robotics

MDPI

Article

RV4JaCa—Towards Runtime Verification of Multi-Agent Systems and Robotic Applications

Debora C. Engelmann [1,2], Angelo Ferrando [2,*], Alison R. Panisson [3], Davide Ancona [2], Rafael H. Bordini [1] and Viviana Mascardi [2]

[1] School of Technology, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre 90619-900, Brazil
[2] Department of Informatics, Bioengineering, Robotics and Systems Engineering, University of Genoa, 16145 Genova, Italy
[3] Department of Computing, Federal University of Santa Catarina, Araranguá 88906-072, Brazil
* Correspondence: angelo.ferrando@unige.it

Abstract: This paper presents a Runtime Verification (RV) approach for Multi-Agent Systems (MAS) using the JaCaMo framework. Our objective is to bring a layer of security to the MAS. This is achieved keeping in mind possible safety-critical uses of the MAS, such as robotic applications. This layer is capable of controlling events during the execution of the system without needing a specific implementation in the behaviour of each agent to recognise the events. In this paper, we mainly focus on MAS when used in the context of hybrid intelligence. This use requires communication between software agents and human beings. In some cases, communication takes place via natural language dialogues. However, this kind of communication brings us to a concern related to controlling the flow of dialogue so that agents can prevent any change in the topic of discussion that could impair their reasoning. The latter may be a problem and undermine the development of the software agents. In this paper, we tackle this problem by proposing and demonstrating the implementation of a framework that aims to control the dialogue flow in a MAS; especially when the MAS communicates with the user through natural language to aid decision-making in a hospital bed allocation scenario.

Keywords: runtime verification; multi-agent systems; robotic applications; JaCaMo framework; explainable artificial intelligence; dialogue systems

Citation: Engelmann, D.C.; Ferrando, A.; Panisson, A.R.; Ancona, D.; Bordini, R.H.; Mascardi, V. RV4JaCa—Towards Runtime Verification of Multi-Agent Systems and Robotic Applications. *Robotics* **2023**, *12*, 49. https://doi.org/10.3390/robotics12020049

Academic Editor: Charalampos Bechlioulis

Received: 14 February 2023
Revised: 13 March 2023
Accepted: 21 March 2023
Published: 24 March 2023

1. Introduction

A characteristic identified as essential in Artificial Intelligence (AI) is explainability, as it provides users with the necessary inputs to make it possible to understand the system's behaviour and inspire confidence in the final outcome. Explainability becomes an essential feature in Multi-Agent Systems (MAS), as it is one of the most powerful paradigms for implementing complex distributed systems powered by artificial intelligence techniques. MAS are built upon core concepts such as distribution, reactivity, and individual rationality. Agents have been widely studied, and an extensive range of tools have been developed, such as agent-oriented programming languages and methodologies [1]. Thus, practical applications of multi-agent technologies have become a reality for solving complex and distributed problems [2], such as robotic applications. In addition, it also allows the execution of various tasks and makes integration with various technologies possible.

Even though, in this paper, we mainly focus on MAS and how to increase their reliability, we keep in mind robotic applications as the target of our MAS development. In some sense, we consider the agent part as the soul of the robot, i.e., its rational part. Consequently, we show how by increasing the agents' reliability, we may increase the robotic target application's reliability as well.

This is not the first time agents and robots are combined to tackle complex problems; indeed, the design of robotic applications is known to benefit from multi-agent solutions [3].

This is mainly determined by the fact that robots are no longer exclusively used in industrial applications, where they operate in highly controllable and predictable environments. Instead, they are used in an increasing number of domains, where the environment is often unpredictable, and agents can have unexpected behaviours. For example, in an emergency search and rescue scenario the environment in which the robots operate is unpredictable: the structure of the buildings where robots are deployed may not be known in advance, and humans can have unpredictable reactions in emergencies. Robots in these applications often benefit from (or require) some level (semi or full) of autonomy. In addition, the missions the robots need to achieve are more and more complex and require multiple robots with different capabilities to collaborate. Thus, multi-agent solutions are required.

Even though MAS solutions can be a natural choice for developing complex and distributed systems, similarly to any other software development technique, they are prone to errors and bugs (whether at the implementation or description level). Standard techniques such as testing and debugging can be deployed to tackle this problem. However, in the case of MAS, the process of testing [4], debugging [5], and verifying [6] such systems can be quite complex. For this and other reasons, more lightweight approaches to guarantee the correct execution of the system are valuable. One technique that can be applied in such cases is Runtime Verification (RV) [7,8]. Differently from other verification techniques, RV is lightweight because it only concerns the analysis of the runtime execution of the system under analysis, which makes RV very similar to testing. However, rather than testing, RV is based on a specification formalism, as it happens in formal verification, to express the properties to be checked against the system's behaviour, and is particularly suitable for monitoring control-oriented properties [9].

In this paper, we present RV4JaCa, an approach to perform the RV of multi-agent systems developed using the JaCaMo framework [10]. RV4JaCa is obtained by extending MASs implemented in JaCaMo with a monitoring feature. In a nutshell, RV4JaCa handles all the engineering pipelines to introduce RV in JaCaMo and to extract and check runs of the MAS against formal properties. In particular, RV4JaCa enables RV of agents' interactions (i.e., messages). As a proof of concept, we demonstrate a robotic case study in the hospital bed allocation domain where RV is used to verify two different Agent Interaction Protocols.

Note that no other RV framework has ever been integrated into JaCaMo. As we point out in Section 5, some partial solutions exist, but no real integration through artefacts has ever been achieved—not in the robotic scenario nor the bed allocation one. As it will become clear in the course of the paper, the integration of RV in JaCaMo is relevant and useful because it allows the developer to better abstract the agents' development. Thanks to the presence of monitors checking whether the agents and users follow a predetermined communication protocol, the agents' logic can be simplified. Since it is the job of the monitors to check for protocol violations, the agents can simply focus on their reasoning and on handling their own tasks. Moreover, thanks to the presence of an agent that serves as the monitor's counterpart in the MAS, the agents can also exploit additional information about protocol violations. Naturally, what we present in this work could have been obtained without using an RV framework in JaCaMo. However, this would have required much more work on the development of the agents; while, with an external RV framework, it is easier and more practical to detach the agents' tasks from the system's requirements (the agents can just focus on their own objectives, and it is up to the RV layer to guarantee the preservation of the communication protocols).

This paper is an extended and improved version of the conference paper [11].

The paper is structured as follows. Section 2 reports the background needed to fully understand the contribution; here, the JaCaMo framework and the notion of RV are introduced. Section 3 presents RV4JaCa, while Section 4 presents its application in a bed allocation case study (enhanced with robotic components with respect to [11]). Section 5 positions the contribution with respect to the state of the art. Finally, Section 6 summarises the contribution's results and points out future developments.

2. Background

2.1. Multi-Agent Systems

Multi-Agent Systems (MAS) are systems composed of multiple agents. They seem to be a natural metaphor for building and understanding a wide range of artificial social systems and can be applied in several different domains [12]. There are two interlocking strands of work in multi-agent systems: the one that concerns individual agents and the one that concerns itself with the collections of these agents. In practice, agents rarely act alone. They usually inhabit an environment that contains other agents. Each agent can control, or partially control, parts of the environment, which is called its "sphere of influence". It may happen that these spheres of influence overlap. Hence, (parts of) the environment may be controlled jointly by more than one agent. In this case, to achieve the desired result, an agent must also consider how other agents may act. These agents will have some knowledge, possibly incomplete, about the other agents [13].

Wooldridge [12] believes that to be able to understand a multi-agent domain, it is essential to understand the type of interaction that occurs between agents. For intelligent autonomous agents, the ability to reach agreements is extremely important, and for this, negotiation and argumentation skills are often necessary. In [14], the authors mention two types of studies in multi-agent systems; the first one is Agent-Centred Multi-Agent Systems (ACMAS), which study the states at the level of an agent and the relationship between those states and their general behaviour, which are then projected in terms of the agent's states of mind. The second one is Organisation-Centred Multi-Agent Systems (OCMAS), which are systems whose foundations reside in the concepts of organisations, groups, communities, roles, and functions, among others. An OCMAS is not considered in terms of mental states but in capacities and constraints, which are considered organisational concepts, as well as functions, tasks, groups, and interaction protocols.

In a multi-agent system, the organisation is the collection of roles, relationships, and authority structures that govern agents' behaviour. Every multi-agent system has some form of organisation, even if it is implicit and informal. Organisations guide the mode of interaction between agents, which may influence data flows, resource allocation, authority relationships, and various other features of the system [15].

In [16], the authors argue that systems that heavily adopt AI techniques are increasingly available, and making them explainable is a priority. Explainable Artificial Intelligence (XAI) is a research field that aims "to make AI systems results more understandable to humans" [17]. These results must be clear (in non-technical terms) and provide justifications about decisions made [18]. In order to achieve a satisfactory level of explainability in multi-agent systems, much communication between agents and humans needs to be performed. However, this can generate an additional point of failure in the execution of the system since if, for example, the predefined communication protocol is not followed, or even if the human decides to change the conversation topic unexpectedly, this can lead to unexpected and probably inappropriate behaviour of the system.

2.2. JaCaMo Framework

JaCaMo is a framework that allows for multi-agent oriented programming. This framework consists of the integration of three previously existing platforms: Jason—for programming autonomous agents, CArtAgO—for programming environmental artefacts, and Moise—for programming multi-agent organisations [10]. A multi-agent system programmed in JaCaMo has Jason agents that are organised and follow roles according to Moise's hierarchical structure. These agents work in environments based on distributed artefacts programmed using CArtAgO. Figure 1 shows an overview of the three dimensions of JaCaMo.

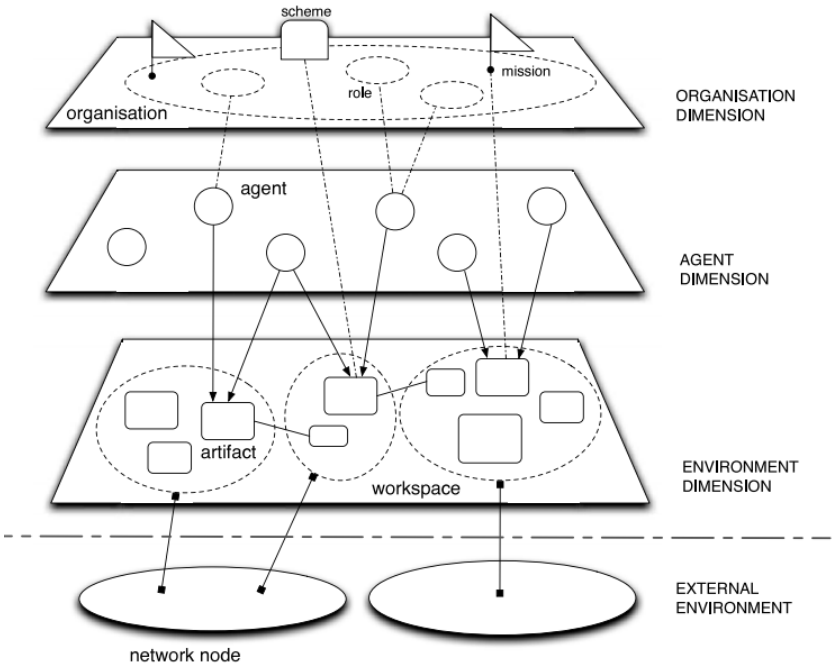

Figure 1. Overview of the three dimensions of JaCaMo [10]. Reproduced with permission from authors, Science of Computer Programming, Volume 78, Issue 6; published by Elsevier, 2013.

Jason (`Agent dimension`) is an agent-oriented programming language that is an interpreter for an extended version of the AgentSpeak [13] language. Agents programmed in Jason use the Belief–Desire–Intention (BDI) model [19]. The main idea of this approach is to model the process of deciding which action to take to achieve certain objectives [19]. Moise [10,20] is related to the `organisation dimension`, where agents can be part of groups and follow specific roles. Furthermore, with Moise, *schemes* are defined, that is, the structure of organisational goals is decomposed into sub-goals and grouped into missions. An organisation is specified in an XML (Extensible Markup Language) file. CArtAgO [21], the `environment dimension`, is used to simulate an environment or interface with a real one; this is where *artefacts* are defined. These artefacts define the environment's structure and behaviour, representing all resources that agents need. Agents can discover, create, and use artefacts during the runtime [10]. Artefacts are programmed in Java. Combining these dimensions provides us with a complete framework for developing multi-agent systems through agents, organisations, and environments.

Naturally, JaCaMo is not the only available option for developing MAS solutions (see [22] for a thorough literature review on agent-based programming techniques). However, JaCaMo is one of the most used frameworks for the development of MAS since it has a large community, and it allows defining the agents in a symbolic way. This last aspect is of paramount importance to achieve a high level of explainability (increasingly expected in robotic applications as well).

2.3. Runtime Verification

Runtime Verification (RV) [7] is a kind of formal verification technique that focuses on checking the behaviour of software/hardware systems. With respect to other formal verification techniques, such as Model Checking [23] and Theorem Provers [24], RV is considered more dynamic and lightweight. This is mainly due to its being completely focused on checking how the system behaves while the latter is currently running. This is important from a complexity perspective. RV does not need to simulate the system in order to check all possible execution scenarios; however, it only analyses what the system produces (i.e., everything that can be observed in the system). This is usually obtained through monitors, which are automatically generated from the specifications of the properties to be checked,

and are nothing more than validation engines, which, given a trace of events generated by the system execution, conclude the satisfaction (resp. violation) of the corresponding properties. In turn, a formal property is the formal representation of how we expect the system should behave. The monitor's job is to verify during the runtime whether such a property holds.

Since monitors are usually deployed together with the system under analysis, they are well suited for checking properties that require being continuously monitored. This is especially true in safety-critical scenarios, where a system's fault can cause injuries, loss of money and even deaths. A key example is autonomous and robotic systems, where *reliability* is vital [25], and the addition of monitors ensuring a correct behaviour is a valuable feature.

In the scenario envisaged in this contribution, we aim to use RV as a safety net (in the sense that the RV layer can be seen as a safety layer since any protocol violation is detected by the monitors (and correct behaviour can be enforced by the agents)) for a message exchange in JaCaMo. As pointed out elsewhere, the protocols involved in the communication between agents and human beings can be very complex and hard to track. Moreover, agents are usually particularly focused on the reasoning and reactive aspects, while the consistency of the protocols is taken for granted. However, above all, in the case of human beings in the loop, such an assumption cannot be made. RV is a suitable candidate to keep track of the protocols to check whether the current agents' enactment is consistent (or not) with the expected protocol. Such consistency checking is extremely important in safety-critical scenarios, as in the healthcare domain, where a protocol violation can be costly.

2.4. Runtime Monitoring Language

Runtime Monitoring Language (https://rmlatdibris.github.io/, accessed on 10 February 2023) (RML [26]) is a Domain-Specific Language (DSL) for specifying highly expressive properties in RV (such as non-context-free ones). We choose to use RML in this work because of its support of parametric specifications and its native use for defining interaction protocols. In fact, the low-level language on which RML is based upon was born for specifying communication protocols [27,28].

Since RML is just a means for our purposes, we only provide a simplified and condensed view of its syntax and semantics. However, the complete presentation can be found in [26]. Indeed, other formalisms can be as easily integrated into RV4JaCa.

In RML, a property is expressed as a tuple $\langle t, ETs \rangle$, with t a term and $ETs = \{ET_1, \ldots, ET_n\}$ a set of event types. An event type ET is represented as a set of pairs $\{k_1 : v_1, \ldots, k_n : v_n\}$, where each pair identifies a specific piece of information (k_i) and its value (v_i). An event Ev is denoted as a set of pairs $\{k'_1 : v'_1, \ldots, k'_m : v'_m\}$. Given an event type ET, an event Ev matches ET if $ET \subseteq Ev$, which means $\forall (k_i : v_i) \in ET \cdot \exists (k_j : v_j) \in Ev \cdot k_i = k_j \wedge v_i = v_j$. In other words, an event type ET specifies the requirements that an event Ev has to satisfy to be considered valid.

An RML term t, with t_1, t_2 and t' as other RML terms, can be:

- ET, denoting a set of singleton traces containing the events Ev s.t. $ET \subseteq Ev$;
- $t_1 \;\; t_2$, denoting the sequential composition of two sets of traces;
- $t_1 \mid t_2$, denoting the unordered composition of two sets of traces (also called shuffle or interleaving);
- $t_1 \wedge t_2$, denoting the intersection of two sets of traces;
- $t_1 \vee t_2$, denoting the union of two sets of traces;
- $\{let \;\; x; \;\; t'\}$, denoting the set of traces t' where the variable x can be used (i.e., the variable x can appear in event types in t' and can be unified with values).
- $t'*$, denoting the set of chains of concatenations of traces in t'

Event types can contain variables. For example, $ET(ag1, ag2) = \{sender : ag1, receiver : ag2\}$, where we do not force any specific value for the sender (resp., receiver) of a message (in this case, the events of interest would be messages). This event type matches all events

containing sender and receiver. When an event matches an event type with variables, such as in this case, the variables obtain the values from the event. For instance, if the event observed would be $Ev = \{sender : "Alice", receiver : "Bob"\}$, it would match ET by unifying its variables as follows: $ag1 = "Alice"$, and $ag2 = "Bob"$. This aspect is important because, as we are going to show in the bed allocation domain, we can use variables in RML terms to enforce a specific order of messages. For instance, in this very high-level example, we could say that when a message from $ag1$ to $ag2$ is observed, the only possible consequent message can be a message from $ag2$ to $ag1$. Since the first event has unified the two variables, the second event will have to be a message from *Bob* to *Alice* (otherwise, this would be considered a violation). Naturally, this is only the intuition behind it, but it should help to grasp the expressiveness of RML and how variables can be exploited at the protocol level to enforce specific orders amongst the messages.

3. Engineering Runtime Verification for Multi-Agent Systems

Our approach, named RV4JaCa (the source code and a running example are available at https://github.com/DeboraEngelmann/RV4JaCa, accessed on 10 February 2023), allows the runtime verification of multi-agent systems based on the JaCaMo platform. In Figure 2, we present an overview of the entire approach. RV4JaCa is composed of:

1. A `Sniffer` class, developed in Java, responsible for observing all communication between agents in the MAS.
2. A CArtAgO artefact, named `RV4JaCa Artefact`, responsible for analysing the messages observed by the `Sniffer`, transforming them into a JSON (JavaScript Object Notation) object and sending it as a REST (Representational State Transfer) request to the `Formal Monitor`. Note that RV4JaCa is not in any way limited to a specific kind of monitor; we used an RML monitor in our tests simply because it was the most suitable candidate for specifying the protocols of our interest. Nonetheless, a different monitor could be as easily integrated as the RML one. In addition, when the `RV4JaCa Artefact` receives the response to the request made to the `Formal Monitor` saying that there was a violation, it can add a belief in the `Monitor` agent's belief base.
3. The `Formal Monitor`, responsible for analysing the events sent by `RV4JaCa Artefact` and verifying the satisfaction or violation of a formal property of interest.
4. A `Monitor` agent, which can be added to the system if it is necessary to interfere with agents' behaviour during the runtime. In this case, if there is a violation, the `RV4JaCa Artefact` adds a belief to the `Monitor`'s belief base. When the agent perceives this addition, it can react by sending a message to the interested agents warning about the violation. This may trigger some consequent recovery mechanism, which usually is fully domain dependent. On the other hand, the `Monitor` agent can also perform different activities depending on the system's needs. Listing 1 shows an example of a plan in Jason that the `Monitor` agent can use to react to a belief addition (`+violation`). It informs an interested agent (receiver) that there was a violation in exchanging messages between two agents. In this simple example, the agent uses the internal action `.send` to inform the `Receiver` agent about the violation. Such a send action is used to inform the receiver of the message classified as a violation of the latter. The information passed as parameters are domain specific; indeed, RV4JaCa is mainly focused on checking whether an interaction protocol is violated at runtime or not. How to react and what to do after a violation has been observed is completely domain specific. Listing 1 is an example of an implementation of the monitor agent, which in this case, only propagates the information to the agents' of interest. Naturally, as we point out later on in the paper, more complex behaviour can be implemented, and the monitor agent is the place where the developer can customise it.

Listing 1. An example of how an agent perceives a violation.

```
+violation(Time,Id,IsReply,Performative,Sender,Receiver,Verdict)
<-
    .print("Message ",Id," from ",Sender," to ",Receiver," at ", Time,
                                                    " has a violation.");
    .send(Receiver,assert,violation(Time,Id,IsReply,Performative,
                                            Sender,Receiver,Verdict)).
```

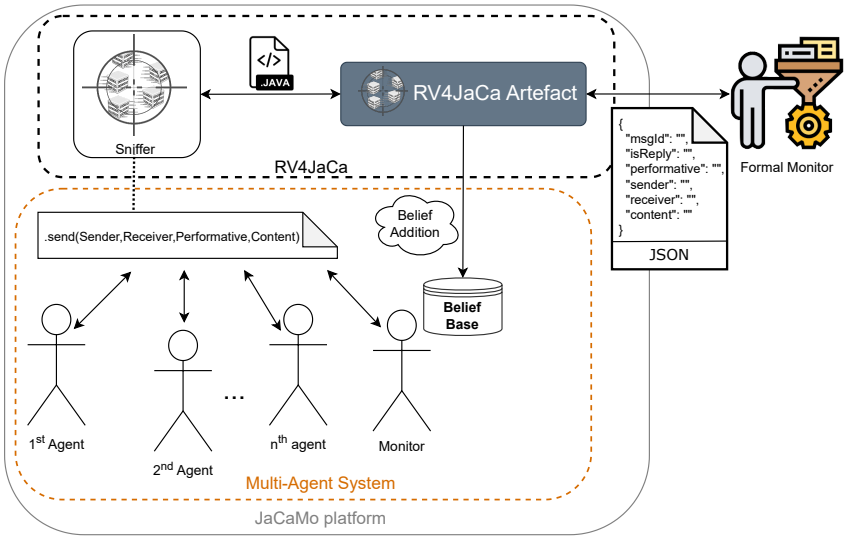

Figure 2. Approach overview for RV in MAS.

Let us now go further into the detail of the engineering aspects of RV4JaCa.

3.1. Agents Instrumentation

The act of instrumentation software systems is common in RV [29] since it allows extracting information about the runtime execution without changing the under-the-hood compiler/interpreter. Through instrumentation, additional instructions can be added to a program (either in its source code or in its executable form, e.g., bytecode in Java [30]).

Since RV does not—usually—require a model of the system, and it exclusively focuses on the observable behaviour of the system under analysis, the act of instrumenting is sufficient to extract all necessary information. In RV4JaCa, the instrumentation step is automatically obtained by adding the `Sniffer` agent to the MAS. Such an agent is available in JaCaMo by design, and it is capable of observing the messages exchanged amongst the agents in the system. Since in RV4JaCa, we are interested in verifying the communication flow amongst the agents, the `Sniffer` agent is the natural candidate to act as an instrumentation step. Thus, thanks to JaCaMo's support by design, to gather all the information we need for the verification, we only need to add the `Sniffer` agent upon the creation of the MAS.

Once the `Sniffer` agent is added to the MAS, all the messages exchanged by the agents can be observed and tracked.

3.2. Monitor Information Exchange

In the previous section, we discussed how the messages can be extracted from the MAS through the `Sniffer` agent. Now, we discuss how such messages are passed to the monitor for performing their verification against a formal specification.

This step is obtained by exploiting CArtAgO artefacts. Specifically, CArtAgO artefacts are implemented in JaCaMo as Java classes. Each artefact can be shared amongst multiple

agents. In the specific case of RV4JaCa, the artefact we use is only accessed by the `Sniffer` class upon detection of a message exchanged between two agents in the system. When a message is passed from the `Sniffer` class to the `RV4JaCa Artefact`, it is translated from its Java representation (internally used in JaCaMo for specifying interactions amongst agents) into a JSON-formatted message. We chose JSON because it is a well-known data interchange format, it is widely used, and it is supported in the majority of programming languages. In this way, RV4JaCa is natively general-purpose since it does not enforce any specific external RV framework to perform the verification of the agents' messages. The only requirement is that the RV technique employed for the verification has to handle JSON messages. However, since JSON is largely supported, this is not a hard requirement to satisfy.

Naturally, the RV software can be deployed as an external component to JaCaMo. Note that this is not in any way a limitation since the RV component could, in theory, be implemented along with the `RV4JaCa Artefact` (straightforwardly in Java). However, since we are interested in reusing existing RV techniques (focusing on one in particular), we assume the RV monitor runs externally to the MAS. The latter aspect is of paramount importance when the MAS is deployed in robotic applications (as in our case) since the robotic system may have limited resources and the possibility of running the additional verification step on a different machine may be necessary.

To communicate with the external RV monitor, the `RV4JaCa Artefact` establishes a TCP connection (over a predetermined URL and port) on which it sends REST requests. Such requests consist of the JSON messages corresponding to the agents' interactions observed by the `Sniffer` agent. For each request, the `RV4JaCa Artefact` expects feedback from the RV monitor. The feedback refers to the current satisfaction, or violation, of the formal specification under analysis. Note that, in this work, as it is common in RV, we mainly focus on safety properties (i.e., properties that can only be violated at runtime), such as "something bad will never happen". Thus, the monitor's feedback can be one of the following outcomes: ? (read "unknown"), and $\perp$ (read "violated"). The former outcome refers to scenarios where the monitor has not yet observed enough information to conclude anything about the satisfaction (resp. violation) of its formal specification. Since we are talking about safety properties, this means that no violation has been observed yet; thus, the MAS is still—for now—behaving as expected (no unexpected message has been observed). The latter outcome refers to scenarios where the monitor has observed a violation. Thus, a message that was not supposed to be observed has been observed. For instance, as we are going to see in our case study, a violation could be an expected change in topic in the communication protocol.

The monitor's outcome is reported at runtime while the system is running. Because of this, it can be used by RV4JaCa to guide the reactive behaviours in the case of protocol violation (as we are going to further explain in the next sections).

Before moving on with the presentation of the verification on the monitor's side, we wish to linger a bit longer on when such verification can be performed. Previously, we exclusively focused on performing RV while the MAS was running. However, the RV step can be performed offline as well (so after the MAS has been executed). In such a case, the `RV4JaCa Artefact`, instead of sending JSON messages as REST requests to an online monitor, logs such messages into a file. Then, after the execution of the system, the log file can be analysed by the monitor as described before, even though the messages are not read from a TCP connection but from the log file previously created. This last aspect is again of paramount importance when the MAS is deployed in scenarios with limited resources (such as in robotics ones). In such scenarios, we can delay the verification until after the system's execution (in this way, avoiding wasting resources at runtime, if not needed).

3.3. Runtime Verification of Agents Interaction Protocols

Once the messages have been gathered by the `Sniffer` agent and propagated through the `RV4JaCa Artefact` to the monitor, the actual verification may happen.

Again, this can happen in different ways; since the RV is external to RV4JaCa, any formalism can be used to describe the properties to verify at runtime. Nonetheless, without a default monitor, RV4JaCa would not be self-contained. Thus, a default monitor for RV4JaCa is given. As previously mentioned, both in Section 2 and at the beginning of Section 3, we exploit RML [26] as the default monitor language for RV4JaCa.

The reasons for choosing RML are various. However, the most relevant one is its natural inclination to be exploited for defining communication protocols. This is mainly due to the fact that RML was born as a formalism to specify Agent Interaction Protocols [28,31,32] and then grew to be used in more general settings.

Once the property of interest has been specified using the RML formalism, the corresponding monitor can be synced. As we showed in Section 2, RML properties are terms that describe the execution traces that are allowed to be observed by running the MAS. From the viewpoint of RV4JaCa, the RML monitor is an external component which communicates through a TCP connection. RML, by default, supports such kind of connection. Thus the resulting RV is obtained by simply running the RML monitor along with the MAS. Every time a message will be observed by the `Sniffer` agent and propagated through the `RV4JaCa Artefact` to the RML monitor, the latter will check such an event against the current state of the RML term. In case the event does not match the current state of the RML term (see Section 2 for more details), the RML monitor returns negative feedback ($\perp$) to the `RV4JaCa Artefact`; otherwise, meaning that the event is acceptable in the current RML state, the RML monitor returns inconclusive feedback (?) to the `RV4JaCa Artefact`. In the second case, a possible reaction inside the MAS is triggered (see next section).

As we are going to see in Section 4, through RML, we can specify Agent Interaction Protocols concerning the expected communication behaviour between human users and agents in a healthcare scenario. This will help us to show the effectiveness of RML and its use in a real-world case study.

3.4. Runtime Enforcement through the `Monitor` Agent

As mentioned in the previous sections, the verification flow concludes with a special agent called `Monitor`. This agent is the one in charge of reacting to the RML monitor's feedback. Specifically, the `Monitor` agent is the single agent with access to the `RV4JaCa Artefact`. However, differently from the `Sniffer` agent, the `Monitor` agent does not push events (the messages) to be sent to the RML monitor but instead subscribes to the `RV4JaCa Artefact` to be updated upon the RML monitor's feedback reception.

When the RML monitor's feedback is received, the `RV4JaCa Artefact` informs the `Monitor` agent about it. This is obtained by updating the `Monitor`'s agent belief base. When this happens, the `Monitor` agent checks the latest feedback (i.e., the current verdict returned by the RML monitor) and decides how to proceed. Now, there are two viable options for the agent. First, if the feedback is inconclusive (?), then the `Monitor` agent does nothing. This is the scenario where the RML monitor has not detected any violation of the formal specification. Second, if the feedback is a violation ($\perp$), then the `Monitor` agent triggers its reactive mechanism. This is the scenario where the RML monitor has detected a violation, and something has to be done to handle it. Note that what the `Monitor` agent does to react upon perceiving a violation is completely domain specific. That is, RV4JaCa only offers a skeleton `Monitor` agent that does nothing, even when a violation is perceived; it is up to the developer to program what the `Monitor` agent should do in such a case.

In what follows, we show how we used RV4JaCa to tackle a healthcare scenario. In that case, we developed the `Monitor` agent to handle property violations at runtime. Since, as we are going to see, the violations are related to protocol violations caused by the human user, the `Monitor` agent's job is to guide the user back to the expected protocol flow. Naturally, as stressed before, the `Monitor` agent's behaviour is completely determined by its domain of use. In other scenarios, the `Monitor` agent may have different responsibilities and enforce more or less invasive recovery actions onto the system.

4. Bed Allocation Case Study

To clarify how our approach works, we developed a case study applying RV4JaCa in the hospital bed allocation domain. The idea behind such a case study is to verify if, when used in a real application domain, our approach is capable of producing the expected effects. That is, to provide the system with an extra security layer, helping agents to perceive and consequently be able to take action to recover from possible failures during runtime without negatively interfering with system performance. However, we do not perform specific performance tests. We replicated tests performed with real hospital data and real hospital staff responsible for bed allocation, seeking to identify whether the use of RV4JaCa would alter the chatbot's response time in a way that would be noticeable to the user. As well as identifying whether the monitor would be able to identify the violation of a specific property caused by a user interaction and return feedback to the system in time for the recovery mechanisms to be executed before the chatbot sends the next response to the user.

4.1. Hospital Bed Allocation Domain

Resource management in hospitals aims to maximise resource usage and avoid hospital overcrowding. In recent decades, healthcare systems have been facing a massive increase in demand, which has led to an ongoing need to improve and optimise operational processes and quality control methods [33]. In addition, hospital managers have studied ways to improve the use of hospital resources and maintain high occupancy rates without creating chaos in the emergency room or long queues [34]. The demands on hospitals and the growing financial constraints make planning and efficient allocation of hospital beds increasingly difficult [35], especially in developing countries such as Brazil.

Brazil had the second highest burden of coronavirus disease 2019 (COVID-19) worldwide. More than 36.55 cases and 0.93 deaths per thousand inhabitants as of 31 December 2020. In addition, as of 1 October 2021, the country recorded the highest number of deaths in the world (402,220) caused by COVID-19 [36]. With a considerable number of COVID-19 patients worldwide during the pandemic, the hospitals faced massive shortages of isolation beds with an appropriate environment to prevent airborne microorganisms from entering corridors, which could result in secondary infections. On the other hand, they still had to consider the needs of non-COVID-19 patients. Even during the pandemic, many non-COVID-19 patients, mainly those in emergency cases, still required hospitalisation. Therefore, a critical management problem faced was how to optimally allocate the limited amount of hospital beds between COVID-19 and non-COVID-19 patients [37].

Hospital beds are scarce, and therefore, allocating them optimally plays an essential role in the overall planning of hospital resources [38]. Availability of beds in specialised wards for each patient's medical condition can reduce errors and improve the quality of patient care [39]. However, when performing an efficient bed allocation, it is necessary to consider many variables that make it difficult for a human to work out the best solutions without any assistance. Furthermore, this is a complex task computationally, so artificial intelligence incorporated into multi-agent systems can be helpful in this context. Effective management of such resources has always been challenging for managers, given that hospitals' settings are highly dynamic and uncertain. Uncertainty in this domain comes from the fact that hospitals must accommodate patients undergoing elective (scheduled) and emergency treatments requiring multiple specialities in a wide range of departments with varying constraints [34] as well as handle emergency cases that are impossible to predict.

This makes bed management an essential part of planning and controlling operational capacity and an activity involving the efficient use of resources [40]. Thus, it would be interesting to have a system that assists in suggesting better bed allocations for the professional responsible for this task.

4.2. MAIDS—Multi-Agent Intentional Dialogue System

We have developed MAIDS [41], a framework that supports the development of multi-agent intentional dialogue systems to assist humans in decision-making, which can be embodied in a robot to walk around the place and interact with the users. Since one of the domains for which we have built an instance of this framework is hospital bed allocation, the closest work to ours in a hospital scenario, as far as we know, is the one reported in [42], where a robot reads the current status of the work floor and uses machine-learning techniques to make suggestions on resource allocation and uses speech recognition to receive feedback from the resource nurse.

Before moving on with the presentation of our framework, we want to linger a bit longer on its robotic implications. As mentioned in the previous paragraph, our framework is highly usable in robotic scenarios. Indeed, robotic scenarios are easily safety-critical; thus, it is generally unwise to deploy autonomous components without any fail-safe or way to control their behaviour. In this work, we consider the application of RV4JaCa in a healthcare scenario, hence the necessity to check the agents behave properly. However, for the same reasons, agents would need to be controlled and, if needed, restrained in a robotic scenario as well. Because of this, all results we present are easily portable to robotic applications, where the agents are used to guide the robotic components.

As it is shown in Figure 3, to provide an interface with natural language processing platforms, such as Dialogflow https://cloud.google.com/dialogflow/es/docs (accessed on 10 February 2023), our framework relies on the use of Dial4JaCa [43]. The `Human user` can interact with the chatbot through text or voice. Dialogflow classifies the interaction intent and sends it to Dial4JaCa, making the request available to the `Communication expert` agent assigned to that specific user. One or more `Communication expert` agents can be instantiated, each one responsible for representing one particular `Human user`. It uses natural language templates [44] to translate the `Assistant` responses (the result of the MAS reasoning) into natural language and send them to its corresponding `Human user`. The `Assistant` agent performs argumentation reasoning [45] and is responsible for communicating with other agents in the search for information. Several `Ontology expert` agents can be instantiated, allowing the MAS to consult different ontologies simultaneously, given that each of these agents is able to interface with a specific ontology through Onto4JaCa. These agents can also perform ontological reasoning using the Pellet reasoner [46] and its open-source continuation effort Openllet https://github.com/Galigator/openllet (accessed on 10 February 2023). In addition, these agents can translate OWL inference rules [47] automatically to defeasible rules (representing argumentation schemes) and use them during the reasoning process.

We can add `Domain-specific agents` and `Domain-specific Artefacts` to the system to address the specificity of different application domains. For example, in the instance used in this case study, we added specific agents for the bed allocation domain (more details about this scenario can be found in [48]. Among those domain-specific agents, the `Validator` agent is responsible for validating bed allocation plans made by the user (via our system interface) using a PDDL (Planning Domain Definition Language) plan validator; the `Optimiser` agent is responsible for making suggestions for optimised allocations using the GLPSol solver of GLPK http://winglpk.sourceforge.net/ (accessed on 10 February 2023) (GNU Linear Programming Kit), which is a free open-source software for solving linear programming problems; and the `Database` agent is responsible for querying and updating the bed allocation system database.

RV4JaCa has been added to that MAS for collecting information about all messages exchanged between agents and sending them through a REST request to the `Formal monitor` (where the properties that need to be checked are defined). After processing a received message, the monitor returns a result that states whether the message sent from one agent to the other violates any of the properties being checked by it. If a property is violated, RV4JaCa can trigger some recovery method for recovering from some possible failure. Another option is adding the information about the violation to the `Monitor`'s agent belief

base. The `Monitor` agent can react to this addition of belief by warning the agents involved in the exchange of messages that there has been a violation. This makes it possible for our agents to take action to recover from the failure that the breach caused.

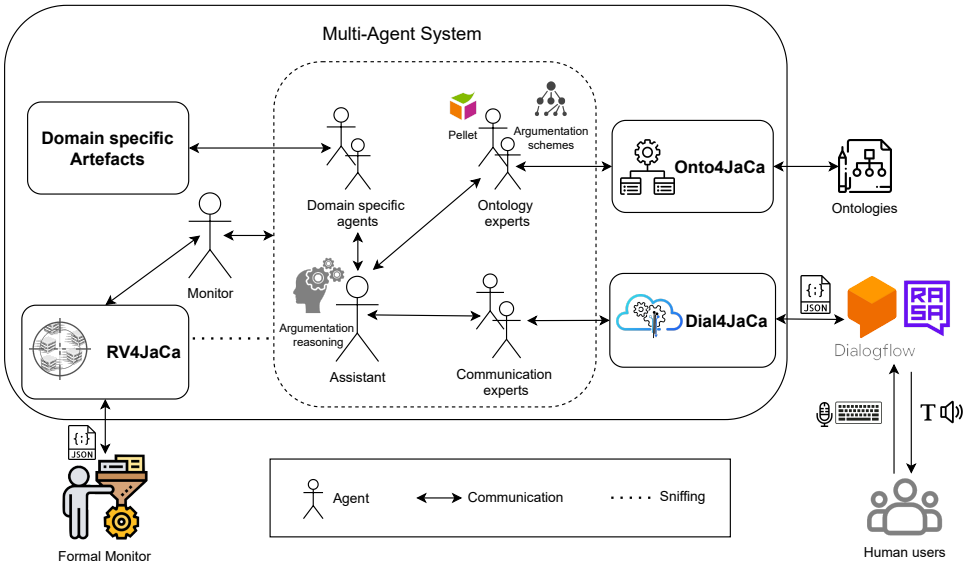

Figure 3. MAIDS Architecture.

MAIDS was implemented in the bed allocation domain, allowing us to test a system developed using our framework in a real-world domain. Hospital São Lucas da PUCRS in Brazil has kindly agreed to support us in evaluating our system. We divided the evaluation of our system into two parts. The first one focused on system functionality. The aim was to verify if the functionalities of the decision support system developed using the MAIDS framework were adequate to the needs of the professionals responsible for allocating beds in the hospital. The second one was intended to verify the expressiveness of our framework when using real data in a real-world domain. In both evaluations, this case study included the following agents:

assistant(a): The internal representation in MAS for a chatbot that assists hospital staff in carrying out bed allocation in a hospital;
ontology (l): An agent with access to ontologies responsible for semantic reasoning using argumentation schemes as defeasible rules generated automatically from the semantic rules contained in the ontology.
operator (o): The internal representation in MAS for the hospital staff member who operates the system for allocating beds;
nurse (n): The internal representation in MAS for a nurse who in that hospital serves as domain expert for bed allocation and whom the operator needs to consult in case of doubt;
database (d): An agent with access to the hospital's general information system for checking details of past and current patients, bed allocations, etc.
validator (v): An agent responsible for validating bed allocation plans using a PDDL plan validator.
optimiser (p): An agent responsible for making suggestions for optimised allocations using an optimiser.

Furthermore, in both evaluations, the following modules were used: Dial4JaCa for communication between the multi-agent system and Dialogflow; Onto4JaCa to access ontology information and rules; and RV4JaCa to observe all conversations between agents and generate logs of these conversations so that they could later be tested using the RV4JaCa module in its entirety.

Although the first evaluation was carried out using fictitious data, it was necessary for us to be able to adapt our system to the needs of the real environment. On the other hand,

for the second evaluation, we loaded our system with real data from the hospital. Then we asked two bed-allocation specialists, who work at that hospital, to evaluate it. Our main objective was to verify if the chatbot's performance matches the points raised by Cohen's desiderata (for more information on Cohen's desiderata, see [49]).

Figure 4 shows our system's interface used during the tests, while Figure 5 shows a conversation between the user and the chatbot.

The ability to monitor the messages exchanged between the agents in this case study is used for two different purposes. The first one is, according to the performative and content of each move, to verify whether the agents are following the predefined communication protocol. This aspect may be crucial for safety-critical and privacy-preserving aspects. For instance, in a healthcare domain, such as bed allocation, the agents might be expected to follow some specific medical guidelines for the communication of personal information (even amongst themselves). Moreover, when in the presence of multiple agents, each one having its own goals, it is common to specify the ideal expected outcome at a more abstract level, where it is more natural to reason upon.

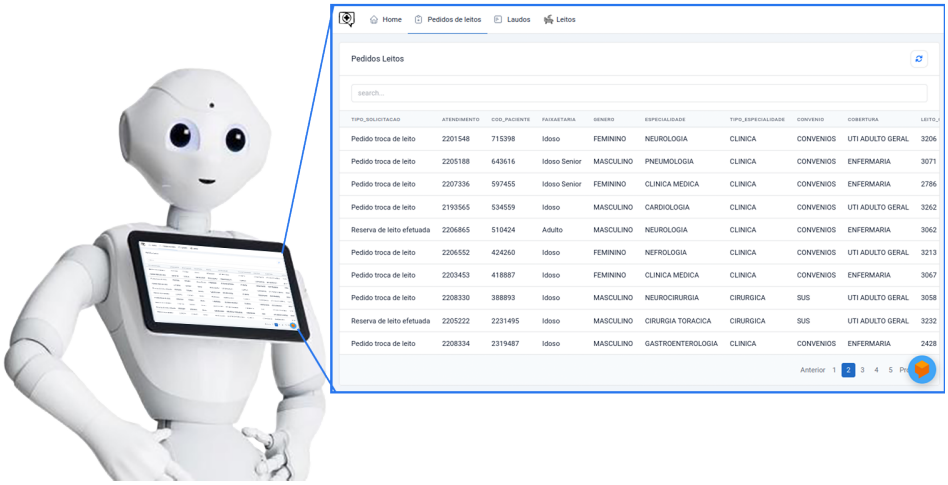

Figure 4. System's interface.

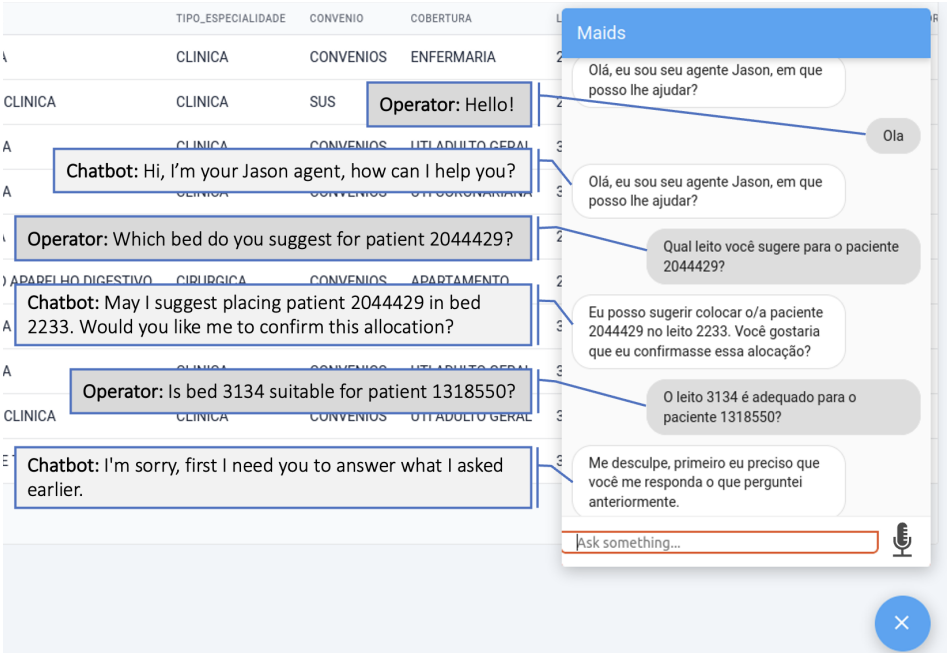

Figure 5. Chatbot's interface.

As we can see in Figure 5, after asking a question to the user ("...Would you like me to confirm this allocation?") and receiving another question instead of an answer ("Is bed 3134 suitable for patient 1318550?"), because of the RV, our chatbot was able to explain an answer was needed before it started doing something else ("I'm sorry, first I need you to answer what I asked earlier."). Furthermore, Listing 2 shows an excerpt of the MAS reasoning log related to that dialogue.

Listing 2. MAS Log.

```
1  [Dial4JaCa] Defining observable property
2  [operator] Request received - Get Suggestion from Dialog
3  [operator] Params: [param("paciente",["2044429"])]
4  [operator] Chatbot of operator asks for suggestions to allocate: ["2044429"]
5  [assistant] Agent operator requesting suggestion.
6  [optimiser] Agent assistant requesting optmised suggestion.
7  [assistant] Result received from agent optimiser
8  [operator] Asking to chatbot: Eu posso sugerir colocar  o/a paciente 2044429 no leito
       2233. Voce gostaria que eu confirmasse essa alocacao?
9  [Dial4JaCa] Reply received from agent
10 [Dial4JaCa] Agent jason response: Eu posso sugerir colocar  o/a paciente 2044429 no
       leito 2233. Voce gostaria que eu confirmasse essa alocacao?
11 [Dial4JaCa] Request received: {...}
12 [Dial4JaCa] Defining observable property
13 [operator] Request received - Verify Suitability from Dialog
14 [operator] Params: [param("paciente","1318550"),param("leito","3134")]
15 [operator] Chatbot of operator is requesting to verify suitability: paciente: 1318550,
       leito: 3134
16 [ontology_specialist] Verifying if adequado("3134","1318550").
17 [monitor] Message mid6 from operator to assistant at Wed Jan 25 10:17:25 BRT 2023 has a
       violation.
18 [assistant] Agent monitor reported a violation.
19 [operator] Asking to chatbot: Me desculpe, primeiro eu preciso que voce me responda o
       que perguntei anteriormente.
20 [Dial4JaCa] Reply received from agent
21 [Dial4JaCa] Agent jason response: Me desculpe, primeiro eu preciso que voce me responda
       o que perguntei anteriormente.
```

The second RML property proposed in this study is to check if a human participant changed the topic of the conversation without the proper conclusion of the previous topic. When we add humans to the agent-to-agent communication loop, the developer has limited control over the human interactions in the dialogue. For example, in some cases, when the MAS is performing some specific task, it is important to be able to finish it before starting a new one. However, when completing this task depends on some human interaction, we cannot guarantee that the human will complete the necessary exchange. Naturally, it would be possible to perform this check within each agent. However, when we have agents specialised in specific tasks, it is preferable to have all the agent plans related to the specific topic rather than other concerns, such as verification, which significantly facilitates implementation and code maintenance.

Below, we report two example properties, written in RML, that have been checked for the bed allocation case study using RV4JaCa.

4.3. RML Properties for the Bed Allocation Domain

The first property, which is presented in Listing 3, concerns checking the user does not change the topic before completing the one currently processed by the agents. In particular, the reported property cares about the 'getValidationResult' topic. This topic relates to the user asking the assistant agent to validate a suggested bed allocation (the corresponding event type is expressed in lines 1–5). When the assistant receives this message, the protocol goes on, causing a sequence of messages exchanged amongst the assistant, the optimiser and the validator agents. Note that this part is not reported because it is not of interest for checking the property. After this step, the assistant agent sends back an answer to the user (the event type is in lines 6–11). Suppose this answer is not empty (i.e., the event contains arg1, and arg2 fields). In that case, the user is expected to conclude the communication with certain content (listed in lines 12–17). For instance, the user could reply with an additional

message containing 'allocValPatients', meaning that the user wants to allocate all patients according to the plan, even if there are failures because the reported failures correspond to rules he is aware of breaking. Naturally, the user might decide not to complete any of the plan allocations that have been validated, as some have flaws. In that scenario, the message would have content 'dontAllocValPatients'. Another possibility is the message has the content 'allocValidValPatients', meaning that the user wants to allocate only those patients where the allocation does not break any rules. The last one is when the user asks for an optimised bed allocation suggestion. In this case, the message would be content 'getOptimisedAllocation'.

Once the events corresponding to previously mentioned messages have been specified (lines 1–26), the actual property can be expressed following the RML syntax (lines 27–30). In more detail, in line 27, we may find the definition of the main term denoting the property to check (which in RML is always called Main). In this scenario, the principal term corresponds to a sequence of subterms named Question. Such a term is defined in line 28 and starts with a question (as defined in lines 1–5). This means, to comply with the protocol, the first event has to be a message containing a 'question' of the 'operator' for the 'assistant' (in this case, regarding the validation of a bed allocation). After that, the property goes on with the Answer term (line 29). Inside it, we find a disjunction between two possible alternatives in the protocol. On the left, we may observe a answer_with_constraint event, which means, according to lines 6–11, that the 'assistant' replied to the 'operator' with a suggested result that the latter has to decide upon. On the right, we may observe an event corresponding to any other answer, which in this specific case denotes the case where the result is empty, meaning that no result is available to be suggested to 'operator'. In the latter case (the right branch), the property ends this cycle since the communication between the two agents is concluded, and new messages concerning new topics can be exchanged in the future. Instead, in the former case (the left branch), the current cycle is not ended because the 'assistant' is still waiting for an answer from the 'operator' regarding the suggested result. This last aspect is handled in the term in line 30, where no question is admissible from the 'operator' except for one amongst the ones listed in lines 12–17. Upon the reception of the event matching one such listed event type, the property cycle ends, and as it happened for the right branch, the protocol can move on.

Now, before presenting another property of interest that we have analysed through RV4JaCa, it is important to detail how a property can be violated. As we mentioned before, we presented the events that are accepted in certain points of the property and why. An RML property is violated whenever, given the current term denoting the current state of the property, and a new event, the property does not accept such an event. For instance, in the property presented in Listing 3, an event which is different from an answer after having observed a question is not acceptable. This can be seen in line 29, where after consuming an event denoting a question, the only possible following events can be an answer requiring additional info (left branch), or a general answer (right branch). Thus, if the observed event is neither of the two, the term is stuck and cannot move on. In RML, this translates into a violation of the property, which is then reported back to the monitor agent, that in turn, will trigger all mechanisms for the agents involved to react properly.

The second RML property we tested in the bed allocation domain is reported in Listing 4. Differently from the property reported in Listing 3, here, we do not check the consistency amongst topics. Instead, we care about checking that an agent always replies to a question before posing a new one. As before, the first part of Listing 4 concerns the definition of which events are of interest for the property (lines 1–10). In this specific case, we have questions (lines 1–5) and answers (lines 6–10). Note that, differently from the previous RML property, here we exploit parameters inside the specification. Indeed, the event types reported in lines 1–10 are all parametric with respect to the agents involved in the communication. This means that such event types do not focus on specific messages exchanged between predefined agents, as it was before, but are free (through RML parameters, we have late binding on the agents involved in the interaction). This makes the

definition of the RML property in line 11 highly parametric, without updating the property for each new agent added to the system. The property is defined in line 11, through the standard Main term in RML. Since the property is parametric, it starts with the let operator, which defines the variables used in the term. In this case, the variables used are ag1, and ag2 (naturally, any other name would have sufficed). After that, the property goes on, expecting a question, followed by a corresponding answer. Here, note that in the first event (i.e., the question), the variables are bound to the agents involved in the communication, while in the second event (i.e., the answer), such variables are ground to the previously initialised values. In this way, a question is free to be sent by any possible agent ag1, to any possible agent ag2 in the system (where ag1 and ag2 are bound to the observed agents involved in the communication); instead, an answer is constrained to be sent by agent ag2 to agent ag1 (with both variables already bound to the respective values through the previously observed question).

Listing 3. The RML specification for checking that no change in topic is observed after the user has requested a validation result.

```
1  question matches {
2      performative:'question',
3      sender:'operator', receiver:'assistant',
4      content:{name:'getValidationResult'}
5  };
6  answer_with_constraint matches
7  {
8      performative:'assert',
9      sender:'assistant', receiver:'operator',
10     content:{name:'answer', name:'result', arg1:_, arg2:_}
11 };
12 constrained_question matches
13     {performative:'question', sender:'operator', receiver:'assistant',
        content:{name:'allocValPatients'}} |
14     {performative:'question', sender:'operator', receiver:'assistant',
        content:{name:'getOptimisedAllocation'}} |
15     {performative:'question', sender:'operator', receiver:'assistant',
        content:{name:'dontAllocValPatients'}} |
16     {performative:'question', sender:'operator', receiver:'assistant',
        content:{name:'allocValidValPatients'}};
17 a_question matches
18 {
19     performative:'question',
20     sender:'operator', receiver:'assistant'
21 };
22 an_answer matches
23 {
24     performative:'assert',
25     sender:'assistant', receiver:'operator'
26 };
27 Main = Question*;
28 Question = (question Answer);
29 Answer = (answer_with_constraint ConstrainedQuestion) \/ (an_answer);
30 ConstrainedQuestion = constrained_question Answer;
```

As before, also with this property, we can ponder on which events can cause a violation. In particular, the property expressed in Listing 4 is violated when after a question between two agents (ag1→ag2), the following event is not the corresponding answer (ag2→ag1) but another message (for instance, another question).

Listing 4. The RML specification for checking that an agent always replies before messaging about something else.

```
question(ag1, ag2) matches
{
    performative:'question',
    sender:ag1, receiver:ag2
};
answer(ag1, ag2) matches
{
    performative:'assert',
    sender:ag1, receiver:ag2
};
Main = {let ag1, ag2; question(ag1, ag2) answer(ag2, ag1)}*;
```

5. Related Works

In past years, some work has focused on formal verification from a more dynamic viewpoint. In [28], the authors presented a framework to verify Agent Interaction Protocols (AIP) during the runtime. The formalism used in this work allows the introduction of variables that are then used to constrain the expected behaviour in a more expressive way. In [31], the same authors proposed an approach to verify AIPs during runtime using multiple monitors. This is obtained by decentralising the global specification (specified as a Trace Expression [27]), which is used to represent the global protocol, into partial specifications denoting the single agents' perspective. Both those approaches are based on the formalism, serving as a building block for RML's semantics. From this perspective, RV4JaCa is an evolution of these approaches in two ways: (i) it allows general-purpose verification since no constraint is assumed on the monitor side, except for being capable of receiving and sending JSON messages; (ii) to be self-contained, RV4JaCa natively supports RML monitors, and because of that, it allows a more intuitive and high-level protocol specification. In [50,51], other approaches to runtime verification of agent interactions are proposed, and in [52], a framework for dynamic adaptive MAS (DAMS-RV) based on an adaptive feedback loop is presented. Other approaches to MAS RV include the spin-off proposals from the SOCS project where the SCIFF computational logic framework [53] is used to provide the semantics of social integrity constraints. To model MAS interaction, expectation-based semantics specify the links between observed and expected events, providing a means to test runtime conformance of an actual conversation with respect to a given interaction protocol [54]. Similar work has been performed using commitments [55].

To the best of our knowledge, RV4JaCa is the first approach that integrates RV within JaCaMo to verify Agent Interaction Protocols.

In the context of the case study used in our work, robotics has been largely used to support the field of healthcare, not only helping those people who provide healthcare but also those who need healthcare (see [56–59] for a recent overview of the advances in the field of robotics applied to healthcare). The field increasingly demands studies on expressive and friendly communication techniques, for example, those observed by integrating key technologies, such as chatbots technologies, multi-agent systems, argumentation and ontologies [41,43,48]. To the best of our knowledge, RV4JaCa is the first approach that allows us to verify an Agent Interaction Protocol in this complex context, thanks to those key integrations also implemented in the JaCaMo Framework. Most important, the proposed approach allows agents to understand when desired properties related to human–agent interaction are violated, allowing them to act accordingly, for example, informing the user that a topic of conversation needs to be finished before changing to a new one (as shown in Section 4.3), which represents a desired behaviour of robots interacting with humans users.

6. Conclusions and Future Work

Communications between agents play a key role in the functioning of a multi-agent system since, in practice, agents rarely act alone, and they usually inhabit an environment that contains other agents. Therefore, an extra layer of security that allows us to verify key aspects of this message exchange adds great value, in addition to great possibilities for

improvement since certain aspects do not need to be considered when developing each of the agents. Using this type of formal verification at runtime allows us to standardise the interaction between agents through previously defined protocols that all agents must follow and, if they do not, react in a way that the execution is not negatively affected by the effects that were caused by this protocol deviation.

On the other hand, the checks performed with RV are not limited to protocol validation. More specific properties of each application domain can also be verified once the monitor has access to the content of the exchanged messages. Even the execution of certain routines or functions according to the direction in which the conversations between the agents go can be performed. For example, recording the results obtained during the agents' reasoning in a database without agents having the responsibility to carry out the registrations themselves. Or even sending an automatic email to a supervisor if any property identified by an agent and communicated to another is outside certain parameters. Therefore, depending on the MAS's domain, there is a range of possibilities in which RV can be used.

Based on that, we proposed RV4JaCa, an approach to integrate multi-agent systems and runtime verification. Our approach was built using JaCaMo and RML, and it provides significant progress toward obtaining guarantees of the correct execution of the MAS. The case study presented in this paper demonstrates the use of RV4JaCa in practice and also uses real data from a real-world domain. In addition, it is important to note that our approach can be applied to different scenarios in different MAS. Based on the presented case study, we created two distinct properties to be checked at runtime in the system. The first one is to allow agents to be alerted if there is an unexpected change in the topic of conversation by the human user who is interacting with the system. The second one is capable of verifying that the previously defined communication protocol between agents is being followed correctly (in particular, question-answer relations).

Note that, as we mention different times in the paper, our approach is not limited to MAS used in isolation. Specifically, we consider MAS deployed in possibly safety-critical environments, such as robotic applications. In this paper, we stress this aspect by showing how to enhance the presented case study with actual robotic components. However, we mainly focus on the engineering implications and uses of RV4JaCa in such a scenario. Even though the robotic application is exploited in the case study, but only envisaged and studied, there is still a need to tackle the technological gap between the JaCaMo agents and the robotic environment. Nonetheless, such a technological gap has already been partially solved in recent works, such as in [60], where a bridge between BDI agents and robotic applications has been proposed. The integration of [60] with RV4JaCa is out of the scope of our paper; however, it would not require too much effort since [60] already supports BDI agents.

As MAS have been used to build systems focused on explainability, RV's extra safety layer is certainly useful since, to achieve explainability in MAS, much communication between agents and humans needs to be carried out. Furthermore, we need to take into account that the developer does not have complete control over the interactions that human users can make during a dialogue. In this sense, RV allows us to avoid unexpected and probably inappropriate system behaviour.

In future work, we plan to further extend RV4JaCa to more than interaction protocols. For instance, it would be relevant to check the agents' state of mind as well. Moreover, through RV4JaCa, we also want to create a library of verifiably correct Agent Interaction Protocols to be used in different scenarios involving both agents and humans in the loop.

Author Contributions: Conceptualization, D.C.E. and A.F.; methodology, D.C.E. and A.F.; software, D.C.E.; validation, D.C.E., A.F., A.R.P., D.A., R.H.B. and V.M.; formal analysis, D.C.E., A.F., A.R.P., D.A., R.H.B. and V.M.; investigation, D.C.E., A.F., A.R.P., D.A., R.H.B. and V.M.; resources, D.C.E. and A.F.; data curation, D.C.E.; writing—original draft preparation, D.C.E. and A.F.; writing—review and editing, D.C.E., A.F., A.R.P., D.A., R.H.B. and V.M.; visualization, D.C.E. and A.F.; supervision, A.F.; project administration, D.C.E.; funding acquisition, D.A. and R.H.B. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially funded by the National Council for Scientific and Technological Development (CNPq), the Coordination for the Improvement of Higher Education Personnel (CAPES), and the MUR project "T-LADIES" (PRIN 2020TL3X8X).

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

Multidisciplinary Digital Publishing Institute	MDPI
Directory of open access journals	DOAJ
Runtime Verification	RV
Multi-Agent Systems	MAS
Artificial Intelligence	AI
Agent-Centred Multi-Agent Systems	ACMAS
Organisation-Centred Multi-Agent Systems	OCMAS
Explainable Artificial Intelligence	XAI
Belief–Desire–Intention	BDI
Extensible Markup Language	XML
Runtime Monitoring Language	RML
Domain-Specific Language	DSL
Representational State Transfer	REST
GNU Linear Programming Kit	GLPK
Planning Domain Definition Language	PDDL
Multi-Agent Intentional Dialogue System	MAIDS
W3C Web Ontology Language	OWL

References

1. Bordini, R.H.; Dastani, M.; Dix, J.; Seghrouchni, A.E.F. (Eds.) *Multi-Agent Programming, Languages, Tools and Applications*; Springer: Berlin/Heidelberg, Germany, 2009. [CrossRef]
2. Schmidt, D.; Panisson, A.R.; Freitas, A.; Bordini, R.H.; Meneguzzi, F.; Vieira, R. An Ontology-Based Mobile Application for Task Managing in Collaborative Groups. In *Proceedings of the Twenty-Ninth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2016, Key Largo, FL, USA, 16–18 May 2016*; Markov, Z., Russell, I., Eds.; AAAI Press: Palo Alto, CA, USA, 2016; pp. 522–526.
3. Cardoso, R.C.; Ferrando, A.; Briola, D.; Menghi, C.; Ahlbrecht, T. Agents and Robots for Reliable Engineered Autonomy: A Perspective from the Organisers of AREA 2020. *J. Sens. Actuator Netw.* **2021**, *10*, 33. [CrossRef]
4. Winikoff, M. BDI agent testability revisited. *Auton. Agents Multi Agent Syst.* **2017**, *31*, 1094–1132. [CrossRef]
5. Winikoff, M. Debugging Agent Programs with Why?: Questions. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, 8–12 May 2017*; Larson, K., Winikoff, M., Das, S., Durfee, E.H., Eds.; ACM: Richland, SC, USA, 2017; pp. 251–259. [CrossRef]
6. Dennis, L.A.; Fisher, M.; Webster, M.P.; Bordini, R.H. Model checking agent programming languages. *Autom. Softw. Eng.* **2012**, *19*, 5–63. [CrossRef]
7. Bartocci, E.; Falcone, Y.; Francalanza, A.; Reger, G. Introduction to Runtime Verification. In *Lectures on Runtime Verification—Introductory and Advanced Topics*; Bartocci, E., Falcone, Y., Eds.; Springer: Cham, Switzerland, 2018; Volume 10457, pp. 1–33. [CrossRef]
8. Leucker, M.; Schallhart, C. A brief account of runtime verification. *J. Log. Algebr. Methods Program.* **2009**, *78*, 293–303. [CrossRef]
9. Ahrendt, W.; Chimento, J.M.; Pace, G.J.; Schneider, G. Verifying data- and control-oriented properties combining static and runtime verification: theory and tools. *Form. Methods Syst. Des.* **2017**, *51*, 200–265. [CrossRef]
10. Boissier, O.; Bordini, R.H.; Hübner, J.F.; Ricci, A.; Santi, A. Multi-agent oriented programming with JaCaMo. *Sci. Comput. Program.* **2013**, *78*, 747–761. [CrossRef]
11. Engelmann, D.C.; Ferrando, A.; Panisson, A.R.; Ancona, D.; Bordini, R.H.; Mascardi, V. RV4JaCa–Runtime Verification for Multi-Agent Systems. In Proceedings of the Second Workshop on Agents and Robots for Reliable Engineered Autonomy, AREA@IJCAI-ECAI 2022, Vienna, Austria, 24 July 2022; pp. 23–36. [CrossRef]
12. Wooldridge, M. *An Introduction to MultiAgent Systems*; John Wiley & Sons Ltd.: Hoboken, NJ, USA, 2002.
13. Bordini, R.H.; Hübner, J.F.; Wooldridge, M. *Programming Multi-Agent Systems in AgentSpeak Using Jason*; John Wiley & Sons: Hoboken, NJ, USA, 2007.

14. Ferber, J.; Gutknecht, O.; Michel, F. From Agents to Organizations: An Organizational View of Multi-agent Systems. In *Proceedings of the Agent-Oriented Software Engineering IV, 4th International Workshop, AOSE 2003, Melbourne, Australia, 15 July 2003*; Giorgini, P., Müller, J.P., Odell, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2003; Volume 2935, pp. 214–230. [CrossRef]
15. Horling, B.; Lesser, V.R. A survey of multi-agent organizational paradigms. *Knowl. Eng. Rev.* **2004**, *19*, 281–316. [CrossRef]
16. Anjomshoae, S.; Najjar, A.; Calvaresi, D.; Främling, K. Explainable agents and robots: Results from a systematic literature review. In Proceedings of the 18th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2019), Montreal, QC, Canada, 13–17 May 2019; pp. 1078–1088. [CrossRef]
17. Adadi, A.; Berrada, M. Peeking inside the black-box: A survey on Explainable Artificial Intelligence (XAI). *IEEE Access* **2018**, *6*, 52138–52160. [CrossRef]
18. Donadello, I.; Dragoni, M.; Eccher, C. Explaining reasoning algorithms with persuasiveness: A case study for a behavioural change system. In Proceedings of the 35th Annual ACM Symposium on Applied Computing, Brno, Czech Republic, 30 March–3 April 2020; pp. 646–653. [CrossRef]
19. Rao, A.S.; Georgeff, M.P. BDI agents: From theory to practice. In Proceedings of the First International Conference on Multiagent Systems, San Francisco, CA, USA, 12–14 June 1995; Volume 95, pp. 312–319.
20. Hubner, J.F.; Sichman, J.S.; Boissier, O. Developing organised multiagent systems using the MOISE+ model: Programming issues at the system and agent levels. *Int. J. Agent Oriented Softw. Eng.* **2007**, *1*, 370–395. [CrossRef]
21. Ricci, A.; Piunti, M.; Viroli, M.; Omicini, A. Environment programming in CArtAgO. In *Multi-Agent Programming*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 259–288. [CrossRef]
22. Cardoso, R.C.; Ferrando, A. A Review of Agent-Based Programming for Multi-Agent Systems. *Computers* **2021**, *10*, 16. [CrossRef]
23. Clarke, E.M. Model checking. In *Proceedings of the International Conference on Foundations of Software Technology and Theoretical Computer Science*; Springer: Berlin/Heidelberg, Germany, 1997; pp. 54–56. [CrossRef]
24. Loveland, D.W. *Automated Theorem Proving: A Logical Basis*; Fundamental Studies in Computer Science; North-Holland: Amsterdam, The Netherlands, 1978; Volume 6.
25. Fisher, M.; Mascardi, V.; Rozier, K.Y.; Schlingloff, B.; Winikoff, M.; Yorke-Smith, N. Towards a framework for certification of reliable autonomous systems. *Auton. Agents Multi Agent Syst.* **2021**, *35*, 8. [CrossRef]
26. Ancona, D.; Franceschini, L.; Ferrando, A.; Mascardi, V. RML: Theory and Practice of a Domain Specific Language for Runtime Verification. *Sci. Comput. Program.* **2021**, *205*, 102610. [CrossRef]
27. Ancona, D.; Ferrando, A.; Mascardi, V. Comparing Trace Expressions and Linear Temporal Logic for Runtime Verification. In *Proceedings of the Theory and Practice of Formal Methods—Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*; Ábrahám, E., Bonsangue, M.M., Johnsen, E.B., Eds.; Springer: Berlin/Heidelberg, Germany, 2016; Volume 9660, pp. 47–64. [CrossRef]
28. Ancona, D.; Ferrando, A.; Mascardi, V. Parametric Runtime Verification of Multiagent Systems. In Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, 8–12 May 2017; pp. 1457–1459. [CrossRef]
29. Falcone, Y.; Havelund, K.; Reger, G. A Tutorial on Runtime Verification. In *Engineering Dependable Software Systems*; Broy, M., Peled, D.A., Kalus, G., Eds.; IOS Press: Amsterdam, The Netherlands 2013; Volume 34, pp. 141–175. [CrossRef]
30. Havelund, K.; Rosu, G. An Overview of the Runtime Verification Tool Java PathExplorer. *Form. Methods Syst. Des.* **2004**, *24*, 189–215. [CrossRef]
31. Ferrando, A.; Ancona, D.; Mascardi, V. Decentralizing MAS Monitoring with DecAMon. In Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, 8–12 May 2017; pp. 239–248. [CrossRef]
32. Ancona, D.; Briola, D.; Ferrando, A.; Mascardi, V. Global Protocols as First Class Entities for Self-Adaptive Agents. In Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2015, Istanbul, Turkey, 4–8 May 2015; pp. 1019–1029.
33. Elalouf, A.; Wachtel, G. Queueing Problems in Emergency Departments: A Review of Practical Approaches and Research Methodologies. In *Proceedings of the Operations Research Forum*; Springer: Berlin/Heidelberg, Germany, 2022; Volume 3, pp. 1–46. [CrossRef]
34. Grübler, M.D.S.; da Costa, C.A.; Righi, R.; Rigo, S.; Chiwiacowsky, L. A hospital bed allocation hybrid model based on situation awareness. *Comput. Inform. Nurs.* **2018**, *36*, 249–255. [CrossRef] [PubMed]
35. Matos, J.; Rodrigues, P.P. Modeling decisions for hospital bed management—A review. In Proceedings of the 4th International Conference on Health Informatics, Rome, Italy, 26–29 January 2011; pp. 504–507. [CrossRef]
36. Bigoni, A.; Malik, A.M.; Tasca, R.; Carrera, M.B.M.; Schiesari, L.M.C.; Gambardella, D.D.; Massuda, A. Brazil's health system functionality amidst of the COVID-19 pandemic: An analysis of resilience. *Lancet Reg. Health Am.* **2022**, *10*, 100222. [CrossRef]
37. Ma, X.; Zhao, X.; Guo, P. Cope with the COVID-19 pandemic: Dynamic bed allocation and patient subsidization in a public healthcare system. *Int. J. Prod. Econ.* **2022**, *243*, 108320. [CrossRef] [PubMed]
38. Teow, K.L.; El-Darzi, E.; Foo, C.; Jin, X.; Sim, J. Intelligent Analysis of Acute Bed Overflow in a Tertiary Hospital in Singapore. *J. Med. Syst.* **2012**, *36*, 1873–1882. [CrossRef] [PubMed]
39. Zhang, C.; Eken, T.; Jørgensen, S.B.; Thoresen, M.; Søvik, S. Effects of patient-level risk factors, departmental allocation and seasonality on intrahospital patient transfer patterns: Network analysis applied on a Norwegian single-centre data set. *BMJ Open* **2022**, *12*, e054545. [CrossRef]

40. Proudlove, N.C.; Gordon, K.; Boaden, R. Can good bed management solve the overcrowding in accident and emergency departments? *Emerg. Med. J.* **2003**, *20*, 149–155. [CrossRef]
41. Engelmann, D.C.; Panisson, A.R.; Vieira, R.; Hübner, J.F.; Mascardi, V.; Bordini, R.H. MAIDS—A Framework for the Development of Multi-Agent Intentional Dialogue Systems. In Proceedings of the 22st International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2023, London, UK, 29 May–2 June 2023.
42. Gombolay, M.C.; Yang, X.J.; Hayes, B.; Seo, N.; Liu, Z.; Wadhwania, S.; Yu, T.; Shah, N.; Golen, T.; Shah, J.A. Robotic assistance in coordination of patient care. In Proceedings of the 12nd Robotics: Science and Systems Conference, Cambridge, MA, USA, 12–14 July 2016; pp. 26–37.
43. Engelmann, D.C.; Damasio, J.; Krausburg, T.; Borges, O.T.; da Silveira Colissi, M.; Panisson, A.R.; Bordini, R.H. Dial4JaCa—A Communication Interface Between Multi-agent Systems and Chatbots. In *Proceedings of the Advances in Practical Applications of Agents, Multi-Agent Systems, and Social Good. The PAAMS Collection—19th International Conference, PAAMS 2021, Salamanca, Spain, 6–8 October 2021*; Dignum, F., Corchado, J.M., de la Prieta, F., Eds.; Springer: Berlin/Heildeberg, Germany, 2021; Volume 12946, pp. 77–88. ._7. [CrossRef]
44. Panisson, A.R.; Engelmann, D.C.; Bordini, R.H. Engineering Explainable Agents: An Argumentation-Based Approach. In *Proceedings of the Engineering Multi-Agent Systems—9th International Workshop, EMAS 2021, Virtual Event, 3–4 May 2021*; Alechina, N., Baldoni, M., Logan, B., Eds.; Springer: Berlin/Heildeberg, Germany, 2021; Volume 13190, pp. 273–291. [CrossRef]
45. Panisson, A.R.; Meneguzzi, F.; Vieira, R.; Bordini, R.H. An approach for argumentation-based reasoning using defeasible logic in multi-agent programming languages. In Proceedings of the 11th International Workshop on Argumentation in Multiagent Systems, Paris, France, 5 May 2014; pp. 1–15.
46. Sirin, E.; Parsia, B.; Grau, B.C.; Kalyanpur, A.; Katz, Y. Pellet: A practical OWL-DL reasoner. *J. Web Semant.* **2007**, *5*, 51–53. [CrossRef]
47. Horrocks, I.; Patel-Schneider, P.F.; Boley, H.; Tabet, S.; Grosof, B.; Dean, M. SWRL: A semantic web rule language combining OWL and RuleML. *W3C Memb. Submiss.* **2004**, *21*, 1–31.
48. Engelmann, D.C.; Cezar, L.D.; Panisson, A.R.; Bordini, R.H. A Conversational Agent to Support Hospital Bed Allocation. In *Proceedings of the Intelligent Systems—10th Brazilian Conference, BRACIS 2021, Virtual Event, 29 November–3 December 2021*; Britto, A., Delgado, K.V., Eds.; Springer: Berlin/Heildeberg, Germany, 2021; Volume 13073, pp. 3–17. [CrossRef]
49. Cohen, P. Foundations of Collaborative Task-Oriented Dialogue: What's in a Slot? In *Proceedings of the 20th Annual SIGdial Meeting on Discourse and Dialogue*; Association for Computational Linguistics: Stockholm, Sweden, 2019; pp. 198–209.
50. Bakar, N.A.; Selamat, A. Runtime Verification of Multi-agent Systems Interaction Quality. In Proceedings of the Intelligent Information and Database Systems—5th Asian Conference, ACIIDS 2013, Kuala Lumpur, Malaysia, 18–20 March 2013; Volume 7802, pp. 435–444. [CrossRef]
51. Roungroongsom, C.; Pradubsuwun, D. Formal Verification of Multi-agent System Based on JADE: A Semi-runtime Approach. In *Recent Advances in Information and Communication Technology 2015*; Springer: Berlin/Heildeberg, Germany, 2015; pp. 297–306. [CrossRef]
52. Lim, Y.J.; Hong, G.; Shin, D.; Jee, E.; Bae, D. A runtime verification framework for dynamically adaptive multi-agent systems. In Proceedings of the 2016 International Conference on Big Data and Smart Computing, BigComp 2016, Hong Kong, China, 18–20 January 2016; pp. 509–512. [CrossRef]
53. Alberti, M.; Gavanelli, M.; Lamma, E.; Mello, P.; Torroni, P. The SCIFF Abductive Proof-Procedure. In Proceedings of the AI*IA 2005: Advances in Artificial Intelligence, 9th Congress of the Italian Association for Artificial Intelligence, Milan, Italy, 21–23 September 2005; Volume 3673, pp. 135–147. [CrossRef]
54. Torroni, P.; Yolum, P.; Singh, M.P.; Alberti, M.; Chesani, F.; Gavanelli, M.; Lamma, E.; Mello, P. Modelling Interactions via Commitments and Expectations. In *Handbook of Research on Multi-Agent Systems—Semantics and Dynamics of Organizational Models*; Dignum, V., Ed.; IGI Global: Hershey, PA, USA, 2009; pp. 263–284. [CrossRef]
55. Chesani, F.; Mello, P.; Montali, M.; Torroni, P. Commitment Tracking via the Reactive Event Calculus. In Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, CA, USA, 11–17 July 2009; pp. 91–96.
56. Kyrarini, M.; Lygerakis, F.; Rajavenkatanarayanan, A.; Sevastopoulos, C.; Nambiappan, H.R.; Chaitanya, K.K.; Babu, A.R.; Mathew, J.; Makedon, F. A survey of robots in healthcare. *Technologies* **2021**, *9*, 8. [CrossRef]
57. Holland, J.; Kingston, L.; McCarthy, C.; Armstrong, E.; O'Dwyer, P.; Merz, F.; McConnell, M. Service robots in the healthcare sector. *Robotics* **2021**, *10*, 47. [CrossRef]
58. Khan, A.; Anwar, Y. Robots in healthcare: A survey. In *Proceedings of the Science and Information Conference*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 280–292.
59. Riek, L.D. Healthcare robotics. *Commun. ACM* **2017**, *60*, 68–78. [CrossRef]
60. Cardoso, R.C.; Ferrando, A.; Dennis, L.A.; Fisher, M. An Interface for Programming Verifiable Autonomous Agents in ROS. In Proceedings of the Multi-Agent Systems and Agreement Technologies—17th European Conference, EUMAS 2020, and 7th International Conference, AT 2020, Thessaloniki, Greece, 14–15 September 2020; Volume 12520, pp. 191–205. [CrossRef]