The Journal of Systems and Software 231 (2026) 112654

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Novel framework for automated testing of ill-defined human–robot interaction environments☆

Aitor Aguirre-Ortuzar [a],*, Íñigo Elguea-Aguinaco [b], Nestor Arana-Arexolaleiba [a], Leire Etxeberria-Elorza [c], Joseba A. Agirre-Bastegieta [c]

[a] Mondragon Unibertsitatea, Electronics and Computer Science Department - Intelligent Systems for Industrial Systems research group, Mondragon, Spain
[b] Electrotécnica Alavesa S.L. (Aldakin Group), Vitoria-Gasteiz, Spain
[c] Mondragon Unibertsitatea, Electronics and Computer Science Department, Mondragon, Spain

ARTICLE INFO

Keywords:
Automated oracles
Cognitive systems
Human–robot interaction
Industrial manipulators
Testing

ABSTRACT

As automated systems advance in complexity, comprehensive testing becomes crucial, particularly for human–robot interaction (HRI) environments where human unpredictability creates ill-defined testing domains that challenge conventional software testing approaches. In interactive robotics, evaluation criteria extend beyond performance to include critical safety considerations. This paper introduces a novel automated testing framework combining runtime monitoring with constraint-based techniques for HRI environments. The framework employs a three-level cognitive oracle architecture – observation, interpretation, and diagnosis – that automatically evaluates the correctness of human and robot actions without requiring expert human intervention. The approach uses constraint-based modeling to handle the non-deterministic nature of HRI scenarios while ensuring safety compliance. Validation through five test cases in a refrigerator disassembly simulation demonstrates the framework's effectiveness in detecting safety violations and procedural errors under environmental uncertainties.

1. Introduction

In recent years, the integration of advanced sensing, data processing, and decision-making technologies has launched a new era of automated systems and smart manufacturing (Lin et al., 2017; Li et al., 2021). However, these enhanced functionalities of automated systems are frequently demonstrated in developmental prototypes, which may lag behind production versions regarding performance and safety. Consequently, before deployment within an actual manufacturing setting, the capabilities of these automated systems require rigorous testing and validation. In this regard, a commonly adopted approach involves using oracles (Liu and Nakajima, 2020).

In a testing context, an oracle serves as a mechanism to determine a system or application's anticipated outcome or behavior under test (Harman et al., 2013). Oracles can be manual (Jahangirova, 2017) or automated (Pezze and Zhang, 2014), and can come from a variety of sources, such as requirements documents that specify the expected behavior of a system or domain experts who can provide insights into the expected behavior of a system based on their domain knowledge. By employing an oracle during the testing process, testers can compare the actual behavior of the system or application against the expected behavior, thereby identifying any discrepancies that may indicate the presence of defects or issues in the software.

In this regard, automated testing refers to using software tools to automate the execution of test cases and the subsequent comparison of the actual results with the expected results. This kind of testing offers several advantages, including faster and more efficient testing, improved test coverage, repeatability, and support for continuous integration and delivery of the developed system (Zhao et al., 2017).

Automated testing is a well-known research area in the software field; however, it poses a significant challenge in human–robot interaction (HRI) environments (Araiza-Illan et al., 2016a; Huck et al., 2023). Unlike conventional software testing, HRI environments can be characterized as ill-defined domains (Lynch et al., 2006), where highly unpredictable actions can arise due to the presence of humans during a task. Additionally, HRI environments encompass various types of hazards, including collision, entanglement, and crushing hazards, as well as electrical or chemical hazards, among others (Rubagotti et al., 2022). In such scenarios, an oracle should be able to observe

☆ Editor: T. Mårtensson.

* Corresponding author.
E-mail addresses: aaguirre@mondragon.edu (A. Aguirre-Ortuzar), ielguea@aldakin.com (Í. Elguea-Aguinaco), narana@mondragon.edu (N. Arana-Arexolaleiba), letxeberria@mondragon.edu (L. Etxeberria-Elorza), jaagirre@mondragon.edu (J.A. Agirre-Bastegieta).

https://doi.org/10.1016/j.jss.2025.112654
Received 2 June 2025; Received in revised form 2 October 2025; Accepted 6 October 2025
Available online 8 October 2025
0164-1212/© 2025 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Fig. 1. (a) Depiction of an actual refrigerator disassembly line. Within this workstation, workers initiate the disassembly process by opening the refrigerators, extracting internal components like drawers and shelves, and proceeding to detach the magnetic gasket affixed to the doors. (b) Transformation of the disassembly line from the previous illustration into a HRI environment, featuring the integration of a collaborative robot. The robot, leveraging RL manipulation skills, performs the magnetic gasket extraction autonomously.

and diagnose the activities performed by humans and robots during a specific task, enabling automated testing without human supervision. These requirements extend beyond the scope of conventional software testing, resulting in a dearth of methodologies and frameworks that specifically address this issue.

Therefore, this paper introduces a comprehensive Artificial Intelligence (AI)-based framework designed to facilitate automated testing in HRI environments. The framework addresses the specific requirements of HRI environments by offering two key components. Firstly, a protocol that enables the definition and execution of automated tests in HRI environments is presented. Secondly, the framework incorporates an oracle grounded on constraint-based modeling techniques that resembles a cognitive system capable of diagnosing and interpreting the activities of robots and humans within HRI environments. This system aims to imitate an expert in testing the robot in a human-in-the-loop environment. To assess the efficacy of the proposed framework, a series of tests was conducted using a simulated HRI environment, where a worker interacted with a collaborative robot controlled by reinforcement learning (RL) in the disassembly of magnetic gaskets from refrigerator doors (Fig. 1).

The contributions of this study are as follows:

- A testing framework capable of diagnosing the correctness of the underlying process in ill-defined and non-deterministic HRI environments.
- A methodology to integrate distributed interactive systems (e.g., robotic systems), enabling the necessary communication and coordination for testing the interaction between HRI environments under test.
- Automated end-to-end testing of procedural tasks through an AI-based cognitive framework eliminates the need for a human-in-the-loop approach.

The content of the paper is organized as follows. Section 2 contains a brief description of related works. In Section 3, a technical overview of the proposed system is presented. Section 4 provides a comprehensive explanation of the functioning of the cognitive framework that evaluates task correctness. Section 5 presents the generation and analysis of test cases in the HRI environment to demonstrate the effectiveness of the framework and the corresponding results obtained. Section 6 outlines the primary areas to be explored in future research, offering points for discussion and reflection. Lastly, Section 7 concludes this study by providing a summary of the acquired knowledge.

2. Related works

Recently, demonstrating the trustworthiness of automation environments by providing formal assurances has become a central concern in verification, validation, and testing endeavors. Approaches to assess system reliability in HRI environments commonly involve formal verification during the design phase, and test-based methods (Kress-Gazit et al., 2021).

2.1. Formal verification methods

Formal verification is a well-known method characterized by testing a system's correctness by mathematically demonstrating that it complies with the specified requirements and ensuring the absence of design faults (Holzmann and Peled, 1995). Within this method, two of the most notable approaches are model checking (Clarke et al., 2018) and theorem proving (Fitting, 2012), the former being used more commonly in HRI environments testing (Choi et al., 2021).

Conventional model checking is a well-established technique for verifying the properties of deterministic systems. In this technique, the system is represented as a finite-state machine, and the properties to be verified are typically expressed in linear-time logic (Farulla and Lamprecht, 2017).

Probabilistic model checking, on the other hand, serves to verify the properties of systems with stochastic behavior. In this case, the system is modeled as a probabilistic model where each state has associated probabilities that represent the likelihood of transitioning from one state to another based on the available actions or inputs (Junges et al., 2016). Probabilistic model checking is a prevalent technique applied in testing HRI environments, owing to its ability to model scenarios characterized by uncertainty resulting from human behavior.

For instance, Lestingi et al. (2020) presented a model-driven approach to verifying HRI environments in healthcare using probabilistic model checking. To this end, the authors formally modeled the scenarios using hybrid automata, which were augmented with stochastic components through probabilistic distributions. For the human aspect, the behavior was exclusively modeled for idle or walking conditions, with differential equations describing the temporal dynamics of fatigue. This study was further extended to introduce a framework enabling the generation of unfolding code from a specific subset of stochastic hybrid automata (Lestingi et al., 2021). This framework ensured the comparability of the observable behavior with the deployed version of the scenario to that defined during the design phase.

Nevertheless, probabilistic model-checking approaches still exhibit certain limitations when applied to HRI environments. These limitations are frequently associated with the formalization of models and the complexity of defining comprehensive properties for HRI, primarily due to the inherent unpredictability linked to human behavior and non-determinism. Obtaining reliable and sufficient data to capture the full range of human behavior can be a significant challenge. Furthermore, verifying large-scale models of HRI systems can result in

substantial computational overheads. Therefore, models simplify reality to make it tractable for analysis. Consequently, incorporating stochastic components and using statistical model checking further necessitate assumptions and abstractions, which may introduce biases and limit the scalability of the results.

2.2. Test-based methods

Test-based methods serve as an alternative to formal approaches in the realm of verification and validation. In particular, simulation-based testing (Wang et al., 2023) plays a crucial role in subjecting the system under test (SUT) to more lifelike stimuli and improving scalability, surpassing the often overly abstract scenarios suitable for formal verification. Simulation-based testing offers a faster and less labor-intensive means to achieve verification goals compared to conventional real-world testing. Coverage, as a metric of verification progress, enables monitoring of the diversity of tests used and assessing their effectiveness and the safety of the system in achieving verification objectives (Araiza-Illan et al., 2016b, 2015). In Huck et al. (2020), for instance, the authors introduced a simulation-based safety testing method using a Monte Carlo tree search algorithm to control a human model. The algorithm was optimized to maximize a risk metric, leading to the generation of high-risk scenarios. Nevertheless, the modeling process involved certain simplifications, and the evaluation of safety criteria focused solely on speed and distance, omitting collision-related considerations.

2.3. Hybrid methods

In practice, neither formal verification nor simulation-based testing alone is sufficient to cover the testing of an entire system comprehensively. Separately, these techniques may not fully explore the entire state space of interactions in realistic detail. Consequently, some studies resort to a combination of these techniques to achieve greater confidence in the correctness of results. For example, Webster et al. (2020) integrated model-checking, simulation-based testing, and user verification in experiments involving a real robot for safety testing in cooperative handover tasks. This approach enabled them to provide comprehensive guarantees of safety and functional correctness, even when discrepancies between the different verification and validation techniques were identified.

Notwithstanding, the authors used a Practical, Robust Implementation and Sustainability Model (PRISM) (Feldstein and Glasgow, 2008) for probabilistic model checking, which can become excessively intricate, particularly when dealing with large and intrinsic systems, leading to over-abstraction and limitations in modeling systems with continuous probabilities or continuous-time behavior. Notably, for systems with high dimensionality and uncertainty, the use of state machines is generally discouraged, since the state dimensionality grows exponentially with the number of variables, resulting in state explosion problems that render analysis computationally infeasible and undermining the practical applicability of the approach.

2.4. Challenges in HRI testing and runtime monitoring

Despite being procedural tasks, successful HRI tasks may depend on a combination of robot and human decision-making processes, encompassing factors such as judgment, creativity, problem-solving skills, and adaptability to the environment. The use of AI technologies for robot control is becoming increasingly prominent (Elguea-Aguinaco et al., 2023). Despite efforts to develop frameworks that enable the verification of machine learning algorithms (Gross et al., 2022), testing these algorithms within complex HRI environments remains challenging, often limited to checking partial specifications (Van Wesel and Goodloe, 2017). The specific motor skills of human operators may also play a key role in effectively accomplishing the given task. Assessing human motor skills involves evaluating physical actions and capabilities, which are not easily captured in a finite-state machine or formal model. Human motor skills are influenced by various factors such as muscle coordination, dexterity, reaction time, and sensory perception, making them difficult to represent using the discrete and abstract nature of model checking.

Recent studies have acknowledged the challenge of testing virtual reality (VR) applications, mainly due to the necessity of evaluating physical interactivity with the systems (Rzig et al., 2022). In Jiménez-Ramírez et al. (2023), the authors proposed an automated testing process for robotic automation projects. The test cases in their approach aimed to verify whether a robot or a human could replicate a specific behavior. However, the approach encountered misclassifications when evaluating human activity due to the inherent uncertainty of human behavior.

Given the constraints and drawbacks of the aforementioned testing approaches, runtime monitoring emerges as a promising alternative. This approach represents a dynamic verification technique in which the system behavior is observed and analyzed in real time, offering continuous feedback and promptly identifying potential violations to guarantee the correctness and robustness of the system (Leucker and Schallhart, 2009). The use of this technique as an approach to testing HRI environments has been relatively underrepresented in the existing literature due to the absence of formal verification guarantees. Nevertheless, this approach can effectively address dynamic environments, handling uncertainties, stochasticity, and partial specifications. In this context, we propose a novel framework that integrates runtime monitoring with simulation-based testing to verify and validate high-dimensional and uncertain HRI environments. By combining these techniques, the framework enables real-time verdicts of human, robotic, or other interactive systems' activities. To the best of the authors' knowledge, this represents the first framework that facilitates assessments of procedural tasks, including evaluating motor skills, within ill-defined domains.

3. Overall system design

The proposed testing framework comprises three modules: the test controller, the SUT, and the oracle. This oracle is known as ULISES (see Fig. 3) and it provides real-time event-driven automated verdicts for test execution in simulation environments through constraint-based oracles. The SUT comprises various interactive systems that communicate with one another in a simulated environment controlled by the test controller. These interactive systems can be any device that interacts directly with a human operator, such as a robot or a camera. For the use case presented in this paper, those interactive systems are deployed within a ROS-based system. Regarding the test controller, it is developed based on the *rostest* package (Foundation, 2019), which extends *roslaunch* by allowing the definition of test nodes. This makes it possible to combine the SUT and the supporting infrastructure launched through *roslaunch* with Python unit tests that execute in the same controlled context. Specifically, each launch file starts the SUT, the NVIDIA Isaac Sim simulator (a physics-based robotics simulation platform), the ROS–MQTT bridge, and additional monitoring nodes (distance calculation, motion planning with MoveIt, and computer vision). The inclusion of these elements ensures that every test begins in a controlled environment, independent of previous executions. The public repository valu3s_uc7 (MGEP, 2024b) provides several examples containing all necessary elements to launch both the test controller and the SUT, while the ULISES oracle executes separately alongside a Mosquitto broker.

The first stage in running a test involves initiating the simulation, which can be executed within any simulation engine. For the use case presented in this paper, the simulation has been implemented using NVIDIA's open platform Isaac Sim, whose SDK engine serves as a flexible platform for the development and implementation of modular

```xml
<launch>
  <!-- Enable simulation clock -->
  <param name="/use_sim_time" value="true" />

  <!-- Start Isaac Sim and environment -->
  <include file="$(find simulation)/launch/default.launch"/>

  <!-- Start MQTT bridge -->
  <include file="$(find mqtt_bridge)/launch/demo.launch"/>

  <!-- Launch distance monitoring and robot planning -->
  <include file="$(find skros_service_tf_distance)/launch/default_demo.launch"/>
  <include file="$(find iiwa_moveit_config)/launch/default.launch"/>

  <!-- Execute the tests -->
  <test test-name="uc7_func_normal" pkg="valu3s_uc7" type="test_uc7_func_normal.py"/>
  <test test-name="uc7_door_not_opening" pkg="valu3s_uc7" type="test_uc7_door_not_opening
</launch>
```

```python
from ulises_test_utils.test_utils import (
    TestLaunchInfo, UlisesTestHelper
)

class FridgeDisassemblyTest(unittest.TestCase):
    def test_fridge_disassembly(self):
        # Define test metadata
        test_info = TestLaunchInfo(
            observation_model_name='FridgeDisassemblyOM.xml',
            task_id='3ef9c5db-630d-4537-b30c-95831447aeb6',
            test_id='1'
        )

        # Synchronize with ULISES
        with UlisesTestHelper(mqtt_host="172.17.6.163", test_launch_info=test_info) as helper:
            # Interact with the SUT
            task.control_piston("pistons_01", True)
            task.control_conveyor("conveyor_00", True)
            task.control_fridge_doors("fridge_00", ["up", "down"], True)
            # ... more interactions ...with the SUT/SIMULATION ENVIROMENT
            # Wait for diagnostic assertion from ULISES
            helper.wait_assert_passed()
```

Fig. 2. An example of the rostest file (left) and the Python test (right).

applications on real robots. In this context, the test controller is responsible for executing the simulation environment, which is composed of different ROS nodes in this case.

On the testing side, every test case is implemented as a Python unit test using the standard unittest library (Python Software Foundation, 2025). To coordinate with ULISES, a ROS Python module test library (MGEP, 2024a) has been developed, which synchronizes ROS test executions with the ULISES oracle via MQTT. At the initiation of each test, an observation model in the form of an XML file is transmitted to ULISES, specifying the test to be executed and the requirements to be evaluated. This observation model defines all the elements to be monitored, along with their features, referred to as properties. These elements, known as observations, are then relayed to the oracle's runtime kernel, which interprets and diagnoses the ongoing tasks in real-time throughout each simulation cycle. Following the analysis of all ROS topics, the oracle transmits the test result via an MQTT message to the test. Based on this message, the test employs an assertion to determine the correctness of the test execution, enabling automated evaluation of actions performed by all interactive systems without human intervention. Examples of test cases can be found in the UlisesTestUtils repository (MGEP, 2024a) or in the Valu3s_uc7 repository (MGEP, 2024b). Fig. 2 shows an example rostest file and a Python test file.

Once the simulation environment becomes active, the ROS–MQTT bridge node is launched to facilitate communication between the simulation and the MQTT broker. The ROS–MQTT bridge plays a crucial role in enabling communication between the ROS-based simulation and the ULISES oracle. The configuration is specified through a YAML file, where each mapping explicitly defines which ROS topics are translated into MQTT messages and vice versa. In our use case, all relevant robot and environment signals are published: conveyor commands and states, fridge door states, robot joint states, and human–robot distance estimations. These are serialized in JSON format and transmitted to an MQTT broker, from which ULISES consumes the data. This setup allows ULISES to observe the ongoing simulation in real-time according to the observation model provided at the start of the test. The complete communication process is depicted in Fig. 3

In addition to the MQTT bridge node, other nodes can be added to the HRI environments under evaluation. For the specific use case being presented in this paper, where a fridge disassembly task is being tested, three other nodes are implemented and launched. The first node continuously monitors the spatial distance between the human collaborator's skeleton and the robot. This information is essential for ensuring safety and preventing any potential collisions or hazardous situations. The second node that has been implemented is the MoveIt node, responsible for motion planning and control of the collaborative robot within the environment. The third node is a computer vision algorithm that allows the identification of the refrigerator and its doors in the workspace. Through the integration of these nodes, the framework enables comprehensive monitoring and control of the HRI dynamics during the simulated scenario.

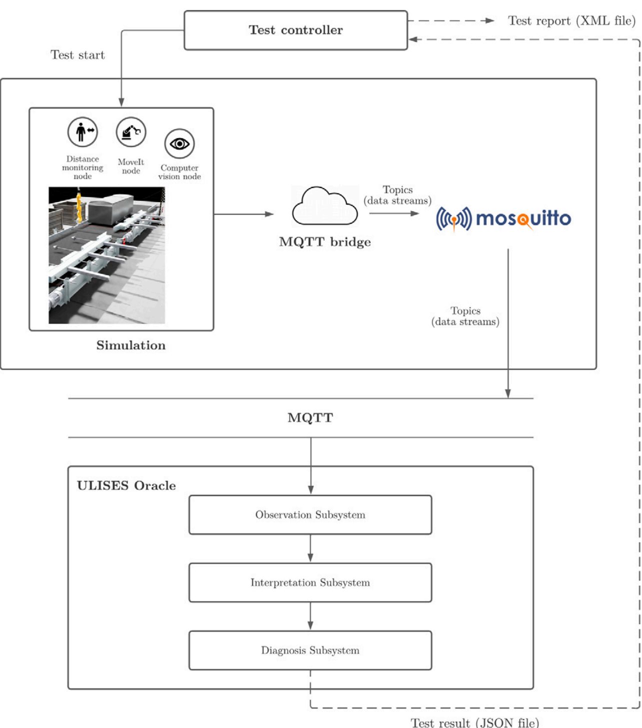

Fig. 3. Communication between the test controller, SUT and the ULISES oracle.

For a more comprehensive understanding of the individual levels constituting ULISES, a more detailed explanation is provided in the following section.

4. HRI testing through ULISES oracle

4.1. ULISES overview

ULISES is a framework that enables the integration of a Virtual Reality Interactive System (VRIS) to create a cognitive oracle within that VRIS. This framework was previously applied to create Intelligent Learning Systems in different VRISs (Aguirre et al., 2012). The framework aims at creating an intelligent systems that replicate the behavior of a human expert who supervises different agents (e.g., human workers or robots) engaged in procedural tasks. During the testing phase of a robot or operator in an actual HRI problem, a human would be able to observe the environment (e.g., the production line), interpret the actions of the agents, and emit a verdict of the correctness of those actions. Likewise, the human tester could conclude and explain the reasons for failures or mistakes while solving the procedural task.

In this context, the framework comprises three core levels that are independent of the domain that is being tested or diagnosed to emulate the assessment a human would make. Those three levels are the Observation, Interpretation, and Diagnosis levels. Experts' knowledge can be implemented in the system by defining these three models. The Observation level integrates events and data streams, transforming them into observation elements that model their significance. Each observation element is domain-independent and encapsulates a fact observed over a period of time. The Interpretation and Diagnosis levels map a human or robot's activity through the use of step and situation elements, which are derived from observations. These three elements facilitate the representation of any procedural activity, including tasks involving motor skills.

Fig. 4 shows the general functioning of the ULISES framework. The ULISES metamodel is defined by three levels of abstraction that generically describe the elements required for each ULISES subsystem to build an oracle for a specific VRIS. These elements specify how observations in the VRIS are described, how they are interpreted, and how tasks are diagnosed. Through particularization for a specific domain, specific models of Observation, Interpretation, and Tasks are generated. This process enables seamless communication and integration between the VRIS and ULISES oracle, with the observation model determining the elements for communication and the task model defining the elements for the task-solving process. Consequently, the framework facilitates the evaluation of test outcomes and determines whether a test is successful.

The runtime kernel of ULISES is designed using a multi-agent architecture that follows the Foundation for Intelligent Physical Agents (FIPA) standard of interoperability. This standard enables communication among the three subsystems within the kernel and also with other external applications. This communication is based on standard subscriptions, requests, and queries, so the agents exchange information only when necessary. The main agents within the system are the input–output, listener, observer, interpretation, and diagnosis agents, which are further described in the following subsections. This FIPA-compliant multi-agent architecture enables horizontal scalability by allowing agents to be distributed across multiple computing nodes while maintaining seamless communication through standardized protocols, making the framework suitable for large-scale industrial HRI environments.

4.2. Observation subsystem

The Observation Subsystem encompasses a set of methodologies that aim to effectively merge and analyze event streams and data derived from the simulation environment, enabling the inference and modeling of their underlying meaning. The output of this analytical process is encapsulated within observation elements that represent significant and observed facts that unfold over a specific period of time.

The observation element is a continuous element defined by a starting and ending timestamp. Additionally, an observation can be defined with properties that further characterize its attributes. For instance, in an HRI setting, the observation agent may observe a specific joint of a robot, which may possess properties such as location, speed, or acceleration. These properties can assume different types, including numeric and archs. As described in Aguirre et al. (2014), the Arch element allows for describing movements with specific properties such as direction, speed, acceleration, concavity, and convexity. This element represents the basis for the posterior diagnosis of complex trajectories.

As previously mentioned in Section 4.1, the runtime kernel comprises several agents, including listener and observer agents. The listener agents are responsible for communicating a VRIS with the observer agent. In the case of a distributed simulated environment, several listener agents can be integrated. On the other hand, the observation agent encapsulates the Observation Subsystem, gathering simulation information from listener agents and transforming it into observations.

For the specific use case presented in this paper, where the SUT is integrated within a Robotic Operating System (ROS) platform, two agents, namely the so-called ROSListenerAgent and ROSObserverAgent, were implemented: the first one actively monitors and listens to events coming from the simulation environment and publishes preprocessed data for other components within the system that are subscribed to this information. The latter agent transforms the subscribed data (specified in the observation model), generates observations, and publishes them so that the Interpretation and Diagnosis subsystems use them as input.

Within this context, the Observation Subsystem runs the list of specifications of the observation model (can be found at MGEP (2025a)) and calls the corresponding observers, such as the RosObserverAgent, to check the occurrence at a specific time step. As a result, this subsystem generates new observations when they are detected, extends the duration of existing observations, and ends them when they are no longer observed.

4.3. Interpretation subsystem

Once the activity within a VRIS is observed, the subsequent step involves interpreting that activity. The Interpretation level provides a generic description of recognizing human or robot activity and conveys this interpretation to the Diagnosis Subsystem in real-time through the Interpreter Agent. The framework introduces two key elements for activity interpretation: steps and situations. The former encompasses the essential attributes for the Interpretation Subsystem to identify human or robot activity. The latter provides the information required to define the context conditions under which each step can be executed. A step will only be interpreted if the context of the corresponding situation is presented within the VRIS. For instance, in the case of a disassembly task in an HRI environment, the oracle needs to determine whether the operator is in a collaborative situation or if the robot is stationary while the human performs the disassembly independently. If the oracle specifically tests a disassembly step in a collaborative situation, all actions performed outside that situation would not be diagnosed. It is important to note that the step element should be interpreted regardless of the correctness of the action, therefore, representing the intentionality of such an action.

Both steps and situations are continuous actions and are described using different types of constraints. These constraints can be applied to observations, situations, and steps. As described in Section 4.2, an observation specifies several attributes related to its temporality and its different kinds of properties. Therefore, the interpretation level defines a methodology to establish constraints by considering all these features. Constraints are explained with different examples that have been applied in the use case (Table 4).

- Temporal constraints: These constraints can relate qualitatively and quantitatively to the intervals (Allen and Ferguson, 1994). Relations can be defined over punctual events, over the duration of different intervals, or over temporal relations between observations, steps, or situations. For instance, example 1 shows how the *ExtractGasket* step should happen within the *OpenFridge* step:

$$OpenFridge[Contains]ExtractGasket \quad (1)$$

- Logical constraints: They allow for establishing logical relations between different constraints. Example 2 specifies that the human's hip and right and left hands should be at least 0.25 m from the robot:

$$\begin{gathered}(Hip.DistanceToRobot > 0.25)AND\\(LeftHand.DistanceToRobot > 0.25)AND\\(RightHand.DistanceToRobot > 0.25)\end{gathered} \quad (2)$$

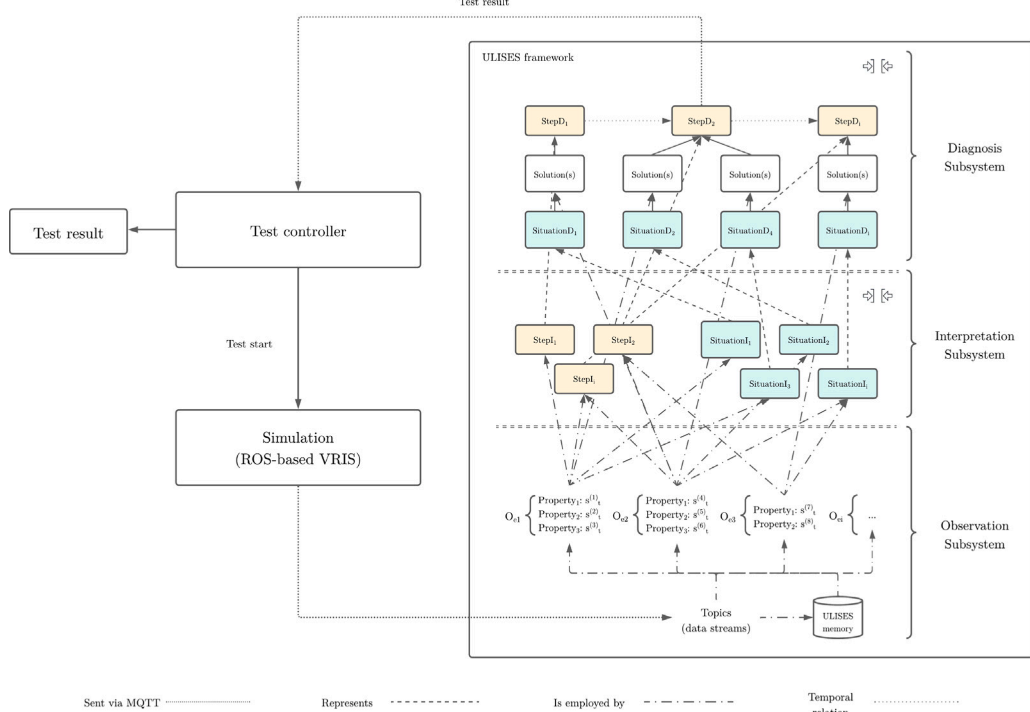

Fig. 4. General architecture of ULISES framework.

- Constraints over properties: These constraints provide a way to limit the values of the observation properties. This type of constraint can also be used to apply constraints over movements (Aguirre et al., 2014). Example 3 defines that the piston of the first conveyor needs to be active.

$$Conveyor.joint_piston_1_state == 1.0 \quad (3)$$

During each interpretation cycle, the interpreter assesses constraints against a set of observations to determine their compliance, retaining the constraint evaluations and corresponding observations in memory. This information is continuously updated to ensure temporal coherence. Apart from constraint evaluations, the Interpretation Subsystem also needs to ascertain the interpretation state of a step or situation, mimicking the reasoning of a real domain expert. For instance, if a robot is in the midst of a movement, an expert might infer that the robot has recently ceased the movement or is continuing with the same action. According to this schema, the interpreter determines whether an interval is initiated, concluded, or extended based on the potential outcomes of constraint evaluation.

This assessment is performed through three different types of constraints:

- General constraints: When these constraints between intervals are detected, the steps are carried out. When they are not met, the Interpreter Subsystem finalizes the step. Therefore, it will not be diagnosed in the upper Diagnosis Subsystem either.
- Initial constraints: When these constraints are met, it will indicate that a step has begun (e.g., the AssemblyFinished step in Table 3). If such constraints are not defined, a step will begin when the general constraint is met.
- End constraints: When these constraints are met, the step will be finished even if the general constraint is still met. If this attribute is not defined, the step will be finalized when the general constraint is not fulfilled.

The uniqueness of this approach lies in the temporal handling of observations and the constraint satisfaction process itself. Unlike conventional methods that solely manage punctual events, this approach deals with observations as time intervals containing properties represented as data streams. The type of constraints that the framework handles and the constraint satisfaction process for steps and situations are further explained in Aguirre et al. (2014).

4.4. Diagnosis subsystem

The Diagnosis Subsystem is responsible for diagnosing the activity of the VRIS and, thus, detecting errors from human and robot activity or other devices while performing a task. This subsystem gathers the observations generated by the Observation Subsystem and the steps and situations interpreted by the Interpretation Subsystem. This information is then sent to the specific diagnosis agent. It is worth noting that the Diagnosis Subsystem supports different diagnosis techniques. In this paper, the implemented oracle utilizes a constraint satisfaction-based diagnosis technique suited to the nature of the applied domain and its ill-defined characteristics. In such domains, the explicit definition of steps to achieve a successful outcome may be lacking, and the sequence of steps and the number of actions that a human may take until a task is solved may vary. In addition, uncertainty and ill-definedness can be involved in evaluating procedural tasks, movements, or motor skills. In this context, the implemented constraint-based modeling allows for defining how certain actions should be performed to identify errors, offering advantages over other approaches such as plan recognition, Hidden Markov Models, or Machine Learning (ML) models, which may incur high computational overhead and high complexity. Moreover, unlike ML models, where solutions are typically classified as "correct" or "incorrect" without providing insight into the underlying causes of errors, constraint-based techniques can retrieve information regarding the specific causes.

The Diagnosis Subsystem follows a common process agnostic of the diagnosing technique. Similar to the Interpretation Subsystem, this subsystem keeps updated observations through notification messages received from the Observation Subsystem, including "new observation", "observation pending", "cancel observation", "update properties", "end observation", and "continue observation". Similarly, the

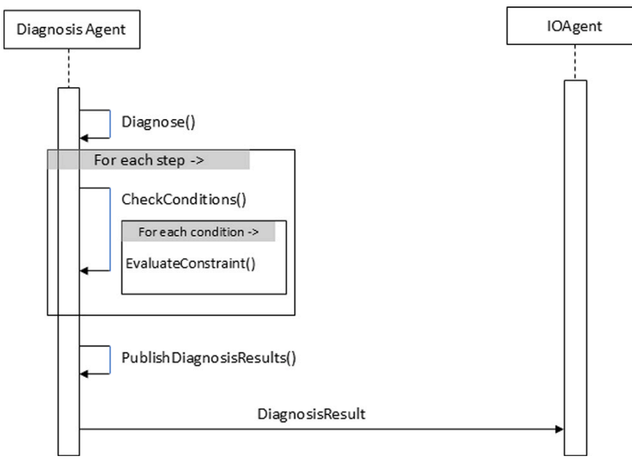

Fig. 5. Function call handling in the diagnosis process with the constraint-satisfaction-based technique. The diagnosis result is transmitted to an Input-Output Agent (IOAgent) so all the agents subscribed to this information can receive the diagnosis results.

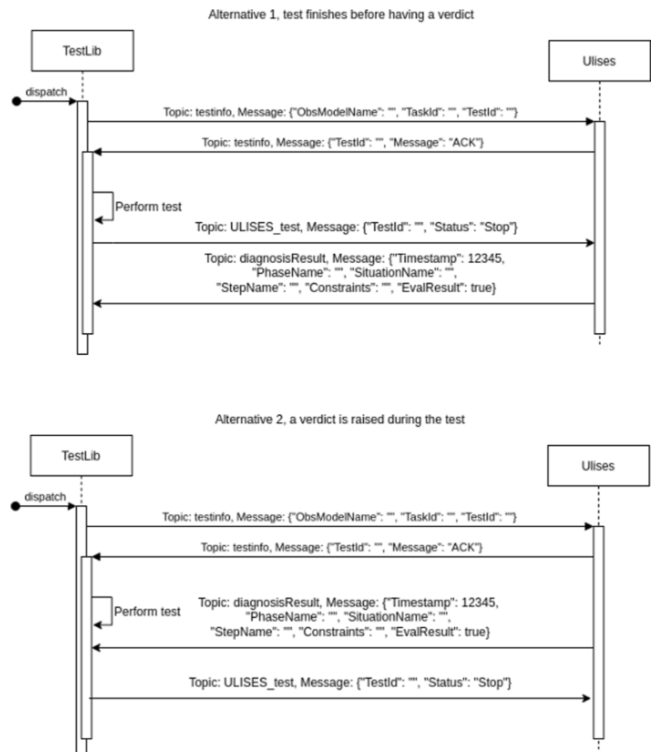

Fig. 6. A protocol that enables the communication between the testing library and the ULISES framework.

Interpreter notifies the Diagnosis Subsystem of state changes in steps and situations. Once all this information has been updated, the Diagnosis Subsystem diagnoses all active situation solutions by generating and sending the corresponding results.

A solution specifies how steps should be solved within a situation. Each step may have different conditions, which are defined through constraints. Therefore, when a condition is not fulfilled, an error is detected. Notably, this technique enables grouping actions that violate the same domain principle, facilitating the differentiation of error contexts as solutions are created within specific situations. This distinction is crucial, as the same error can have different interpretations depending on the situation (context) in which it occurs.

A solution specification can include multiple steps to define a solution to a situation, and each step may contain a set of conditions that determine its correctness. The Diagnose function (implemented by a diagnosis agent) is invoked during each diagnostic cycle for each active situation. This function evaluates the conditions defined in each step (Fig. 5). Each condition is associated with a constraint assessed when checking the condition. Additionally, the condition elements also incorporate other information, such as the weight of the condition regarding its importance in the situation, a penalization value within the context of the step, and the domain principle that the condition breaks (see Fig. 5).

After evaluating all the steps, a diagnosis result is generated. If the evaluation result is FALSE, indicating that at least one condition is not fulfilled, a test result is produced, which contains the following information:

- *Condition names*: name(s) of the conditions that were not met. With these conditions, the test controller can know why the test failed.
- *Timestamp*: timestamp of the instant the test result was generated.
- *Step name*: name of the step in which one or various constraints were unmet.
- *Situation name*: name of the situation in which the solution was evaluated.
- *Evaluation Result*: test result of a situation; it can hold TRUE or FALSE values.
- *TestId*: identification of the test that was executed by the oracle.

If the diagnostic evaluation yields a FALSE result, the diagnosis agent generates a test result that is subsequently published by an agent through MQTT. This publication serves as an alert to the test controller, indicating an unsuccessful test verdict (see Fig. 6). In response to an unsuccessful verdict, the test controller terminates the simulation, discounting the ongoing test. Conversely, if the evaluation result is TRUE for all steps across each simulation cycle, the oracle generates a test result marking the EvaluationResult attribute as TRUE once the simulation concludes.

A more detailed diagram of the observation, interpretation, and diagnosis layers can be found at MGEP (2025b).

4.5. Framework extensibility

As mentioned above, the framework's modular architecture enables seamless integration with any VRIS. Fig. 7 shows details about the relationship between the VRIS and the oracle. The framework's modular architecture enables seamless integration with different VRISs through a three-component approach that separates data acquisition, processing, and temporal evaluation responsibilities. VRIS Integration requires implementing a custom `VrisObserver` that establishes a connection to the VRIS endpoint and handles raw data reception from virtual environment events (hand positions, object interactions, gesture detection, etc.). However, the observer alone is insufficient, developers must also implement custom Spy Objects with two critical functions: `UpdateData()` to process incoming VRIS data and convert it into a standardized observation format, and `EvaluateObservation()` to apply domain-specific logic that determines when observations should start, continue, finish, or be cancelled.

The integration architecture includes a third essential component, the Spy Observer, which handles temporal aspects that the basic observer cannot manage. While the `VrisObserver` captures instantaneous VRIS events and the Spy Object processes individual data points, the Spy Observer implements temporal evaluation logic through its `UpdateData()` and `Evaluate()` functions to track observation duration, frequency analysis, and complex spatiotemporal relationships across time windows. For VRIS scenarios, this enables the creation of advanced observation properties that depend on time, such

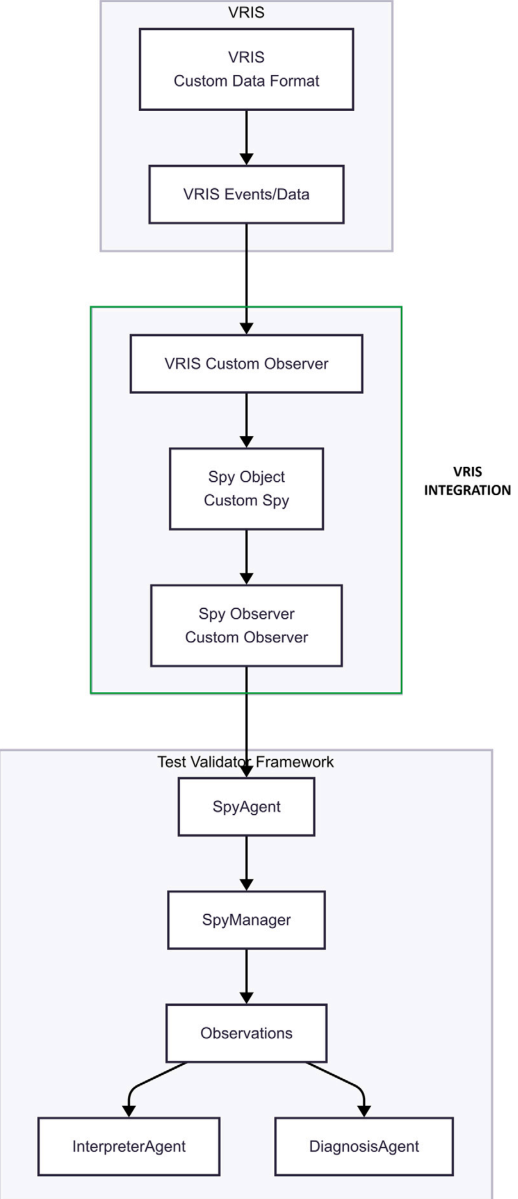

Fig. 7. Integration between the ULISES oracle and a VRIS. Amplitation of the integration provided in MGEP (2025b).

as getting the first or the second derivative of a specific observation property. This three-layer separation ensures that VRIS-specific data handling, constraint processing, and temporal analysis remain modular and maintainable while leveraging the framework's standardized observation and diagnosis infrastructure. Further information about this VRIS integration architecture can be found at MGEP (2025b).

Regarding technologies implemented in the ULISES oracle, the framework makes use of a mixed language architecture, which combines C++/CLI bridge components with managed .NET framework elements to enable seamless integration with C++-based simulators and interactive systems. The C++/CLI observer components (such as `RosObserver` and custom simulator observers) can directly utilize native C++ libraries, including ROS APIs, OpenCV, hardware drivers, and legacy system interfaces, while maintaining interoperability with the managed C# agents (SpyAgent, InterpreterAgent, DiagnosisAgent, etc.) that implement the core testing logic. This architectural design eliminates the need for complex inter-process communication or API wrappers, allowing developers to implement the critical `UpdateData()` and `EvaluateObservation()` functions using native C++ performance for computationally intensive operations (e.g., real-time computer vision, signal processing, hardware communication) while leveraging the .NET ecosystem for high-level orchestration, data models, or business logic. For VRIS integration specifically, this means that resource-intensive virtual reality data processing, such as continuous hand tracking, spatial collision detection, or gesture recognition, can be implemented in high-performance C++/CLI Spy Objects if required.

Fig. 8. Magnetic gasket extraction in an actual disassembly plant. The worker uses a screwdriver to pry the gasket slightly and pull it out.

5. Experiments

5.1. Task description

To assess the capability of our approach, we relied on an actual simplified scenario: removing magnetic gaskets from refrigerator doors. This task was previously performed entirely manually. The workers used a screwdriver to pry up one corner of the gasket and remove it (see Fig. 8). However, continuous non-ergonomic postures could lead to musculoskeletal disorders in the long run. Hence, it was decided to automate this task. In this context, a collaborative workspace was proposed.

Disassembly begins when the refrigerator is placed on the conveyor belt, initiating movement along the disassembly line. The conveyor transports the refrigerator until a laser sensor detects it. Upon detection, the conveyor belt stops, and the hydraulic pistons activate, pushing the refrigerator against the belt wall to align and secure its position. The worker opens the refrigerator door at this stage, enabling access to the internal components. The robot leverages computer vision capabilities to identify the location of the magnetic gasket within the refrigerator door. Using this visual information, the robot approaches the gasket and executes the gasket removal task. For this proof-of-concept, the magnetic gasket is represented by two rigid parts: a slotted base fixed to the refrigerator door and a gasket inserted into the corresponding slot. While the robot focuses on removing the gasket, the human collaborator concurrently removes drawers and shelves from within the household appliance. The human collaborator closes the refrigerator door once the robot and worker have completed their corresponding tasks. Following the closure, the refrigerator resumes movement along the conveyor belt to the next station in the disassembly line. Fig. 1b shows the actual disassembly line in the simulation featuring the integration of the collaborative robot.

Table 1
Test cases definition.

Number	Test case	Description
1	Conveyor belt pistons failure	The pistons of the conveyor belt are not activated. Therefore, the refrigerator is not aligned and fixed for disassembly.
2	Refrigerator door opening failure	The human worker does not open the refrigerator door.
3	Gasket extraction failure	The robot does not extract the gasket correctly.
4	Gasket extraction direction failure	The robot disassembles the gasket to the human's side.
5	Safety distance failure	The robot not only disassembles the gasket to the human's side but also exceeds the minimum distance threshold established of 0.25 m.

During the execution of the robotic task, once the robot has successfully identified and securely grasped the magnetic gasket, the robot applies a control policy acquired through RL techniques. This policy allows the robot to generalize the extraction process to gaskets with different tolerances and rotational configurations. Furthermore, the RL policy also performs real-time collision avoidance for safety reasons. This functionality allows the robot to detect potential collisions and promptly modify the direction of the gasket extraction, diverting it to the side opposite the worker's location. By dynamically adapting its actions, the robot mitigates the risk of accidental contact or interference with the human collaborator, thus maintaining a safe and seamless work environment (Elguea-Aguinaco et al., 2022).

5.2. Test cases generation

Six test cases were designed to evaluate the system's end-to-end functionality based on the described use case. These test cases encompass a range of scenarios and interactions representative of the system's intended use and are presented in Table 1.

5.3. HRI environment testing

Six task models were devised to diagnose and render assessments for the tests defined in Table 1, each tailored to address the six distinct testing situations. These task models share common interpretation and observation models, the details of which are elaborated upon in subsequent subsections.

5.3.1. Observation model

The observation model defines the observations needed as input for the interpretation and task models. The observer agent contains an implementation that reads the observation model. This agent generates the required observations based on the data collected from the VRIS in each simulation cycle.

For this proof-of-concept, the observation model encompasses seven observations. For a breakdown of these observations, their respective properties, and where they are obtained from, refer to Table 2.

5.3.2. Interpretation model

Table 3 defines and describes the steps that compose the interpretation model for the current use case. It is worth noting that some steps must be defined because they need to be diagnosed. Other steps can be defined as part of the definition of other steps. For such cases, constraints use the "STEPS::" keyword as part of their syntax. Table 4 depicts an example of such a situation, where the *Disassembling* step within the *Running* situation needs the *OpenFridge* step to be interpreted for its interpretation.

5.3.3. Task model

Table 4 depicts the solutions for each situation, namely the *Running* and *Disassembly* situations. Each table row refers to the solution of a specific test. Each solution defines how all the situations and their corresponding steps should be solved, and each step can be composed of one or more conditions.

5.4. Test results

To prove the efficacy of the proposed framework, an evaluation was performed involving correct and erroneous executions of test cases.

To illustrate the diagnostic capabilities inherent to the ULISES framework during simulation execution and the subsequent generation of test results, Test Cases 2 and 4 have been selected as illustrative examples. Figs. 9 and 10, respectively, depict these test cases when executed incorrectly and correctly.

Test Case 2 evaluates whether the human opens the refrigerator door before the disassembly process begins. In Fig. 9a, it can be observed that the refrigerator door remains closed throughout the entire simulation. As a result, the diagnosis output classifies the *Disassembling* step as FALSE, indicated by the red bar in Fig. 9b. Consequently, the ULISES Diagnosis Subsystem generates a corresponding test result indicating that this simulation instance fails to meet the criteria defined in Test Case 2. In contrast, when the refrigerator door is opened (see Fig. 9c), the *Disassembly* situation is represented by a blue line, clearly labeled on the left panel and aligned vertically with the corresponding bar. Additionally, two green bars indicate the execution of the *Disassembling* and *OpenFridge* steps. As shown in Fig. 9d, the *OpenFridge* step begins after the *Disassembling* step has started. According to the specifications of Test Case 2, which require that the refrigerator door be opened after the disassembly phase has begun, the test result is considered successful.

Test Case 4 evaluates whether the gasket is extracted toward the human or in the opposite direction. To reduce computational overhead, most of the surrounding environment elements were hidden during this test case. In Fig. 10a, the robot extracts the part toward the human. As a result, the *ExtractGasket* step is classified as FALSE, as indicated in Fig. 10b. In contrast, in Fig. 10c, the robot extracts the part away from the human, thereby avoiding a potential collision. In this case, the *Running* situation is represented by a blue bar, and the *ExtractGasket* step appears in green (see Fig. 10d), indicating a successful test outcome.

As shown in Fig. 4, after the determination of test pass/fail status by the Diagnosis Subsystem, it proceeds to generate a JSON message encapsulating the test result. This message is subsequently transmitted to the test controller.

The following text details the test result generated by the Diagnosis Subsystem pertaining to Test Case 2, specifically, the simulation wherein the refrigerator door remains unopened:

"Timestamp":"1694187242", "EvalResult":"False", "StepName":
"Assembling", "SituationName":"Running", "CondName":"Fridge door".

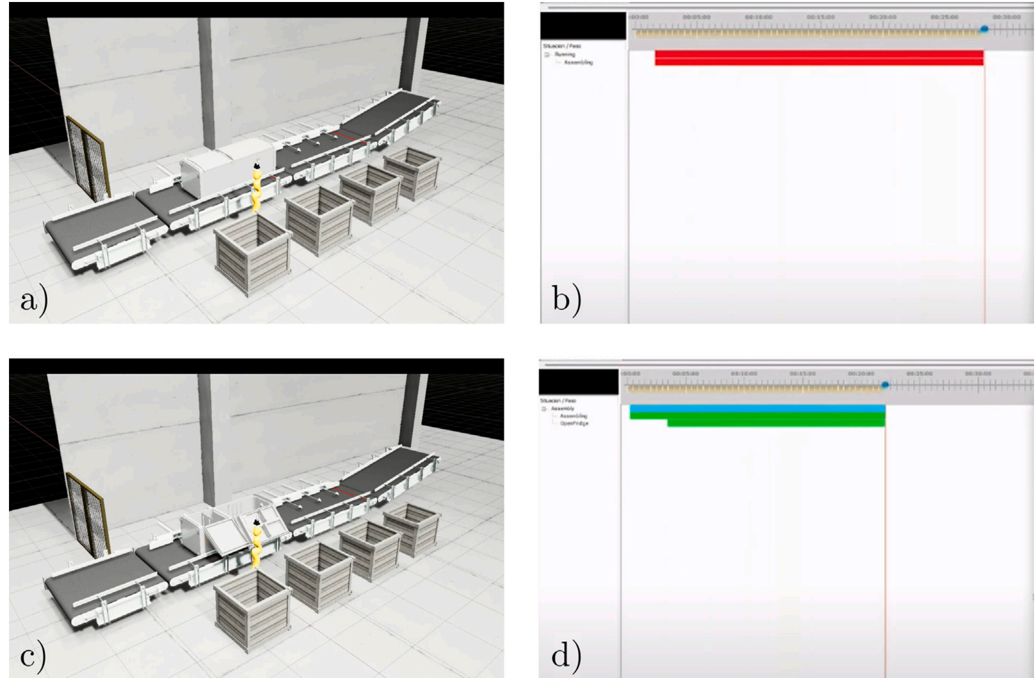

Fig. 9. (a) Incorrect representation of Test Case 2 where the refrigerator door is not open. (b) Diagnosis generated by ULISES for Test Case 2, where the step *Disassembling* within the situation *Disassembly* is classified as FALSE. (c) Correct representation of Test Case 2 where the refrigerator door is open. (d) Diagnosis generated by ULISES for Test Case 2, identifying the steps *Disassembling* and *OpenFridge* as correct.

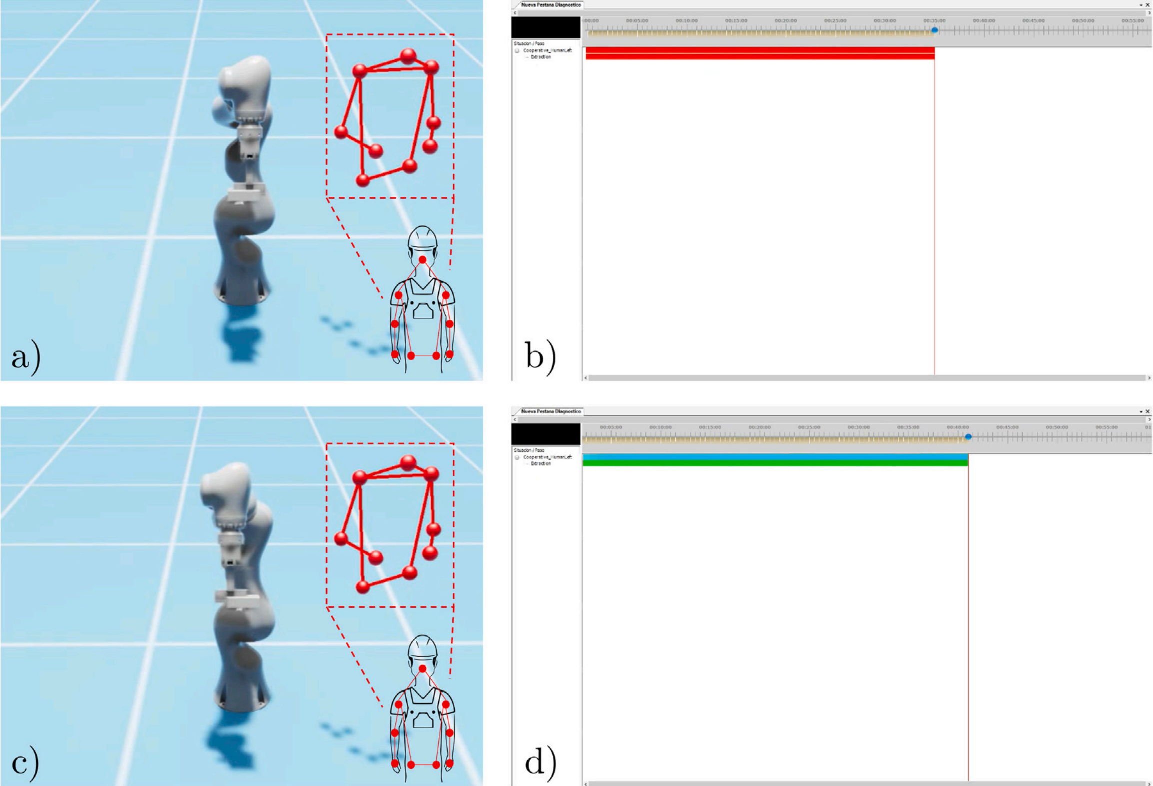

Fig. 10. (a) Incorrect representation of Test Case 4 where the robot extracts the gasket toward the same side of the worker. In the simulation, the human's upper body is represented using skeleton tracking data from a real worker in the disassembly plant. (b) Diagnosis generated by ULISES for Test Case 4, where the step *ExtractGasket* within the situation *Running* is classified as FALSE. (c) Correct representation of Test Case 4 where the refrigerator extracts the gasket toward the opposite side of the worker, avoiding a potential collision. (d) Diagnosis generated by ULISES for Test Case 4, where the step *ExtractGasket* is classified as correct.

Table 2
Observations and properties defined in the observation model.

Observation	Properties	Description
Fridge	conveyor_00_ joint_command upper_joint_position	This observation parameterizes the data acquired from the refrigerator during the simulation. Specifically, the "upper_joint_position" property has been integrated, manifesting as an integer that denotes the state of the upper joint of the refrigerator, which can be categorized as either 'closed', 'open', or 'opening.'
Conveyor 1	laser_detection joint_piston_1_state	The conveyor observation encapsulates observable data from conveyor number 1 (there are three conveyors in the simulation). The "laser_detection" property within this observation identifies whether the laser is actively detecting the refrigerator. At the same time, the "joint_piston_1_state" characterizes the status of a specific joint within the conveyor system. This joint state information indicates whether the conveyor is in motion or stationary.
Robot	1)finger_joint_ revolute_state 2)finger_pos_x 3) MoveLeft 4)MoveRight	This observation is used to monitor the status of the robot. It uses the "finger_joint_revolute_state" property for the gripper's condition, distinguishing between an open or closed state. The "finger_pos_x" attribute captures the x-coordinate values of the gripper's position. Related to this latter property are the "MoveLeft" and "MoveRight" properties, both of which are boolean variables. These properties are contingent on the direction of gripper movement; when the gripper moves left, "MoveLeft" is set to 1, while "MoveRight" is set to 1 when the gripper moves in the right direction.
Gasket	distance_to_ fridge_increasing	This observation monitors the gasket and incorporates the "distance_to_fridge_increasing" boolean property. This property assumes a value of 1 when there is an increase in the distance between the refrigerator and the gasket.
Hip LeftHand RightHand	distance_to_robot	These observations pertain to the states of the operator's hip, left, and right hand. Each of these observations incorporates a property that quantifies the distance between the respective human joint and the robot.

Table 3
Definition of the steps that compose the interpretation model using the constraint-based modeling techniques.

Step(s)	Observation(s)	Instance(s)	Constraint(s)
OpenFridge	Fridge	f	*General*: f_upper_ joint_position > 3.0
Description: This step is active while the fridge door is open.			
Disassembling	Conveyor 1	c	*General*: c_laser_ Detection > 0.5
Description: This step is activated when the fridge arrives to conveyor's end, where it is detected by the conveyor's laser.			
DisassemblyFinished	Conveyor 1	c	*(1) Start*: c_laser_ Detection == 1.0 *(2) General*: c_laser_ Detection == 0.0
Description: The step starts when the disassembly process is finished. The fridge has been in the disassembly position, and afterwards, the conveyor brings it to the next conveyor.			
ExtractGasket	(1) Robot (2) Gasket	(1) r (2) g	*(1) Start*: c_laser_ Detection == 1.0 *(2) General*: c_laser_ Detection == 0.0
Description: This step is active while the gasket is being extracted. It begins once the robot grabs the gasket.			

Table 4
Situations and the implemented solutions for each of the steps in the different situations.

Applies to test number	Situations: Steps	Conditions	Observations: Instances	Constraints
	Disassembly: Disassembling	Pistons opening	Conveyor 1: c	c.joint_piston_1_state == 1.0
Description: Throughout the disassembly procedure, the refrigerator needs to be secured by the conveyor's piston..				
2	Running: Disassembling	Fridge door	STEPS:: Disassembling: afSTEPS:: OpenFridge: of	Duration(of.Start, af.Start) < 3000
Description: In order to carry out the process properly, it is necessary to open the refrigerator once it is in the disassembling position. If the door is opened during any other moment, the test will not be passed.				
3	Disassembly: ExtractGasket	Sequence	STEPS:: OpenFridge: of STEPS:: ExtractGasket: eg	of[Contains] eg
Description: The robot should only try to extract the gasket only if the refrigerator's door has been opened. During the gasket extraction, the refrigerator door must remain opened. This constraint can be represented by the contains operator, which indicates that the starting moment of OpenFridge is smaller than the starting moment of ExtractGasket, and the ending moment of OpenFridge is greater than the ending moment of ExtractGasket.				
3	Running: DisassemblyFinished	Sequence	STEPS:: ExtractGasket: eg STEPS:: DisassemblyFinished: af	Duration(af.End, eg.Start) < 1000
Description: The aim of this condition is to test that the gasket extraction has been carried out before the refrigerator leaves the disassembly position and is carried to the next conveyor.				
4	Running: ExtractGasket	Safety	(1) Robot: r (2) Hip: h (3) RobotMoveLeft: ml (4) RobotMoveRight: mr	((h.x<r.finger_pos_x) AND Duration(ml) < 500) OR (hip.x>r.finger_pos_x) AND Duration(mr) < 500))
Description: This condition tests whether the robot is extracting the gasket in the right direction. If the worker is positioned on the left side of the robot, it should extract the gasket in the right direction and vice versa.				
5	Disassembly: OpenFridge	Safety distance	(1) Hip: h (2) LeftHand: lh (3) RightHand: rh	(h.DistanceToRobot > 0.25) AND (lh.DistanceToRobot > 0.25) AND (rh.DistanceToRobot > 0.25)
Description: Once the disassembling process has started, the safety distance between the robot and the operator must be maintained.				

Upon receipt of the message mentioned earlier, the test controller generates an XML-based Test Report file, containing the corresponding test result. This report format aligns with the compatibility requirements of Google Test (also known as gtest), a unit testing library developed by Google.

The simulation and experimental results are available at https://www.youtube.com/watch?v=W2QP4UkZ1js.

6. Discussion

Recent advancements in automated systems have underscored the need for developing more comprehensive testing mechanisms to assess technical and safety functionalities of these types of systems. In this regard, a discernible trend is the reduction of labor costs through the automation of end-to-end testing procedures (Gamido and Gamido, 2019).

Nevertheless, in ill-defined domains like HRI environments, where unpredictability arises from both human actions and intelligent decision-making agents, existing testing methodologies exhibit inherent constraints (Riccio et al., 2020). These limitations are particularly evident in conventional approaches like probabilistic model checking, which primarily caters to purely probabilistic systems devoid of non-deterministic elements (Legay et al., 2010).

In this context, this paper introduces a specialized framework designed for testing ill-defined domains, combining runtime monitoring and test-based methodologies. Although this approach may not provide the same exhaustive assessment as formal verification methods, it holds a distinct advantage over model checking, particularly in scenarios characterized by substantial uncertainty. Rather than attempting to model every conceivable state within the procedural task resolution process, the presented framework adopts an oracle-based approach, allowing a comprehensive diagnosis of all activities within the SUT. To fulfill this objective, the testing mechanism leverages the specifications delineated for elements at the Interpretation and Diagnosis levels. The Interpretation Subsystem excels at grasping the purpose or intentionality behind activities executed by humans or robots by interpreting steps and situations, irrespective of their correctness. Meanwhile, the Diagnosis Subsystem evaluates the correctness of each step within a given situation, offering insights into the context in which these actions occur. Both steps and situations are subject to constraint-based modeling techniques at both levels. These constraints describe intricate temporal relationships, including intervals, logical conditions, property constraints for bounding observation values, and movement-related restrictions.

Regarding the applicability of the testing system presented, it can extend to any VRIS and even to real-world environments. In this context, the foundational architecture of ULISES hinges on the domain-agnostic nature of its three constituent layers: Observation, Interpretation, and Diagnosis. These three subsystems operate independently

from the HRI environments and offer the versatility to represent any procedural task, ensuring seamless integration within a specific domain through the specification of observation, interpretation, and task models. In fact, the ULISES framework has already been integrated with several VRISs, such as a gardening simulator (Ostiategui et al., 2010), a truck simulator (Rodero, 2009), and a mixed-reality scenario for practicing tennis serves (Aguirre et al., 2014), demonstrating its potential across different domains. This last use case showcased the capabilities of the ULISES framework to diagnose different features of motor skills, such as trajectories or synchronization, besides the procedural task. Two demonstrations of these functionalities can be found at https://www.youtube.com/watch?v=EMM7VVVN_Ww and https://www.youtube.com/watch?v=Z8L2UDMHEAs.

The integration effort of a new virtual or real HRI environment with the ULISES oracle is also worth mentioning. In this regard, ULISES is accompanied by an authoring tool (presented in Aguirre et al. (2012) to minimize the programming effort: the PATH authoring tool. This tool allows for the definition of interpretation and task models without coding and also provides a debugging interface to test if the models are implemented correctly (9 and 10). Although ULISES is a domain-independent framework, it still requires some programming effort at the Observation level to maintain its flexibility. Since the observations from the environment are specific to each system, the *observer* elements need to be programmed. In this proof-of-concept, the ROSObserverAgent has been the main component programmed to provide the mentioned integration in the use case presented in this paper.

Nonetheless, although the platform provides a no-code authoring tool and debugging capabilities, creating interpretation and task models is not trivial. Testers must reflect on how to define steps and situations at the interpretation level, and extensive debugging is needed to refine constraints, especially if the steps composing the task include diagnosing complex movements.

Lastly, it is worth mentioning that test results are enriched with semantic information. Starting from the observation element, the system is designed for extracting the semantics of the actions performed by humans and robots in the context of procedural tasks. This feature enables the creation of comprehensive test results where the failure of a test is easily understandable.

6.1. Computational considerations

ULISES provides several advantages over other field approaches regarding the ill-defined nature of the VRISs and procedural tasks. When examining conventional model-checking approaches, several limitations become apparent in the context of HRI environments. Traditional model-checking relies on deterministic finite state machines that cannot adequately represent the inherent unpredictability of human behavior and non-deterministic elements present in HRI environments. The computational complexity grows exponentially with the number of variables, resulting in state explosion problems that render analysis infeasible for realistic HRI scenarios. Probabilistic model checking partially addresses these limitations by incorporating stochastic components through probabilistic models. However, this approach still encounters significant challenges when modeling human behavior, often reducing it to simplistic states (e.g., idle or walking) with differential equations describing temporal dynamics. Additionally, probabilistic approaches face difficulties in comparing the expected behavior with the deployed version of the system, particularly in identifying specific reasons for test failures.

ULISES framework overcomes these limitations through its constraint-based diagnostic approach. Unlike formal verification methods that attempt to model all possible states, our framework evaluates the correctness of activities as they occur within the environment. This approach precisely identifies test failures through semantically rich diagnostic results that specify both the violated condition and the situational context in which the failure occurred. The framework's three-level cognitive architecture enables comprehensive diagnosis without the computational overhead associated with state explosion. Moreover, the constraint-based modeling techniques allow the representation of complex temporal relationships, logical conditions, and movement-related restrictions that are essential for evaluating physical interactions but difficult to formalize in conventional verification approaches. This capability is particularly valuable given the current lack of methodologies addressing physical interactions among various agents within testing environments. By focusing on diagnosing actual behavior rather than exhaustively exploring all possible states, ULISES provides a more scalable and practical solution for testing HRI environments characterized by uncertainty and ill-defined domains.

While this work does not include a comprehensive quantitative comparison with probabilistic model checking approaches, such a comparison presents fundamental methodological challenges due to the paradigmatic differences between discrete state-space exploration and continuous constraint-based monitoring. Nevertheless, a theoretical analysis of state complexity will be presented to explain the computational advantages of the proposed approach.

To illustrate these advantages, consider Test Case 3 4, which verifies that the temporal sequence constraint of the gasket extraction must occur entirely within the time period when the refrigerator door is open.

$$OpenFridge[Contains]ExtractGasket \tag{4}$$

For a probabilistic model checking approach using tools like PRISM, this scenario would require modeling several state variables: `FridgeDoorState` (Closed, Opening, Open, Closing = 4 states), `RobotTaskState` (Idle, Approaching, Grasping, Extracting, Completed = 5 states), `TimeStep` for temporal verification (conservatively 100 time units), and `ExtractGasketPhase` (NotStarted, InProgress, Completed = 3 states). The resulting state space would contain 4 × 5 ×100 × 3 = 6000 states minimum. However, proper verification of the temporal `Contains` operator requires exploring all possible interleavings where the constraint could be violated, potentially expanding the state space to 105̂-106̂ states with adequate temporal granularity.

In contrast, ULISES evaluates the same constraint directly on execution traces using a single constraint, which internally performs the following verification:

$$of.StartTime \leq eg.StartTime \text{ AND } of.EndTime \geq eg.EndTime \tag{5}$$

This constraint-based approach requires zero explicit states, performing direct timestamp comparisons on actual observation properties. While probabilistic model checking tools excel at design-time verification of abstract models, ULISES provides runtime monitoring of actual system execution, avoiding state explosion entirely while maintaining the precision of continuous constraint evaluation over real sensor data.

Another feature that is worthy of mention in probabilistic model checking approaches is the fundamental challenge they face regarding HRI environments: the requirement to pre-define probabilistic transition models for human behavior. For example, a PRISM-based verification would necessitate specifying transition probabilities such as "probability 0.7 that the worker opens the fridge within 5 seconds" or "probability 0.3 that the worker moves left when the robot approaches". However, human behavior exhibits significant variability across individuals, skill levels, fatigue states, and environmental contexts, making accurate probability estimation practically infeasible. Moreover, static probabilistic models cannot accommodate unforeseen human actions that fall outside the predefined state space, a critical limitation in ill-defined HRI domains where behavioral unpredictability is inherent. In contrast, ULISES's constraint-based approach eliminates the need for behavioral prediction by directly evaluating whatever actions actually occur during execution. Rather than attempting to model

the probability of specific human behaviors, the framework assesses whether observed actions satisfy safety and correctness constraints in real-time, providing robust testing capabilities that adapt naturally to the inherent unpredictability of human operators without requiring extensive behavioral data collection or model updates.

6.2. Scalability considerations

While the current implementation has been successfully deployed on a single computer with real-time performance and acceptable latency, the observation model complexity demonstrates the substantial data processing requirements inherent in HRI testing environments. As evidenced by the current observation model configuration (MGEP, 2025a), the framework currently monitors 167 properties across 12 distinct observation types, including multiple conveyors, robotic joints (7-DOF IIWA robot plus gripper), human skeletal tracking, and various environmental sensors, all processed simultaneously during test execution.

For industrial scenarios involving multiple robots and human operators, the data volume would scale multiplicatively, necessitating distributed processing capabilities. The FIPA-compliant agent architecture provides a natural scaling path through parallelization: individual observer agents could be dedicated to specific subsystems (e.g., one agent per robot, one per conveyor line, one per human operator), and interpretation and diagnosis agents could also be distributed to handle concurrent test scenarios.

The modular nature of the observation model, where each observation can be processed independently by specialized observer agents, facilitates this distribution strategy. This approach would maintain the real-time performance characteristics while accommodating the increased computational demands of large-scale collaborative environments with multiple concurrent HRI scenarios.

Despite the framework's ability to scale, real-time is a requirement for the oracle to work. The system's temporal constraint networks require immediate evaluation because complex temporal relationships between observations form an interconnected dependency graph where changes propagate through the entire constraint space. Dynamic intervals must be updated continuously as observations maintain evolving temporal properties, including start time, end time, and confirmed-until timestamps, with the system implementing interval enlargement mechanisms to ensure temporal coverage remains current. Furthermore, the temporal nature of constraint evaluation inherently limits the system to real-time processing speed even if the testing is carried out on simulation, as observations must unfold over their actual duration to assess temporal relationships properly. Lastly, as mentioned in 4.4, the system can detect multiple non-fulfillments of different conditions at the same time, but the diagnosis result just stores the information for a single situation, and therefore, for a single VRIS. As soon as a condition or conditions are not met within a situation, the test is terminated to conserve computational resources. Consequently, it is not possible to store diagnosis results for multiple situations. Nevertheless, this could be easily fixed by waiting until the simulation or the real task is finished, which would consume more computational resources.

7. Conclusions and future work

This paper introduced a novel AI-based framework for automated HRI environments testing, specifically addressing the challenges of testing in ill-defined domains. The results demonstrated how the ULISES framework successfully combines runtime monitoring with test-based techniques to provide comprehensive testing capabilities for HRI environments. Through the implementation of a three-level cognitive architecture, observation, interpretation, and diagnosis, the framework enables automated testing without human supervision, effectively eliminating the need for a human-in-the-loop approach.

The proposed framework's efficacy was demonstrated through a case study involving a refrigerator disassembly scenario, in which a robot controlled by an RL policy interacted with a human worker. The framework successfully diagnosed various test cases, including safety violations, equipment failures, and procedural errors, proving its capability to handle the unpredictability inherent in HRI scenarios.

A key advantage of the framework is its ability to work with continuous events rather than just discrete ones, which makes it suitable for evaluating procedural tasks involving complex motor skills. Additionally, the interoperability mechanism implemented through the MQTT protocol facilitates communication between distributed systems, enhancing the framework's versatility across different testing environments.

Future work will focus on extending the framework to handle more complex scenarios and diverse industrial applications. This includes applying the framework to multiple robots simultaneously while still providing detailed error explanations. Additionally, while the current implementation focuses solely on runtime monitoring and constraint-based testing, integrating ULISES with formal verification methods represents a promising direction for enhancing safety guarantees in critical industrial applications. A hybrid approach could leverage formal verification during the design phase to prove safety properties of the robot control system and workspace layout, while ULISES provides runtime monitoring to ensure these properties hold during actual human–robot interaction. For instance, formal methods could verify that robot motion planning algorithms mathematically guarantee collision avoidance within predefined workspace boundaries, while ULISES monitors real-time compliance with safety constraints such as minimum distance thresholds and validates that human behavior remains within the formally verified operational envelope. This complementary approach would combine the exhaustive guarantees of formal verification with the adaptive monitoring capabilities needed for unpredictable human behavior, providing stronger safety assurances for safety-critical applications.

ULISES framework represents a significant step toward comprehensive automated testing of collaborative robotic systems, contributing to both the reliability and safety of HRI in industrial settings.

CRediT authorship contribution statement

Aitor Aguirre-Ortuzar: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Conceptualization. **Íñigo Elguea-Aguinaco:** Writing – review & editing, Writing – original draft, Visualization, Project administration. **Nestor Arana-Arexolaleiba:** Writing – review & editing, Project administration. **Leire Etxeberria-Elorza:** Software. **Joseba A. Agirre-Bastegieta:** Writing – review & editing, Writing – original draft, Supervision, Software, Project administration, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Joseba Agirre Bastegieta reports financial support was provided by H2020-ECSEL Joint Undertaking project no. 876852. Aitor Aguirre-Ortuzar reports financial support was provided by Department of Education, Universities and Research of the Basque Government. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially supported by the Department of Education, Universities and Research of the Basque Government through funding

of the Intelligent Systems for Industrial Systems research group of Mondragon Unibertsitatea (IT1676-22). Additional funding was provided by the H2020-ECSEL Joint Undertaking project no. 876852 "Verification and Validation of Automated Systems' Safety and Security - VALU3S" and the Basque Government Department of Economic Development, Sustainability and Environment through the Bikaintek 2020 program.

Data availability

Data will be made available on request.

References

Aguirre, A., Lozano-Rodero, A., Matey, L.M., Villamañe, M., Ferrero, B., 2014. A novel approach to diagnosing motor skills. IEEE Trans. Learn. Technol. 7 (4), 304–318.

Aguirre, A., Lozano-Rodero, A., Villamañe, M., Ferrero, B., Matey, L.M., 2012. OLYMPUS: An intelligent interactive learning platform for procedural tasks. In: GRAPP/IVAPP. pp. 543–550.

Allen, J.F., Ferguson, G., 1994. Actions and events in interval temporal logic. J. Logic Comput. 4 (5), 531–579.

Araiza-Illan, D., Pipe, A.G., Eder, K., 2016a. Intelligent agent-based stimulation for testing robotic software in human-robot interactions. In: Proceedings of the 3rd Workshop on Model-Driven Robot Software Engineering. pp. 9–16.

Araiza-Illan, D., Western, D., Pipe, A., Eder, K., 2015. Coverage-driven verification— an approach to verify code for robots that directly interact with humans. In: Hardware and Software: Verification and Testing: 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings 11. Springer, pp. 69–84.

Araiza-Illan, D., Western, D., Pipe, A.G., Eder, K., 2016b. Systematic and realistic testing in simulation of control code for robots in collaborative human-robot interactions. In: Towards Autonomous Robotic Systems: 17th Annual Conference, TAROS 2016, Sheffield, UK, June 26–July 1, 2016, Proceedings 17. Springer, pp. 20–32.

Choi, B.J., Park, J., Park, C.H., 2021. Formal verification for human-robot interaction in medical environments. In: Companion of the 2021 ACM/IEEE International Conference on Human-Robot Interaction. pp. 181–185.

Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R., et al., 2018. Handbook of Model Checking. vol. 10, Springer.

Elguea-Aguinaco, Í.n., Serrano-Muñoz, A., Chrysostomou, D., Inziarte-Hidalgo, I., Bøgh, S., Arana-Arexolaleiba, N., 2022. Goal-conditioned reinforcement learning within a human-robot disassembly environment. Appl. Sci. 12 (22), 11610.

Elguea-Aguinaco, Í.n., Serrano-Muñoz, A., Chrysostomou, D., Inziarte-Hidalgo, I., Bøgh, S., Arana-Arexolaleiba, N., 2023. A review on reinforcement learning for contact-rich robotic manipulation tasks. Robot. Comput.-Integr. Manuf. 81, 102517.

Farulla, G.A., Lamprecht, A.-L., 2017. Model checking of security properties: a case study on human-robot interaction processes. In: 2017 12th International Conference on Design & Technology of Integrated Systems in Nanoscale Era. DTIS, IEEE, pp. 1–6.

Feldstein, A.C., Glasgow, R.E., 2008. A practical, robust implementation and sustainability model (PRISM) for integrating research findings into practice. Jt. Comm. J. Qual. Patient Saf. 34 (4), 228–243.

Fitting, M., 2012. First-order Logic and Automated Theorem Proving. Springer Science & Business Media.

Foundation, O.S.R., 2019. Rostest - ROS Wiki. Integration test suite based on roslaunch that is compatible with xUnit frameworks, http://wiki.ros.org/rostest.

Gamido, H.V., Gamido, M.V., 2019. Comparative review of the features of automated software testing tools. Int. J. Electr. Comput. Eng. 9 (5), 4473.

Gross, D., Jansen, N., Junges, S., Pérez, G.A., 2022. COOL-MC: a comprehensive tool for reinforcement learning and model checking. In: International Symposium on Dependable Software Engineering: Theories, Tools, and Applications. Springer, pp. 41–49.

Harman, M., McMinn, P., Shahbaz, M., Yoo, S., 2013. A Comprehensive Survey of Trends in Oracles for Software Testing. Tech. Rep. CS-13-01, University of Sheffield, Department of Computer Science, Citeseer.

Holzmann, G.J., Peled, D., 1995. An improvement in formal verification. In: Formal Description Techniques VII: Proceedings of the 7th IFIP WG 6.1 International Conference on Formal Description Techniques. Springer, pp. 197–211.

Huck, T.P., Ledermann, C., Kröger, T., 2020. Simulation-based testing for early safety-validation of robot systems. In: 2020 IEEE Symposium on Product Compliance Engineering-(SPCE Portland). IEEE, pp. 1–6.

Huck, T.P., Selvaraj, Y., Cronrath, C., Ledermann, C., Fabian, M., Lennartson, B., Kröger, T., 2023. Hazard analysis of collaborative automation systems: A two-layer approach based on supervisory control and simulation. In: 2023 IEEE International Conference on Robotics and Automation. ICRA, IEEE, pp. 10560–10566.

Jahangirova, G., 2017. Oracle problem in software testing. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 444–447.

Jiménez-Ramírez, A., Chacón-Montero, J., Wojdynsky, T., Gonzalez Enriquez, J., 2023. Automated testing in robotic process automation projects. J. Softw.: Evol. Process. 35 (3), e2259.

Junges, S., Jansen, N., Katoen, J.-P., Topcu, U., 2016. Probabilistic model checking for complex cognitive tasks–a case study in human-robot interaction. arXiv preprint arXiv:1610.09409.

Kress-Gazit, H., Eder, K., Hoffman, G., Admoni, H., Argall, B., Ehlers, R., Heckman, C., Jansen, N., Knepper, R., Křetínskỳ, J., et al., 2021. Formalizing and guaranteeing human-robot interaction. Commun. ACM 64 (9), 78–84.

Legay, A., Delahaye, B., Bensalem, S., 2010. Statistical model checking: An overview. In: International Conference on Runtime Verification. Springer, pp. 122–135.

Lestingi, L., Askarpour, M., Bersani, M.M., Rossi, M., 2020. Formal verification of human-robot interaction in healthcare scenarios. In: Software Engineering and Formal Methods: 18th International Conference, SEFM 2020, Amsterdam, the Netherlands, September 14–18, 2020, Proceedings 18. Springer, pp. 303–324.

Lestingi, L., Askarpour, M., Bersani, M.M., Rossi, M., 2021. A deployment framework for formally verified human-robot interactions. IEEE Access 9, 136616–136635.

Leucker, M., Schallhart, C., 2009. A brief account of runtime verification. J. Log. Algebr. Program. 78 (5), 293–303.

Li, S., Wang, R., Zheng, P., Wang, L., 2021. Towards proactive human–robot collaboration: A foreseeable cognitive manufacturing paradigm. J. Manuf. Syst. 60, 547–552.

Lin, Y.C., Hung, M.H., Huang, H.C., Chen, C.C., Yang, H.C., Hsieh, Y.S., Cheng, F.T., 2017. Development of advanced manufacturing cloud of things (AMCoT)—A smart manufacturing platform.. IEEE Robot. Autom. Lett. 2 (3), 1809–1816.

Liu, S., Nakajima, S., 2020. Automatic test case and test oracle generation based on functional scenarios in formal specifications for conformance testing. IEEE Trans. Softw. Eng. 48 (2), 691–712.

Lynch, C.F., Ashley, K.D., Aleven, V., Pinkwart, N., 2006. Defining ill-defined domains; a literature survey. In: Intelligent Tutoring Systems (ITS 2006): Workshop on Intelligent Tutoring Systems for Ill-Defined Domains.

MGEP, 2024a. Supplementary material for "novel framework for automated testing of ill-defined human-robot interaction environments": ULISES test utils. http://dx.doi.org/10.5281/zenodo.17218681, Zenodo, URL https://zenodo.org/records/17218681, A ROS Python module to synchronize ros tests executions with external V&V tool ULISES via ROS-MQTT bridge, Zenodo software repository.

MGEP, 2024b. Supplementary material for "novel framework for automated testing of ill-defined human-robot interaction environments": Valu3s UC7. http://dx.doi.org/10.5281/zenodo.17219151, Zenodo, URL https://zenodo.org/records/17219151, UC7 demonstrator shows how to coordinate simulation-based testing activity in human-robot collaborative environments. Provides real-time, automated verdict of test execution of simulation environments through constrained based-oracles using Simulation-Based Testing for Human-Robot Collaboration, Zenodo software repository.

MGEP, 2025a. Supplementary material for "novel framework for automated testing of ill-defined human- robot interaction environments": Observation model. http://dx.doi.org/10.5281/zenodo.16739371, Zenodo, URL https://doi.org/10.5281/zenodo.16739371, XML file containing the observation model for the use case of the paper "Novel Framework for Automated Testing of Ill-Defined Human- Robot Interaction Environments".

MGEP, 2025b. Supplementary material for "novel framework for automated testing of ill-defined human-robot interaction environments": Complete sequence diagrams and technical documentation. http://dx.doi.org/10.5281/zenodo.16739244, Zenodo, URL https://doi.org/10.5281/zenodo.16739244, Diagrams containing architecture overview, interpreter agent, diagnosis agent, and observer agent and VRIS integration specifications.

Ostiategui, F., Amundarain, A., Lozano-Rodero, A., Matey, L., 2010. Gardening work simulation tool in virtual reality for disabled people tutorial. In: Proceedings of Integrated Design and Manufacturing-Virtual Concept. IDMME'10.

Pezze, M., Zhang, C., 2014. Automated test oracles: A survey. In: Advances in Computers. vol. 95, Elsevier, pp. 1–48.

Python Software Foundation, 2025. unittest — Unit Testing Framework. Python Software Foundation, URL https://docs.python.org/3/library/unittest.html, Python 3.13.5 documentation.

Riccio, V., Jahangirova, G., Stocco, A., Humbatova, N., Weiss, M., Tonella, P., 2020. Testing machine learning based systems: a systematic mapping. Empir. Softw. Eng. 25, 5193–5254.

Rodero, A.L., 2009. Metodología de desarrollo de sistemas interactivos inteligentes de ayuda al aprendizaje de tareas procedimentales basados en realidad virtual y mixta (Ph.D. thesis). Universidad de Navarra.

Rubagotti, M., Tusseyeva, I., Baltabayeva, S., Summers, D., Sandygulova, A., 2022. Perceived safety in physical human–robot interaction—A survey. Robot. Auton. Syst. 151, 104047.

Rzig, D.E., Iqbal, N., Attisano, I., Qin, X., Hassan, F., 2022. Characterizing virtual reality software testing. arXiv preprint arXiv:2211.01992.

Van Wesel, P., Goodloe, A.E., 2017. Challenges in the Verification of Reinforcement Learning Algorithms. Tech. Rep..

Wang, K.-J., Lin, C.J., Tadesse, A.A., Woldegiorgis, B.H., 2023. Modeling of human–robot collaboration for flexible assembly—a hidden semi-Markov-based simulation approach. Int. J. Adv. Manuf. Technol. 126 (11), 5371–5389.

Webster, M., Western, D., Araiza-Illan, D., Dixon, C., Eder, K., Fisher, M., Pipe, A.G., 2020. A corroborative approach to verification and validation of human–robot teams. Int. J. Robot. Res. 39 (1), 73–99.

Zhao, Y., Serebrenik, A., Zhou, Y., Filkov, V., Vasilescu, B., 2017. The impact of continuous integration on other software development practices: a large-scale empirical study. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 60–71.