100]), while the orange line corresponds to the battery charge ([0 – 10]). Human 2 starts walking at $t \simeq 320s$, about 270s after the first human has been served, causing mission failure ($b_{ch} \simeq 0\%$).

react immediately, which is a common human mistake [32]. The consequently extended duration of the mission leads to the robot being fully discharged before its completion,[7] which is one of the two possible causes of *failure*.

The analyst is now able to assess the unsuccessful simulation run. They might decide that this manifestation of human free will is plausible in real life and critical enough to motivate a scenario refinement and an additional iteration of the design-time analysis. Since humans cannot be programmatically instructed to perform actions in a machine-like manner (neither in the formal model nor in real life), the most sensible refinement action would be selecting a higher value of $\mathsf{C}_{\mathsf{start}}$. If multiple mobile platforms are available, this can be realized by choosing a different robot from the fleet, otherwise by recharging the robot before starting the mission.

The new configuration is shown in Table 7 as Experiment 1b, with the updated value of $\mathsf{C}_{\mathsf{start}} = 90\%$. The second iteration of SMC experiments, with $\tau = 70s$, yields the same success rate $P_{\leq\tau}(\diamond \text{ scs}) \in [0.90186, 1]$ (see Table 8). Rerunning the batch of simulations leads to similar results as for Experiment 1a: the average completion time is approximately 66s and the success rate within 70s is 94.5%. Note that metrics concerning fatigue are almost unchanged with respect to Experiment 1a and are still comparable to the ones estimated with the formal model. As expected, the residual battery charge is higher than for Experiment 1a (about 89%) coherently with the different value of $\mathsf{C}_{\mathsf{start}}$.

B. EXPERIMENT 2: FATIGUE-CRITICAL CONFIGURATION

The setup for Experiment 2 is more critical in terms of human endurance to physical strain. Specifically, criticality arises from the different fatigue profiles Elderly/Healthy and Young/Sick, as in Table 7. In practice, the second subject could be an employee with an undiagnosed condition affecting their respiratory capacity and physical endurance.

[7] Although the robot can recharge itself both in the formal model and during simulation, it is effectively busy serving a *leader* human when it reaches the recharge threshold ($C = 20\%$). Under these circumstances, the robot cannot halt service delivery even if the human is delaying their action.

For this experiment, we have calculated the same metrics described in Section VI-A, whose resulting values are also shown in Table 8. The chosen value for τ in this case is 80s, which is slightly higher than for Experiment 1 since humans walk at a slower pace due to the critical health conditions. The SMC experiment yields a success probability range of [0.90186, 1]. Therefore, as in Experiment 1, the analyst would have sufficient evidence to consider the mission fit for deployment. Also in this case, the results obtained at design-time are confirmed by deploying the application. The average mission completion time is 70.95s which, as expected, is slightly higher than for Experiment 1 and compliant with the chosen time-bound. More specifically, all the performed simulations were successfully completed within $\tau = 80s$ ($\mathcal{P}_{\leq\tau}(\diamond \mathsf{scs}) = 100\%$). The expected values for the fatigue peaks are significantly higher than in the previous experiment ($\sim 20\% \gg \sim 1\%$) due to the different fatigue profiles, but they are still compliant with the values calculated through Uppaal at design-time. The same stands for the robot battery charge. Therefore, also in this second experiment, the deployed system's behavior shows evidence of an accurate transposition of the formal model.

Nevertheless, suppose now that the second subject—with a more critical fatigue profile—follows an erratic trajectory and not the shortest one. This situation can be simulated in the virtual environment. Fig.11 shows a specific simulation run in which the second human exhibits this behavior and keeps walking until full exhaustion. The first human reaches their destination in approximately 50s (as in Fig.11, they stop walking and start resting). The second human starts walking at around $t = 20s$ and keeps moving for about 450s. As in Table 7, a subject with the Young/Sick fatigue profile can walk non-stop at most for approximately 7.5min ($\mathsf{MET}_2 = 460.51s$). Therefore, the mission fails after about 500s because the second human reaches the maximum endurable value of fatigue.

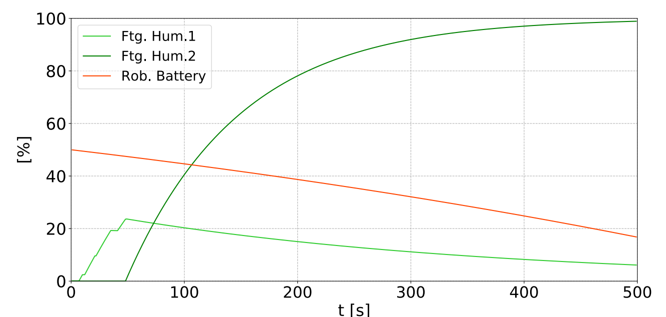


FIGURE 11. Plot of the unsuccessful simulation run for Experiment 2. Green lines correspond to human fatigue ([0 – 100]), while the orange line corresponds to the battery charge ([0 – 100]). The plot shows that the first human is successfully served in about 50s as expected, whereas the second one keeps moving until they reach full exhaustion leading to mission failure ($F_{h2} \simeq 100\%$).

This specific manifestation of human autonomy is not covered by the formal model in its current development stage. It follows that SMC experiments do not account for

this possibility and yield a very high value for the chances of success. Indeed, as explained in Section V, the formal model accounts for human autonomy only to some extent; in particular, it covers the possibility of a human *disobeying* a command or freely deciding *when* to start or stop. On the other hand, it currently does not allow for the possibility that the human freely strays from the planned trajectory, as in the simulation in Fig.11.

C. DISCUSSION

Firstly, the experiments in Section VI-A and Section VI-B provide evidence that the formally modeled and the virtually simulated agents behave correspondingly. Comparing the average completion times and success rates allows us to conclude that the deployed robot controller issues the same commands as the corresponding automaton and with the same timing, leading to mission success in a comparable amount of time. As for human fatigue and battery charge metrics, it is not surprising that fatigue is—apparently—less accurately simulated than charge: the average formal model-to-deployment error for fatigue is 5.35% while it is only 0.13% for the battery charge. This discrepancy is due to the higher degree of variability of fatigue ascribable to the unpredictability of human behavior. On the other hand, battery charge evolution in time is entirely deterministic, thus leading to a minor error.

In this regard, we highlight possible limitations to our comparison between the formal model and the deployed system. Firstly, experiments are entirely carried out in a simulated environment: running the robotic application in a more dynamic setting such as a real hospital corridor may lead to unexpected contingencies that alter the presented metrics. Nevertheless, as previously discussed, simulation is sufficient to assess the accuracy of the model-to-code translation principle, which is the core of this work. Furthermore, the fatigue-related metrics involved in the discussion of the experimental results may also be refined. In fact, during simulation, we exploit benchmark data to simulate human fatigue without considering the variability due to subjects' characteristics that would provide a more comprehensive picture of the human fatigue phenomenon. In the future, we plan to include the fatigue's volatility in the simulated environment to run a more in-depth analysis.

By assessing these experimental results it is possible to identify two types of model-to-reality discrepancies that can be detected through deployment. In the first case, as in Experiment 1, even if the deployment highlights a failure due to a gap in the formal model, it is still feasible to counteract it by tuning the parameters of the scenario, e.g., the initial battery charge C_{start}. In the second case, of which Experiment 2 is an example, the failure caused by the gap in the formal model can only be tackled at design-time by refining the formal model itself. The fundamental difference is that the first case can be directly handled by the analyst (Type I refinement), whereas the second case requires expertise in formal modeling that they are unlikely to possess (Type II refinement). The next step in the development of the approach is, therefore, the creation of an automated procedure that processes data from scenarios deployed in real-life settings and refines the SHA network. By doing so, it will become possible to identify and counteract Type II failures with multiple iterations of the design-time analysis.

VII. RELATED WORK

The work presented in this paper mainly revolves around a code generation technique that translates a Stochastic Hybrid Automata network into a robotic deployment framework. There are several works in literature with a comparable goal, though often targeting a different formalism or a different phase of the software development process. In some cases, testing rather than deployment is the main purpose, as TA can be exploited to automatically generate offline and online test-cases (e.g., for ROS packages [33]). There are also previous attempts at porting a formal model to a simulation environment. One such example is the TestIt framework, that generates a simulator-agnostic simulation environment for multi-agent robotic applications starting from Timed Automata [34]. Other works focus on code analysis and explore the possibility to highlight potential flaws in a ROS-based infrastructure through model-checking [23]. Another notable example is the work by Wang *et al.* [35], presenting a model-driven framework to convert a TA network into C++ code, which is tested on a robot grasping task.

This brief survey suggests that developing code generation techniques starting from formal models is valuable to the robotic field. The approach allows for testing and deployment of robotic applications whose properties (for example, concerning safety or efficiency) have been formally verified. On the other hand, the assistive robotics domain, to which this work is tailored, calls for a sound mathematical formulation of human physiological and behavioral properties to be involved in the formal verification process. More complex time-dynamics mean that Timed Automata—including the non-deterministic or probabilistic extensions—or temporal logic-based notations [36] no longer suffice as they require a *hybrid* formalism.

Pre-existing works that introduce translation principles of Hybrid Automata into executable code do not suit robotic applications. In some cases, modular architectures with parallel components are the target [37], whereas interactive robotic applications require a hierarchical deployment structure with an intermediary middleware layer. As previously noted, some works target testing and code coverage rather than deployment [38], or model-to-model transformations, such as HA to SLSF diagrams [39]. To the best of the authors' knowledge, the work presented in this paper is the first introducing a mapping principle between a HA network and a software architecture compatible with a ROS-based robotic system.

The research line on verification and simulation techniques for analyzing human-robot interaction is also worthy of discussion. Simulation can be used to estimate the final level of satisfaction of the human *customer* after short-term human-robot interactions [40]. Some works focus on the planning

phase of the robotic mission, for example analyzing alternative workflows for the task driven by *human input* [41]. Quintas *et al.* [42] investigate how an agent's performance is affected by interaction workflows in its decision-making process. The framework processes a high-level description of the scenario and the human that will interact with the robot to generate a mission plan *graph*. The inclusion of a human operator model is also of paramount importance in the Virtual Commissioning (VC) of collaborative manufacturing tasks, whose purpose is to test in advance the reaction of the system to malfunctions [43]. Webster *et al.* [44] also argue that different verification and validation (V&V) techniques are not fully exhaustive when they are used alone, but should be combined into a *corroborative* approach to considerably increase their effectiveness.

To conclude, an approach based on formal methods, such as the one presented in this paper, can significantly benefit the software development lifecycle in terms of dependability. The vast majority of industry professionals who have employed formal verification techniques report a quality boost for the final product [45], and demand for this kind of approach is rapidly growing [46]. This kind of approach can prove especially beneficial to cyber-physical systems, where robots interact with the environment and humans [4]. Therefore, software running on robots in charge of decision-making must result from a trustworthy verification process. The deployment approach presented in this paper paves the way for a dependable code generation technique retaining the features verified while designing the robotic application.

VIII. CONCLUSION AND FUTURE WORK

We have presented a model-driven deployment approach to generate deployable code from a specific subset of SHA. If applied to the assistive robotics domain, the methodology allows roboticists to deploy formally-verified interactive applications or simulate them in a realistic virtual environment. The model-to-code conversion principle ensures that the deployed system behaves correspondingly to the original formal model, thus guaranteeing the retention of properties verified at design time. We provide empirical evidence of code conformity through experimental use cases from the healthcare setting. Simulation can also prove valuable to induce problematic contingencies which are not covered by the formal model and might call for a refinement of the robotic mission.

The approach lends itself to several future development directions. First and foremost, the framework ought to be validated in a real environment, as mentioned in Section III. Furthermore, the framework will be further expanded with a module in charge of automatically refining the SHA network, specifically the human behavior model, based on logs from real-life runs. We also plan to improve the user-friendliness of the approach by developing a Domain-Specific Language (DSL) that allows users to configure scenarios in more detail.

REFERENCES

[1] T. Gwon, H. Park, D. Seo, S. Lee, D. Kim, and S. Jeon, "A study on safety evaluation criteria of the personal carrier robot based on ISO 13482," in *Proc. 3rd IEEE Int. Conf. Robotic Comput. (IRC)*, Feb. 2019, pp. 477–482.

[2] SPARC, "Robotics 2020 multi-annual roadmap for robotics in Europe," SPARC Robot., Brussels, Belgium, Tech. Rep. 1, 2016.

[3] S. García, D. Strüber, D. Brugali, T. Berger, and P. Pelliccione, "Robotics software engineering: A perspective from the service robotics domain," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Nov. 2020, pp. 593–604.

[4] S. García, D. Strüber, D. Brugali, A. Di Fava, P. Schillinger, P. Pelliccione, and T. Berger, "Variability modeling of service robots: Experiences and challenges," in *Proc. 13th Int. Workshop Variability Modeling Softw.-Intensive Syst.*, Feb. 2019, pp. 1–6.

[5] L. Lestingi, M. Askarpour, M. M. Bersani, and M. Rossi, "Formal verification of human-robot interaction in healthcare scenarios," in *Proc. SEFM*. Amsterdam, The Netherlands: Springer, 2020, pp. 303–324.

[6] L. Lestingi, M. Askarpour, M. M. Bersani, and M. Rossi, "A model-driven approach for the formal analysis of human-robot interaction scenarios," in *Proc. IEEE Int. Conf. Syst., Man, Cybern. (SMC)*, Oct. 2020, pp. 1907–1914.

[7] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theor. Comput. Sci.*, vol. 138, no. 1, pp. 3–34, 1995.

[8] S. Feo-Arenis, M. Vujinović, and B. Westphal, "On implementable timed automata," in *Proc. Int. Conf. Formal Techn. Distrib. Objects, Compon., Syst.* Valletta, Malta: Springer, 2020, pp. 78–95.

[9] R. Alur and D. L. Dill, "A theory of timed automata," *Theory Comput. Scince*, vol. 126, no. 2, pp. 183–235, 1994.

[10] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *Formal Methods for the Design of Real-Time Systems*. Bertinoro, Italy: Springer, 2004, pp. 200–236.

[11] A. David, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, J. Van Vliet, and Z. Wang, "Statistical model checking for networks of priced timed automata," in *Proc. Int. Conf. Formal Modeling Anal. Timed Syst.* Aalborg, Denmark: Springer, 2011, pp. 80–96.

[12] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen, "Uppaal SMC tutorial," *Int. J. Softw. Tools Technol. Transf.*, vol. 17, no. 4, pp. 397–415, Aug. 2015.

[13] G. Agha and K. Palmskog, "A survey of statistical model checking," *ACM Trans. Model. Comput. Simul.*, vol. 28, no. 1, pp. 1–39, Jan. 2018.

[14] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: An open-source robot operating system," in *Proc. ICRA*, vol. 3, no. 3, 2009, p. 5.

[15] M. Askarpour, M. Rossi, and O. Tiryakiler, "Co-simulation of human-robot collaboration: From temporal logic to 3D simulation," 2020, *arXiv:2007.11737*. [Online]. Available: http://arxiv.org/abs/2007.11737

[16] E. Rohmer, S. P. N. Singh, and M. Freese, "CoppeliaSim (formerly V-REP): A versatile and scalable robot simulation framework," in *Proc. IROS*, Nov. 2013, pp. 1321–1326.

[17] P. Payne, M. Lopetegui, and S. Yu, "A review of clinical workflow studies and methods," in *Cognitive Informatics*. Cham, Switzerland: Springer, 2019, pp. 47–61.

[18] R. Acharya, N. Kannathal, O. W. Sing, L. Y. Ping, and T. Chua, "Heart rate analysis in normal subjects of various age groups," *Biomed. Eng. OnLine*, vol. 3, no. 1, p. 24, Dec. 2004.

[19] T. Chettibi, M. Haddad, H. E. Lehtihet, and W. Khalil, "Suboptimal trajectory generation for industrial robots using trapezoidal velocity profiles," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Oct. 2006, pp. 729–735.

[20] H. Vangheluwe, J. De Lara, and P. J. Mosterman, "An introduction to multi-paradigm modelling and simulation," in *Proc. Int. Conf. AI, Simulation Planning High Autonomy Syst.*, 2002, pp. 9–20.

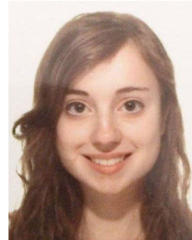
[21] Y. Van Tendeloo and H. Vangheluwe, "The modelverse: A tool for multi-paradigm modelling and simulation," in *Proc. Winter Simulation Conf. (WSC)*, Dec. 2017, pp. 944–955.

[22] J. M. O'Kane, "A gentle introduction to ROS," Dept. Comput. Sci. Eng., Univ. South Carolina, Columbia, SC, USA, Tech. Rep. 1, 2014.

[23] R. Halder, J. Proenca, N. Macedo, and A. Santos, "Formal verification of ROS-based robotic applications using timed-automata," in *Proc. IEEE/ACM 5th Int. FME Workshop Formal Methods Softw. Eng. (FormaliSE)*, May 2017, pp. 44–50.

[24] D. Tardioli, R. Parasuraman, and P. Ögren, "Pound: A multi-master ROS node for reducing delay and jitter in wireless multi-robot networks," *Robot. Auto. Syst.*, vol. 111, pp. 73–87, Jan. 2019.

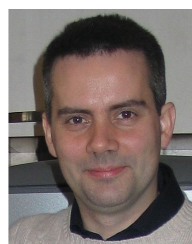
[25] O. Svenson, "Decision making and the search for fundamental psychological regularities: What can be learned from a process perspective?" *Organizational Behav. Hum. Decis. Processes*, vol. 65, no. 3, pp. 252–267, Mar. 1996.
[26] S. Konz, "Work/rest: Part II-the scientific basis (knowledge base) for the guide 1," *Ergonom. Guidelines Problem Solving*, vol. 1, no. 401, p. 38, 2000.
[27] M. Lutz, D. Stampfer, A. Lotz, and C. Schlegel, "Service robot control architectures for flexible and robust real-world task execution: Best practices and patterns," *Informatik*, vol. P-232, pp. 1295–1306, Jul. 2014.
[28] N. Li, C. Tsigkanos, Z. Jin, Z. Hu, and C. Ghezzi, "Early validation of cyber–physical space systems via multi-concerns integration," *J. Syst. Softw.*, vol. 170, Dec. 2020, Art. no. 110742.
[29] K. Yoshino, T. Motoshige, T. Araki, and K. Matsuoka, "Effect of prolonged free-walking fatigue on gait and physiological rhythm," *J. Biomech.*, vol. 37, no. 8, pp. 1271–1280, Aug. 2004.
[30] (2020). *HRI Deployment*. [Online]. Available: https://github.com/LesLivia/hri_deployment
[31] (2020). *HRI Deployment Demo*. [Online]. Available: https://tinyurl.com/y2p2lrbx
[32] M. Askarpour, D. Mandrioli, M. Rossi, and F. Vicentini, "Formal model of human erroneous behavior for safety analysis in collaborative robotics," *Robot. Comput. Integr. Manuf.*, vol. 57, pp. 465–476, Jun. 2019.
[33] J. Ernits, E. Halling, G. Kanter, and J. Vain, "Model-based integration testing of ROS packages: A mobile robot case study," in *Proc. Eur. Conf. Mobile Robots (ECMR)*, Sep. 2015, pp. 1–7.
[34] G. Kanter and J. Vain, "Model-based testing of autonomous robots using TestIt," *J. Reliable Intell. Environ.*, vol. 6, no. 1, pp. 15–30, 2020.
[35] R. Wang, Y. Guan, H. Song, X. Li, X. Li, Z. Shi, and X. Song, "A formal model-based design method for robotic systems," *IEEE Syst. J.*, vol. 13, no. 1, pp. 1096–1107, Mar. 2019.
[36] F. Vicentini, M. Askarpour, M. G. Rossi, and D. Mandrioli, "Safety assessment of collaborative robotics through automated formal verification," *IEEE Trans. Robot.*, vol. 36, no. 1, pp. 42–61, Feb. 2020.
[37] A. Malik, P. S. Roop, S. Andalam, M. Trew, and M. Mendler, "Modular compilation of hybrid systems for emulation and large scale simulation," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, pp. 1–21, Oct. 2017.
[38] J. Eddeland, J. G. Cepeda, R. Fransen, S. Miremadi, M. Fabian, and K. Åkesson, "Automated mode coverage analysis for cyber-physical systems using hybrid automata," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 9260–9265, 2017.
[39] S. Bak, O. A. Beg, S. Bogomolov, T. T. Johnson, L. V. Nguyen, and C. Schilling, "Hybrid automata: From verification to implementation," *Int. J. Softw. Tools Technol. Transf.*, vol. 21, no. 1, pp. 87–104, Feb. 2019.
[40] K. Zheng, D. F. Glas, T. Kanda, H. Ishiguro, and N. Hagita, "Designing and implementing a human–robot team for social interactions," *IEEE Trans. Syst., Man, Cybern. Syst.*, vol. 43, no. 4, pp. 843–859, Jul. 2013.
[41] Y. Kim and W. C. Yoon, "Generating task-oriented interactions of service robots," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 44, no. 8, pp. 981–994, Aug. 2014.
[42] J. Quintas, G. S. Martins, L. Santos, P. Menezes, and J. Dias, "Toward a context-aware human–robot interaction framework based on cognitive development," *IEEE Trans. Syst., Man, Cybern. Syst.*, vol. 49, no. 1, pp. 227–237, Jan. 2019.
[43] P. Rueckert, S. Muenkewarf, and K. Tracht, "Human-in-the-loop simulation for virtual commissioning of human-robot-collaboration," *Proc. CIRP*, vol. 88, pp. 229–233, Jan. 2020.
[44] M. Webster, D. Western, D. Araiza-Illan, C. Dixon, K. Eder, M. Fisher, and A. G. Pipe, "A corroborative approach to verification and validation of human–robot teams," *Int. J. Robot. Res.*, vol. 39, no. 1, pp. 73–99, Jan. 2020.
[45] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Comput. Surv.*, vol. 41, no. 4, pp. 1–36, 2009.
[46] M. Gleirscher and D. Marmsoler, "Formal methods in dependable systems engineering: A survey of professionals from Europe and North America," *Empirical Softw. Eng.*, vol. 25, no. 6, pp. 4473–4546, Nov. 2020.



LIVIA LESTINGI received the M.Sc. degree in automation engineering from the Politecnico di Milano, in 2017, where she is currently pursuing the Ph.D. degree in information technology. Her research interests include the analysis of human–robot interaction through formal methods and formal modeling techniques of human behavior.

MEHRNOOSH ASKARPOUR is currently an Adjunct Assistant Professor with McMaster University. Her current research interests include verification of safety-critical system properties and application of formal methods for safe robotics and autonomous vehicles.

MARCELLO M. BERSANI is currently a Senior Assistant Professor with the Politecnico di Milano. His research interests include formal methods, temporal logic, and verification.



MATTEO ROSSI is currently an Associate Professor with the Politecnico di Milano. His research interests include formal methods for safety-critical and real-time systems, architectures for real-time distributed systems, and transportation systems both from the point of view of their design, and of their application in urban mobility scenarios.