MDPI

Article

VEsNA, a Framework for Virtual Environments via Natural Language Agents and Its Application to Factory Automation †

Andrea Gatti and Viviana Mascardi *

Department of Informatics, Bioengineering, Robotics and Systems Engineering (DIBRIS)—University of Genoa, Via Dodecaneso 35, 16146 Genoa, Italy; andrea.gatti@edu.unige.it

* Correspondence: viviana.mascardi@unige.it

† This Paper Extends the Work in Gatti, A.; Mascardi, V. Towards VEsNA, a Framework for Managing Virtual Environments via Natural Language Agents. In Proceedings of the Second Workshop on Agents and Robots for reliable Engineered Autonomy, AREA@IJCAI-ECAI 2022, Vienna, Austria, 24 July 2022; Cardoso, R.C., Ferrando, A., Papacchini, F., Askarpour, M., Dennis, L.A., Eds.; Volume 362, EPTCS, pp. 65–80. https://doi.org/10.4204/EPTCS.362.8.

Abstract: Automating a factory where robots are involved is neither trivial nor cheap. Engineering the factory automation process in such a way that return of interest is maximized and risk for workers and equipment is minimized is hence, of paramount importance. Simulation can be a game changer in this scenario but requires advanced programming skills that domain experts and industrial designers might not have. In this paper, we present the preliminary design and implementation of a general-purpose framework for creating and exploiting Virtual Environments via Natural language Agents (VEsNA). VEsNA takes advantage of agent-based technologies and natural language processing to enhance the design of virtual environments. The natural language input provided to VEsNA is understood by a chatbot and passed to an intelligent cognitive agent that implements the logic behind displacing objects in the virtual environment. In the complete VEsNA vision, for which this paper provides the building blocks, the intelligent agent will be able to reason on this displacement and on its compliance with legal and normative constraints. It will also be able to implement what-if analysis and case-based reasoning. Objects populating the virtual environment will include active objects and will populate a dynamic simulation whose outcomes will be interpreted by the cognitive agent; further autonomous agents, representing workers in the factory, will be added to make the virtual environment even more realistic; explanations and suggestions will be passed back to the user by the chatbot.

Keywords: virtual reality; chatbot; intelligent agents; factory automation

Citation: Gatti, A.; Mascardi, V. VEsNA, a Framework for Virtual Environments via Natural Language Agents and Its Application to Factory Automation. *Robotics* **2023**, *12*, 46. https://doi.org/10.3390/robotics12020046

Academic Editor: Charalampos Bechlioulis

Received: 15 February 2023
Revised: 16 March 2023
Accepted: 17 March 2023
Published: 21 March 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction and Motivation

Factory automation is a safety-critical task that can significantly help in increasing production, but that does not come free of charge. Robots may be very expensive, and their impact on the production pipeline is not always predictable. For this reason, many factory automation commercial software applications exist, including Computer-Aided Design (CAD) tools and simulators (See, for example, www.fastsuite.com/solutions-products/market-specific/factory-automation, store.indusuite.com/products/software/, www.createasoft.com/automation-simulation, https://new.siemens.com/global/en/products/automation/topic-areas/simulation-for-automation.html, accessed on 19 January 2023). Those applications are highly customized for factory automation. They are complete and efficient in that domain but require advanced programming skills to be used, which might prevent some categories of users from taking advantage of them—for example, small factories where industrial designers do not have sufficient computer science background or where the price of those commercial software applications cannot be afforded. In those situations, a cheap and "extremely-easy-to-setup" virtual twin of the factory, along with the robots and the

workers therein, might bring many benefits, even if it is less effective and complete than the simulations generated by ad-hoc toolkits.

Consider, for example, the following scenario. Alice runs a small factory where various types of metallic components for industrial automation are assembled. She is a visionary businesswoman and, in her vision, safety deserves first place. Alice received funding for making her factory more efficient and productive by installing new robots. She wants to identify the safest configuration of robots in the industrial building that she rented, but her limited budget does not allow her to pay for the license of a commercial CAD or simulation tool. Alice is aware of specific issues raised by the robots she is going to buy and install, which may generate hard and soft constraints to meet and of general legal requirements that may prevent her from making some choices. She also knows that robots may undergo malfunctioning but that also human workers may not behave in the correct way. To ensure safety, she would like to anticipate—or at least to understand—what might go wrong with different configurations of the robots. For this reason, she needs a (possibly inexpensive and open) tool that allows her to:

1. displace elements (different kinds of robots, furniture, ...) into a virtual environment via a user-friendly interface based on natural language interaction in order to identify the best configuration of the building;
2. check that displaced objects meet the hard and soft safety and legal constraints related to their position and interactions;
3. provide a natural language explanation of why some constraints are not met by a given configuration and suggest alternatives.

These three goals might be achieved by adding a natural language interface on top of any existing Computer-Aided Design tool, but Alice needs something more sophisticated. Indeed, she would like to insert dynamic elements into the virtual environment so that the result is not just a static rendering but a running simulation. In particular, she wants to add virtual humans, and she wants to anticipate what may happen in the case of unexpected or wrong maneuvers made by the human workers. The tool that Alice is looking for should also allow her to

4. run simulations in the virtual environment, get statistics about them and explain—using a natural language interface—what may go wrong.

However, this is still not enough. Once the best configuration is devised, Alice wants to train her employees to move and act in the factory before they start working in the real environment. The last feature of the tool Alice needs is to

5. allow workers to *enter* the virtual and dynamic environment, interact with robots therein, learn what is safe and what is not, and get the most effective and realistic training with natural language explanations.

The tool Alice is looking for should

1. understand commands issued in natural language; those commands might range from the simplest “Add a Yaskawa MA2010 in front on the right” to the more sophisticated “Add workers with some profile (experienced, unexperienced, reliable, etc...)”; the tool should provide a chatbot-like natural language interface;
2. be aware of rules (normative, physical, domain-dependent) and be able to verify whether a given configuration—that may include robots and workers—meets them; the tool should be able to reason on facts represented in a symbolic way (“a robot is placed in this position and an unexperienced worker is placed in that position”), on their logical consequences, and on the rules that may be broken by them;
3. be able to synthesize natural language explanations of its logical reasoning flow, besides reasoning and understanding natural language;
4. ensure a one-to-one correspondence between facts representing symbolic knowledge amenable for logical reasoning and objects placed in a virtual environment—equipped with a realistic graphical interface—where dynamic behaviors may be added: the tool should be suitable for running realistic simulations;

5 allow the virtual environment to be made available as a virtual reality for training purposes, with no extra effort.

To the best of our knowledge, no such a tool exists: this paper takes the initial steps toward it by describing VEsNA, an implemented general-purpose framework for managing Virtual Environments via Natural language Agents (VEsNA is freely available to the community from https://github.com/driacats/VEsNA, accessed on 19 January 2023). At its current stage of development, VEsNA is far from offering the services that Alice needs. Nevertheless, the three technologies it builds upon have the *potential* to cope with all her needs, and some promising results have already been achieved in the natural language interaction, reasoning, explanation, simulation, and training in the virtual reality dimension, although no integration among these capabilities has been provided yet. In fact, VEsNA exploits

(i) **Dialogflow** (Thus far, Dialogflow is the only non-open-source technology in VEsNA; we have already developed a new VEsNA release, still under testing and hence not available to the community yet, where we substituted it with an open-source equivalent application, Rasa (https://rasa.com/, accessed on 19 January 2023), to make VEsNA completely open-source) for building a chatbot-like interface,
(ii) **JaCaMo** for integrating knowledge coming from the interaction with the user into a cognitive, rule-driven agent able to reason about this knowledge and to provide human-readable explanations;
(iii) **Unity** for building the dynamic virtual environment, and letting human users immerse in it.

The goal that VEsNA aims to achieve in the future is to provide Alice and many other users with an integrated environment for managing virtual environments via natural language and cognitive agents able to support decision-making thanks to their reasoning and explanatory capabilities: an example of interaction between Alice and VEsNA is depicted in Figure 1.

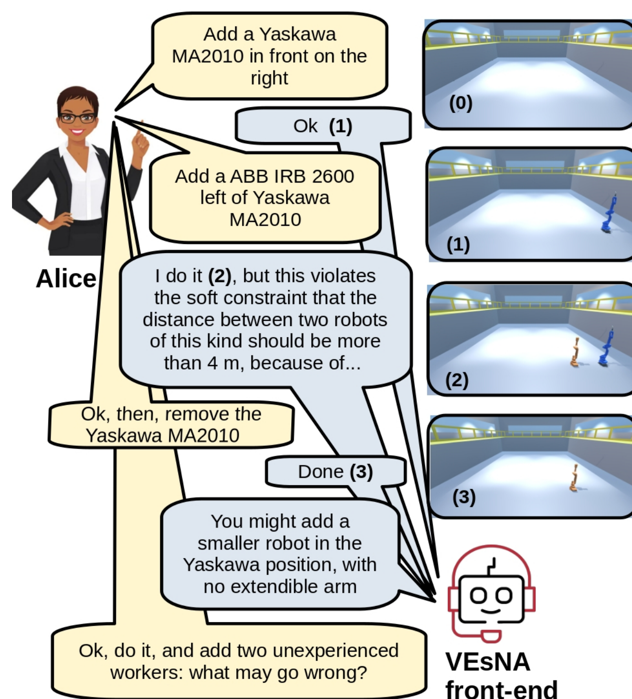


Figure 1. Example of visionary interaction between a user and VEsNA. On the right, the initial empty scene tagged with number **(0)** and, below, the scenes resulting from actions operated by VEsNA, according to the dialogue on the left. Numbers are also inserted in the dialogue to better clarify which part of the conversation results into which action and hence into which scene.

The problem that VEsNA solves in its current state is to integrate Dialogflow, JaCaMo, and Unity into a single framework and to provide the means to *displace the elements relevant for finding the best configuration of the industrial building into a simulated environment via a user-friendly interface based on natural language interaction*.

The paper is organized in the following way. Section 2 introduces the three technologies VEsNA builds upon and motivates our confidence that filling the gap between the VEsNA-as-is and the VEsNA-to-be is technically feasible. Section 3 positions our contribution with regard to the state-of-the-art. Section 4 illustrates the VEsNA architecture and workflow, along with some technical details that allow users to better understand VEsNA and run experiments with it, and Section 5 presents an example in the factory automation domain. Section 6 concludes the paper and discusses future developments.

2. Background

Dialogflow [1] is a *lifelike conversational AI with state-of-the-art virtual agents* developed by Google. It allows users to create personal chatbots, namely **conversational agents** equipped with **intents**, **entities**, and **fulfillment**.

The Dialogflow component that handles concurrent conversations with end-users is named the **Dialogflow agent**. As reported in the Dialogflow documentation (https://cloud.google.com/dialogflow/es/docs/agents-overview, accessed on 19 January 2023), *"it is a natural language understanding module that understands the nuances of human language. [...] A Dialogflow agent is similar to a human call center agent"*: it is trained to handle expected conversation scenarios, and the training does not need to be overly explicit.

During a conversation between humans, a human speaker can utter different types of sentences, each one with a different intentional meaning. That meaning can be identified as the *intent* of that sentence. For example, if someone says *"Hello!"*, the hearer can infer that the (friendly, socially-oriented) intent is to greet them. If someone says *"Go away!"*, the (unfriendly, command-like, action-oriented) intent is to have some concrete action performed by the hearer—namely, moving away. The speech act theory [2,3] provides a theoretical and philosophical basis for this intention-driven communication model. In order to explain the map between sentences and intents to the chatbot agent, the agent's developer should provide examples of sentences that convey that intent for each intent that is relevant to the application.

Real sentences are much more complex than *"Hello!"* and *"Go away!"*: they usually add some contextual information to the main intent. For example, in the *"My name is Bob"* sentence, the speaker's (friendly, socially-oriented) intent is to introduce himself with additional information about his name. The agent must be instructed that the *name* is something it should remember about persons when they introduce themselves. This goal can be achieved by creating an *entity Person* with a field *name*, which we refer to as a "parameter". In the set of sentences associated with the intent, the string that identifies the name should be highlighted by the agent's developer to let the agent learn how to retrieve the person's name inside sentences tagged with the "introduce himself" intent.

Once the sentence intent has been understood and the parameters have been retrieved, the agent must provide a meaningful and appropriate answer. By default, the answer is a fixed sentence whose only variable parts are those related to parameters identified in the input sentence. For example, if the user types *"Hello! My name is Bob"*, the agent can identify the name and answer something like *"Hi Bob! Nice to meet you!"*, but nothing more advanced. This is where *fulfillment* comes into play. The fulfillment is a sort of help from home: if the agent cannot answer messages for some specific intent, those messages are forwarded to an external, specialized source that is waiting. Fulfillment provides a field where the user can insert the URL address of the service to query. The service at that address will be consulted only for those intents that require it; in that case, Dialogflow will wait for the answer and will forward it to the user.

JaCaMo [4,5] *"is a framework for Multiagent Programming that combines three separate technologies, each of them being well-known on its own and developed for a number of years, so they are fairly robust and fully-fledged"* [6].

The three technologies that JaCaMo integrates are:

(1) **Jason** [7], for programming autonomous agents characterized by mentalistic notions, such as beliefs, goals, desires, intentions, and the ability to reason;
(2) **CArtAgO** [8,9], for programming environment artifacts;
(3) **MOISE** [10], for programming multiagent organizations.

In our work, we will use only Jason and (in an indirect way) CArtAgO. By using JaCaMo, we can build a multiagent system along with its environment. JaCaMo agents follow the Beliefs-Desires-Intentions (BDI) model [11,12] and are implemented in Jason, which is a variant of the logic-based AgentSpeak(L) language [13]. The Jason elements that are more relevant for programming one individual cognitive agent are:

- **Beliefs:** the set of facts the agent knows,
- **Goals:** the set of goals the agent wants to achieve,
- **Plans:** the set of pre-compiled, operational plans the agent can use to achieve its goals.

For example, a simple reactive agent that turns on the light either when someone enters the room, when someone issues the "turn light on" voice command, or any other command with the same semantics would need to know

(1) either when someone enters the room or when the "turn light on" command has been issued, and
(2) how to turn on the light.

The first piece of knowledge is a *belief* (something the agent knows about the world either because it is informed of that fact by some other agent via communication, because that fact about the environment is sensed via sensors or provided by artifacts, or because the agent itself generates that piece of knowledge) and the second one is a *plan* (a recipe to achieve some goal via a sequence of operational steps). Considering the first way to turn the light on, triggered by people entering the room, the scenario might involve a sensor that checks the presence of people in the room and triggers the addition or removal of belief from the agent's belief base depending on its measurement. Last, the agent has two *goals*:

1. turn on the light when the right condition is met,
2. turn off the light when no one is there.

We assume that the agent's plans offer instructions on how to turn on the light and how to turn it off. Therefore, depending on its beliefs about the presence of people in the room (that, in turn, depend on the sensor's outcome or on a command being issued), the agent decides to adopt one plan or the other in order to achieve its goal.

Each agent inside the multiagent system has its own source code file written in AgentSpeak(L). If the agent described earlier is called the *light-manager*, there will be a file `light-manager.asl` with the code necessary to describe the agent's behavior. AgentSpeak(L) has a syntax similar to Prolog's one [14], as shown by this example:

```
1  +!turn_on_light(Sensor)
2      :   (Sensor == ``on'')
3      <-  turn_on;
4          !say(``Welcome!'').
```

The triggering event of the plan is *!turn_on_light* as written in row 1, and it depends on one argument, the actual value associated with the *Sensor* variable. In row 2 there is the *context* of the plan after the colon: if the sensor's value is "*on*" then the plan is applicable. Rows 3 and 4 define the actions to be performed, separated by a semicolon.

Actions that are preceded by "*!*" are named achievement goals and trigger the execution of other plans, while those without represent functions offered by artifacts.

The agent needs a way to communicate with the environment. Usually, CArtAgO artifacts are used for this purpose. In our example, we assume to have an artifact controlling

the sensor and another one receiving information from a Dialogflow agent that listens for the user's commands (As anticipated in Section 2, the term 'Dialogflow agent' comes from the Dialogflow documentation. The overloading of the word 'agent' may raise some confusion in this paper, given that VEsNA also integrates JaCaMo agents, which we have built on purpose for VEsNA. We will always use 'Dialogflow agent' to mean the built-in software module that comes with Dialogflow to make it work, and we emphasize here that the agents that we have designed and built are the JaCaMo ones). The bridge between Dialogflow and JaCaMo is built with **Dial4Jaca** [15,16], which provides a set of CArtAgO artifacts that run throughout the execution of the JaCaMo agent. Dial4Jaca starts a listener and has the knowledge to receive and interpret messages from Dialogflow. These messages are not the ones the user writes on the chat but a standardized and structured format that Dialogflow uses when it has to use fullfillments. Such messages contain the intent name, and the entities' parameters identified (e.g., the object the user is talking about in the scene). When the messages are received by a JaCaMo agent, they are parsed and then added to the agent's beliefs using Jason's syntax. Finally, the addition of such new beliefs inside the JaCaMo agent's belief base will cause a reaction driven by the most suitable plan among those in the agent's plan base. The use of Jason and JaCaMo paves the way to supporting those advanced features mentioned in Section 1, namely sophisticated reasoning capabilities and goal-driven planning [17–19], exploitation of formal and semi-formal methods to implement monitoring and safety checks [20–23], explainability, also in connection with Dialogflow thanks to Dial4JaCa [24,25].

Unity [26] is a cross-platform game engine that allows developers to create scenes and add objects to such scenes by dragging and dropping them from a palette to the scene. Objects inserted into a scene can be more or less realistic and may have physical properties or not, depending on what is needed in the application domain of choice. When an object is put inside a scene, a script written in C# can be attached to it. From that script, it is possible to instantiate other objects inside the scene and modify (resp., destroy) those already placed there. Unity may allow users to have a controllable running simulation where elements have some degree of autonomy [27], and to turn that simulated environment into a virtual reality [28]. Unity also supports virtual reality, and its use with headsets for training purposes is well-known and documented [29].

3. Related Work

Virtual Reality and Multiagent Systems (MAS). We open this section with an analysis of the connections between virtual reality and Multiagent Systems (MAS), starting from the idea that existing game engines and simulation platforms are suited to act as platforms for building MASs. This research field, especially when we restrict it to Unity and when agents need learning capabilities, is—surprisingly—still poorly explored.

One of the first works dealing with agents and Unity dates back to 2014 [30] and presents a multiagent system based on Unity 4 that allows simulating the three-dimensional way-finding behavior of several hundreds of airport passengers on an average gaming personal computer. Although very preliminary, that work inspired successive research where—however—the focus was not on the use of a game engine as a general-purpose MAS platform but rather on specific problems that a 3D realistic simulation raises, such as signage visibility to improve pathfinding [31], and on ad-hoc simulations [32–34].

In [35], Juliani et al. move a step toward generalization and present a taxonomy of existing simulation platforms that enable the development of learning environments that are rich in visual, physical, task, and social complexity. In their paper, the authors argue that modern game engines are uniquely suited to act as general platforms and examine Unity and the open-source Unity ML-Agents Toolkit (https://github.com/Unity-Technologies/ml-agents, accessed on 19 January 2023) as case studies.

Before these works were conceived, an interesting strand of papers dealing with Virtual Institutions appeared [36–41]. In A. Bogdanovych et al.'s works, the 3D virtual environment where actions take place in Second Life (https://secondlife.com/, accessed

on 19 January 2023) (Unity was just a newborn project at that time, while Second Life was already gaining attention) and the agents' behavior is coordinated and constrained by electronic institutions. An electronic institution is a composition of roles, their properties and relationships, norms of behavior with respect to these roles, an ontology used by virtual agents for communications with each other, acceptable interaction protocols representing the activities in an institution along with their relationships, and a role flow policy establishing how virtual agents can change their roles. Each virtual agent is characterized by its appearance and its cultural knowledge, namely its beliefs. The Virtual Institutions technology integrates the concepts of the electronic institution with 3D virtual worlds and provides tools for formal specification of institutional rules, verification of their correctness, mapping those to a given virtual world,and enforcing the institutional rules on all participants at deployment.

Virtual Reality and Logic Programming. Given that BDI agents and the AgentSpeak(L) language in particular, borrow many concepts from Logic Programming, in this section, we also consider the relations between Logic Programming and virtual reality, discovering that the literature exists, although it is not vast.

LogiMOO [42] dates back to the late nineties of the last millennium and exploits Prolog for distributing group-work over the internet in user-crafted virtual places where virtual objects and agents live. The virtual reality handled by LogiMOO was, of course, very different from today's one, with no graphical interface at all, but issues related to the object's manipulation and awareness of position were already present.

In SADE (Smart Architectural Design Environment [43]), users can design, configure and visualize architectural spaces in Unity. SADE simplifies the design process by taking into account formal design rules expressed in Prolog, which contain domain knowledge, such as construction law, technical conditions, design patterns, as well as preferences of a designer.

The ThinkEngine [44] is a plugin for Unity that allows developers to program "Brains" using Answer Set Programming, ASP [45,46]. Brains can be attached to Unity non-player-characters; they can drive the overall game logic and can be used, in general, for delivering AI at the tactical or strategic level within the game at hand. One brain can be configured by selecting those sensors that provide inputs to it; a given brain; identifying events that trigger a brain reasoning task; crafting an ASP program that implements the desired decision-making strategy; selecting the actuators and wiring them to the brain, if it is reactive; selecting and programming the set of actions to be performed during the execution of plans if the brain is a planning brain.

Moreover, also connected with our work, at least from the point of view of the logic programming technologies exploited and the final goal to generate virtual environments is the recent work by S. Costantini et al. [47]. In that work, the authors explore the possibility of building a constraint-procedural logic generator for 2D environments and develop a deep Q-learning model based neural network agent able to address the NP search problem in the virtual space; the agent has the goal of exploring the generated virtual environment to seek for a target, improving its performance through a reinforced learning process. Extending that work to the generation of 3D environments seems feasible, and the capability of exploring the generated environment would help us find the best way to position robots in the factory, making sure that human workers can always find a safe path.

Natural Language Processing and MAS. One of the first works combining MAS and natural language processing dates back almost thirty years ago: E. Csuhaj-Varjú described a multiagent framework for generating natural languages motivated by grammar systems from formal language theory [48].

Ten years later, a project about understanding a natural language input using multiagent system techniques was presented by M.M. Aref [49]. The system combined a lexical structural approach and a cognitive structural approach consisting of six modules: speech-to-text, text-to-speech, morphological, semantic, discourse, and query analyzers. These

agents communicate with each other to construct agent subsocieties representing the user input.

In 2004, V.Y. Yoon et al. proposed, in [50], a natural language interface for a multiagent system. They presented the work on an agent-based system called MACS (Multiagent Contracting System) designed to provide advice in the pre-award phase of a defense contract. The analysis of the user input is delegated to a specialized agent that converts natural language into Interagent Communication Language. Thanks to the Bayesian Learning agent, the request is sent to the more appropriate specialty agent that has a set of predefined answers. In that case, the natural language processing is inserted into the multiagent system giving one agent the responsibility of it.

In [51], the authors propose a multiagent system involved in the understanding process. With regard to [50], their design gives the system more freedom, allowing the comprehension of sentences with complex constructions. In addition, the meaning of a sentence is not chosen from a predefined finite set but is generated on the fly.

More recently, and more consistently with our own work, S. Trott et al. described an implemented system that supports deep semantic natural language understanding for controlling systems with multiple simulated robot agents [52]. The system supports bidirectional human-agent and agent-agent interaction using shared protocols with content expressing actions or intentions. Later extensions overcome some limitations due to robot's plans that may be disrupted because of the dynamically changing environment [53].

Virtual Reality and BDI agents. When moving to the integration of BDI-style agents into game engines, we notice that, interestingly, Virtual Institutions share challenges and techniques with both the 'agents and virtual reality' fields and the 'Logic Programming and virtual reality' ones and solve these issues via an ad-hoc implemented library based on the BDI architecture [41]. Although Virtual Institutions do not employ Logic Programming, they adopt a declarative, rule-based approach amenable to reasoning and verification; agents are driven by beliefs represented as logical facts, and the actions they can perform are also represented in a symbolic way.

Outside the Virtual Institutions strand, one of the first works to mention is the PRESTO project [54]. PRESTO (Plausible Representation of Emergency Scenarios for Training Operations) was an industrial R&D project spanning 2013–2016, aimed at creating an all-round development environment for non playing characters' behaviors, for developing BDI agents that could play as 'intelligent opponents' in small-scale military 3D serious games. As reported by P. Busetta et al., the outcomes of PRESTO were significantly constrained by the immaturity of game engines such as Unity, besides the immaturity of platforms for BDI agents.

In the Master Thesis by N. Poli dating back 2018 [55], simple BDI agents were implemented using a lightweight Prolog engine, tuProlog [56], that overcame some limitations of UnityProlog (https://github.com/ianhorswill/UnityProlog, accessed on 19 January 2023), an existing Prolog interpreter compatible with Unity3D. A roadmap to exploit game engines to model MAS that also discusses the results achieved in [55] has been published by S. Mariani and A. Omicini in 2016 [57].

Similarly to work by N. Poli, the work by Marín-Lora et al. [58] describes a game engine to create games as multiagent systems where the behavior specification system is based on first-order logic and is hence closer to a declarative approach than a purely procedural programming language.

Simulations that exploit Unity as the engine and visualization tool and the BDI model as a reference for implementing individual agents have been developed in a few domains, including large urban areas [59], fire evacuation [60], first aid emergency [61], gas and oil industry [62], and bushfires in Australia [63]. Those works are driven by an application to simulate and lack re-usability in other contexts.

The work closest to ours, at least in its final goal of creating a general-purpose extension of Unity that integrates BDI agents, is hence the one by A. Brännström and J.C. Nieves in [64]. There, the authors introduce UnityIIS, a lightweight framework for implement-

ing intelligent interactive systems that integrate symbolic knowledge bases for reasoning, planning, and rational decision-making in interactions with humans. This is performed by integrating Web Ontology Language (OWL)-based reasoning [65] and ASP-based planning software into Unity. Using the components of the UnityIIS framework, the authors developed an Augmented Reality chatbot following a BDI model: beliefs are the agent's internal knowledge of its environment, which are updated during the interaction from new observations; desires are goals that the agent aims to fulfill, which are updated during the interaction by reflecting upon new beliefs; intentions are what goals the agent has chosen to achieve, selected in a deliberation process and used for generating a plan. The belief of the agent is represented in an OWL ontology, as also suggested by other authors in the past [66,67]. The UnityIIS framework enables belief revisions (OWL/ASP file updates) at run time by interweaving the agent's control loop with the Unity game loop. The Unity game loop performs cyclic update iterations at a given frequency. During each frame, external inputs are processed, the game status is updated, and graphics are redrawn.

Albeit sharing some similarities, there are also differences between VEsNA and UnityIIS. The main one is that UnityIIS does not integrate JaCaMo with Unity: the BDI model is used as a reference but is not implemented using a standard agent programming language as Jason. Rather, UnityIIS relies on OWL and ASP as languages for modeling knowledge and declarative behavior of cognitive agents. The second one is that the chatbot described in [64] is an example of the application of UnityIIS, in the same way as factory automation is an example of the application of VEsNA. In VEsNA, the chatbot is one of the three pillars of the framework and not just an application. What we might borrow from the UnityIIS model is the adoption of OWL to model knowledge and reason about it in an interoperable, portable way. What UnityIIS might borrow from VEsNA is the closer integration of a standard framework for BDI agents into the system, JaCaMo, along with all its libraries and add-ons, rather than the adoption of more generic logic-based languages such as ASP and OWL.

4. VEsNA Design and Implementation

In this section, we provide some details on VEsNA design and implementation, and we describe how a user may interact with it. Figure 2 reports a high-level architectural scheme of VEsNA while Figure 3 shows a sequence diagram of the main execution steps. Users that want to try out the framework have to download VEsNAfrom https://github.com/driacats/VEsNA (accessed on 19 January 2023) and

1. create a scene on Unity, import the `listener.cs` script file, and attach it to the floor object;
2. import the Dialogflow zip file on a new Dialogflow agent;
3. move to the *JaCaMo* directory and launch the multiagent system with `gradle run`.

After the execution of these steps, VEsNA is ready for use. In the proposed scenario, the scene represents a factory, and the objects are robots but VEsNA works in every context. In order to use custom objects users have to add sentences dealing with those objects in Dialogflow.

On the VEsNA screen, the user can find a chat and a pre-built scene made with Unity representing whatever he/she needs, based on the application domain; the scene has at least one empty floor. The user can describe how the scene is organized via the chat, using natural language (Thus far we have run experiments with English only, but given that Dialogflow natively supports multiple languages, https://cloud.google.com/dialogflow/es/docs/agents-multilingual, accessed on 19 January 2023, letting the user talk in his/her own language should also be possible) and so telling the VEsNA agent what to do. For example, the user could say (thus far, interactions are via a textual interface only; nonetheless, to plug a voice-to-text component into the VEsNA control flow should not raise any technical issues) *"Add that object in the scene!"*. This would make the object appear inside the scene. However, the translation from the message to graphics animation requires various intermediate steps.

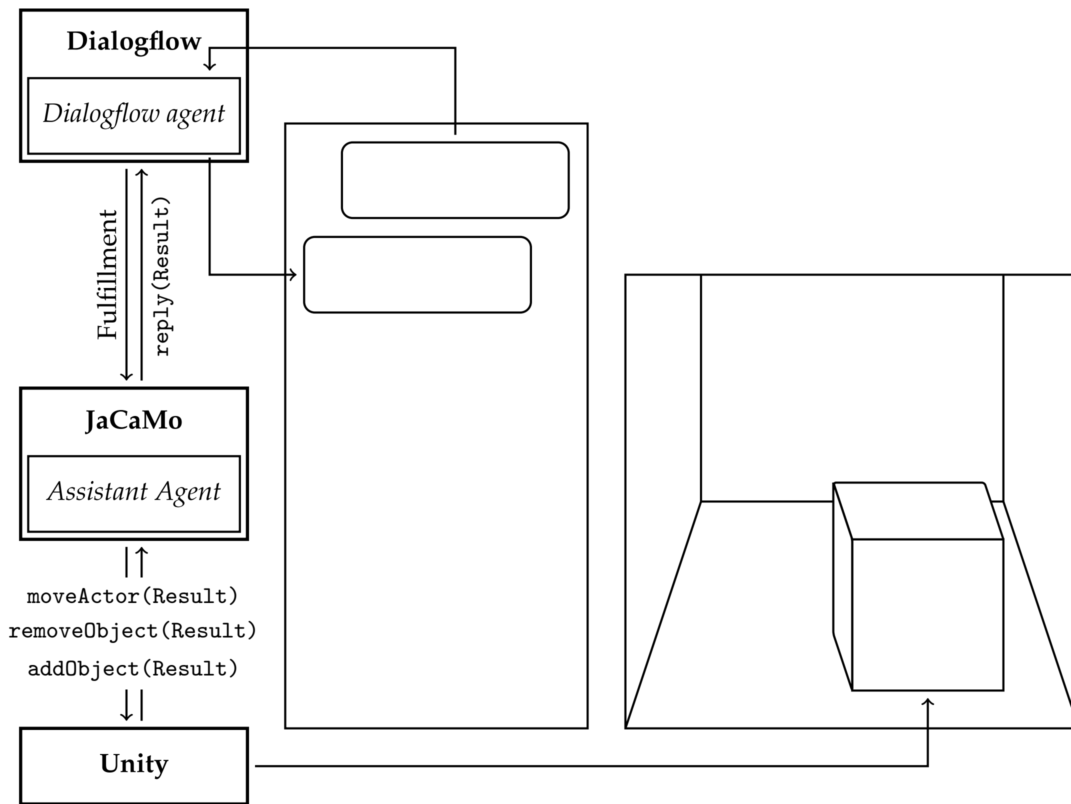


Figure 2. VEsNA architectural scheme.

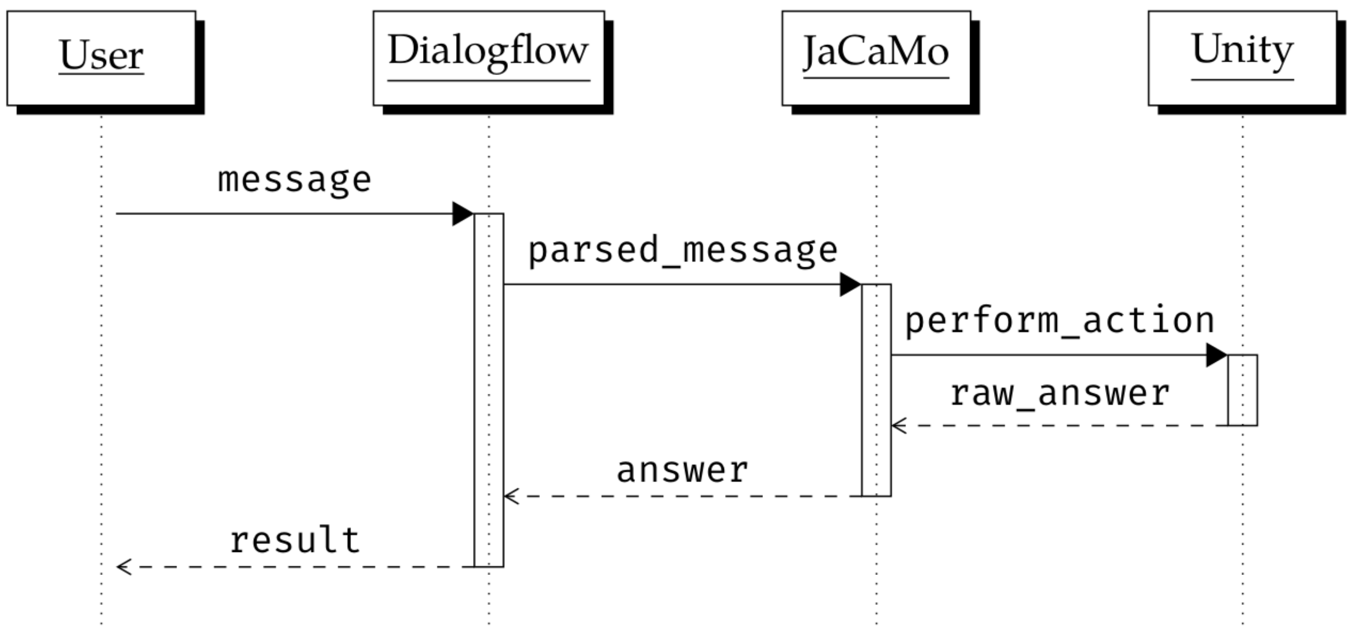


Figure 3. Sequence Diagram of the execution. The actual functions called change depending on the user input. This figure represents a general skeleton of the interactions among VEsNA components.

The Dialogflow agent knows two entities: the *object* and the *employee*. The object entity has three fields: one for the object name, one for the position on the vertical axis, and one for the position on the horizontal axis. The object name field is necessary to describe an object entity, the two fields for the positions are additional knowledge. The employee entity instead has two fields: one for the employee name and the other for the direction

in which the user wants to move it. In this case, both fields are necessary to describe the employee entity.

When a message is received, the Dialogflow agent understands if the action the user wants to perform—namely the user's intent—is an addition, a removal, or a motion.

If the intent is an addition, the Dialogflow agent identifies the name of the object the user wants to add along with its position in the scene. This information is automatically extracted from the user's sentence through natural language processing. In more detail, the position can be described by the user in two different ways. The first is by using **global positioning** on the entire space available. In the current implementation, the space is divided into nine available positions. Hence, the user can say *"Ok agent, put that object in front on the right"* and the agent will understand what it is expected to do.

While the global positioning may work well with small scenarios, it is likely to be too coarse grained with big ones; however, this may represent the "first" positioning of objects inside the scene. Indeed, other objects may be added later on by referring to relative positions (with regard to already present objects in the scene). The user can add new objects that **reference** those already in scene. In that case, it would be possible to say *"Ok, I told you that there is that object on the right. Well, behind it there is this other object"*. This is performed by giving a unique name to each object added inside the scene. Such a name can be used any time it is needed to identify an object as a reference: when a user adds an object, the answer contains the *reference name* for it.

Furthermore, while Dialogflow can identify every object name inserted by the user, an error will be generated if the name does not correspond to one of the available objects. To make the system work, the user has to insert the name of the object in Unity in such a way that Dialogflow can recognize it. For the moment, this also applies to identifiers of already added objects: they must be among those provided by Unity and communicated to the user. We are working to overcome this limitation by introducing scopes that will allow the user a more natural description of the scene, making prior knowledge available and supporting the ability to refer to the last added objects without having to remember their identifiers.

If, instead, the intent of the user is to remove an object, the Dialogflow agent identifies the *reference name* of the object to be removed inside the sentence. In this case, the name is the only piece of information needed. When the message is delivered, the object corresponding to the *reference name* is destroyed. If the referenced object does not exist, an error message is returned.

The last intent that can be identified by Dialogflow is motion. In this case, the Dialogflow agent identifies two pieces of information inside the sentence: the *reference name* of the object to move and the direction to follow. The user can write sentences like "*Move employee1 to the right*" and the employee will make a step to the right. Thus far, motion is not intended for objects but for employees only. If the user tries to move an object instead of an employee, an error message is generated: motion needs some specific instructions on the Unity side that are implemented only for the employee game object.

From a technical point of view, once the user's sentence has been processed by Dialogflow, the resulting intent is generated. In the case of the addition of a new object inside the scene, the intent *AddObject* is used and trained to identify *who* is the object to add and *where* to add it. In the case of removal, the intent *RemoveObject* is used and trained to identify *who* is the object to remove. The motion makes use of *MoveIntent*, which is trained to identify *who* is the employee to move and in which *direction*. All these intents have fulfillment flagged so that when a sentence is traced back to that intent, a request is propagated to the address where the JaCaMo agent is listening. Only intents with the fulfillment flagged are propagated to the JaCaMo agent; all other intents are not propagated, which means are not in need of an agent to be handled (e.g., sentences not concerning the scene).

As a minor technical aspect, given that Dialogflow runs on the cloud while JaCaMo runs locally on the user's machine, the local machine must be made accessible from outside its LAN. This is achieved by using **ngrok** (https://ngrok.com/, accessed on 19 January

2023), which creates an HTTP tunnel, making a specific private port of the machine publicly available. Since Dial4Jaca creates a listener on the classic 8080 port, we use ngrok to make such listener accessible from Dialogflow. If another framework for developing the chatbot were used, such as the already mentioned Rasa that we integrated into the new VEsNA release or the viral ChatGPT (https://chat.openai.com accessed on 19 January 2023), this external communication might be avoided. This aspect is indeed peculiar to Dialogflow rather than to VEsNA. In the new VEsNA release, the adoption of Rasa has solved not only the bottleneck due to the ngrok HTTP interface but also all the cybersecurity issues that it might have raised.

After the intent has been generated by Dialogflow, it is propagated to the JaCaMo multiagent system along with the needed pieces of information. Currently, the JaCaMo multiagent system is composed of only one agent, the *assistant agent*, but in the future, it will contain one agent for each employee on the scene, and it will provide intelligence to the entire system: intelligent cognitive agents will allow virtual employees to decide what to do, where to move and how to do things with autonomy, trying to get a given goal provided by the user completed. In the first VEsNA releases, goals will be very simple and precise as the ones that are introduced in this paper ("*make a step*", "*take that object*", "*turn on the robot*", etc.), but in our plans for the future, they will become more and more sophisticated and less operational, still being expressed at a high level, using natural language. The intelligence of the system will make the simulations more realistic and less predetermined since there will not be a predefined sequence of actions, but each employee will be free to follow different paths to obtain the same result.

The JaCaMo *assistant agent* is responsible for the reception, the computations, and the forwarding of the request coming from the Dialogflow agent. For each intent that the Dialogflow agent can recognize, there is a Jason *plan* in the assistant agent's Jason code that is triggered when the corresponding message is received. Depending on the requested action, the assistant agent acquires and organizes the necessary information and forwards them to Unity in a structured way. In particular:

- if the requested action is an addition, the assistant agent sends Unity the name of the object and the position on the horizontal axis and on the vertical axis.
- if it is a removal, the assistant agent sends the *reference name* of the object to remove and a *removal* flag.
- if it is a motion, the assistant agent sends the *reference name* of the employee to move, the direction in which it should move, and a *motion* flag.

For the moment, communication between the assistant agent and Unity is implemented using the HTTP protocol as follows:

1. `http://unity_address:port/object_name/pos_x/pos_y`
2. `http://unity_address:port/remove/reference_name`
3. `http://unity_address:port/move/reference_name/direction`

Respectively for addition, removal, and motion.

Unity has an asynchronous listener waiting for requests and receives all the information sent by the assistant agent as strings. There are three possible string structures that can be handled by the Unity listener, one for each command that has been designed. In the case of an addition, the listener expects up to three instructions, and at least one (the name of the object) is necessary. If users do not provide any information on the position on one or both axis, a center position will be inferred. In this case, inside the string, there is no clear instruction to add the object. In the case of removal and motion, the first piece of the string describes the operation requested, respectively `remove` and `move`. The Unity listener splits the string on the "/" character producing an array of strings, and checks the head of the array. If the word is "*remove*", then it only looks at the second element of the array that contains the name of the object. If the word is "*move*", the Unity listener looks in positions `1` and `2` and extracts the name of the employee and the direction. If, instead that position does not contain one of the two instructions, the required operation is an addition, and the

first position contains the name of the object to be added, and if positions 1 and 2 contain some values, they are the values for horizontal and vertical positions. If these positions contain nothing, the listener writes "`center`" in the corresponding variables.

For example, if the user writes in the chat *"Put that object on the right"*, what the Unity listener receives is `''that object, centre, right''`. After that, Unity converts the received string into a vector that describes the object's position inside the scene. In the case of global positioning, the resulting conversion is simple and is computed with regard to the size of the scene. However, for relative positioning, Unity has to look for the object used as a reference to compute the new position with regard to the latter. The global position is not hard-coded but is computed with respect to the floor size. This is why the floor has to be a single object: Unity uses its dimensions in order to compute where the center is, where the right is, where the behind is, and so on. All positions are computed starting from the center and moving in the desired direction with an offset of 25% of the entire floor side length, as shown in Figure 4.

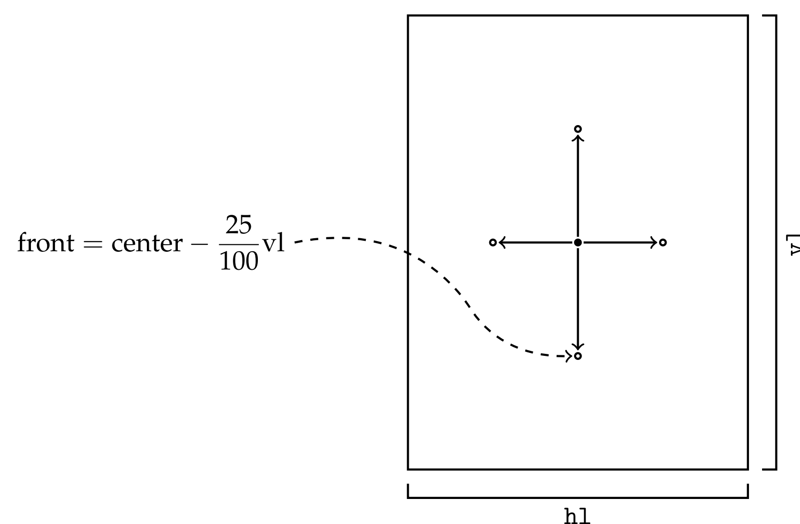


Figure 4. Computation of global positions.

The relative position computation instead

1. looks for the object referenced by the user;
2. gets its position;
3. computes the new position summing up the object one with an offset in the given direction. The offset is computed as 10% of the correspondent axis.

The procedure is shown graphically in Figure 5.

Once the position for the new object is found, a physical check is performed on the Unity side by exploiting native Unity's colliders to make sure that the selected place has not already been taken by another object. If the position is already taken, an error message is sent back to the assistant agent. Otherwise, the object is added, and a "done" message is sent back to the assistant agent containing the unique name of the object just added.

When Unity receives a removal instruction, it simply looks for the object with the reference name given and destroys it. If other objects had been previously added in a position relative to the removed one, they would remain on the scene and will maintain their positions. If the object does not exist, an error message is returned to the assistant agent.

When the user gives the instruction to move an employee, the Unity listener looks for the employee with the given *reference name* for the removal, and it makes it move in the direction it is already walking in. The motion is implemented inside a script that has to be attached to the employee game object. The game object itself has some predefined animations that can be used and that make the simulation look more natural. Directions

are fixed and are not relative to the employee, and, for the moment, there is no check on the content of the space in the chosen direction: if something is in the path, the employee will try, in any case, to perform the assigned motion and the consequences are handled by the physical constraint checker of Unity.

$$\text{right of } \texttt{obj0} = \text{center of } \texttt{obj0} + \frac{\text{bbox}}{2} + \frac{10}{100}\text{length}$$

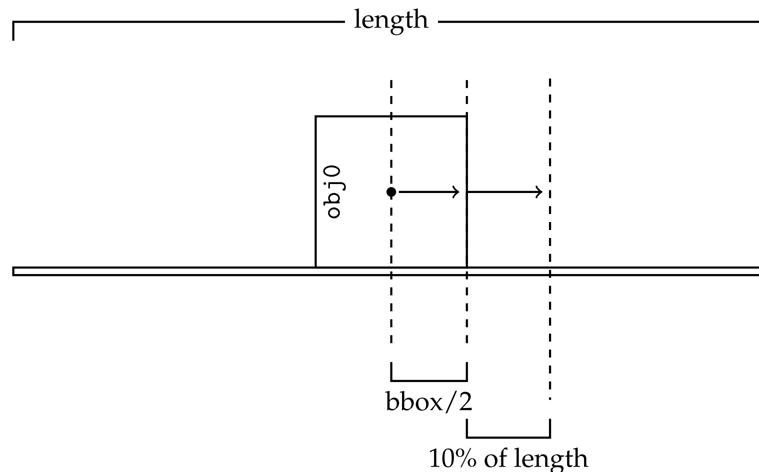
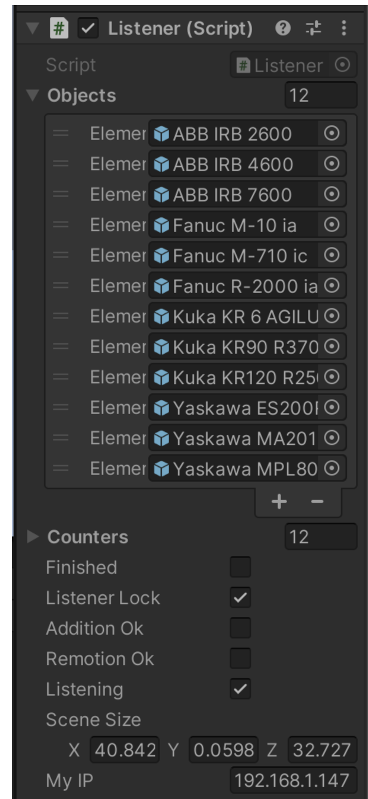


Figure 5. Computation of relative positions.

The script attached to the Unity floor looks like this in the editor:



- The objects that are stored in a dynamic list, the user can add any object therein. The names that are displayed are those the user must use. If, for example, users have two tables and call the two game objects "*small table*" and "*big table*", then they can write on the chat "*Insert the big table in the scene*", but if they write "*Put a table in the scene*", nothing will happen. The names, therefore, are up to the users.
- The other dynamic list is the counters one. This list is created by the script itself and has one entry for each object provided by the user. By using this list, the script is able to generate unique names: for each object added, it concatenates its name with the value of the corresponding counter and increments it. For the moment, there is no way to re-use the numeric identifiers associated with objects that have been removed: the counter goes on increasing its value by one.
- The boolean values (`Finished`, `Listener Lock`, `Addition Ok`, `Remotion Ok`, `Listening`) are technical flags used to manage the HTTP listener.
- The Scene Size is a 3D vector with the dimensions of the floor (Y component is near 0).
- My IP is the local address of the machine, and it is used to configure the HTTP listener.

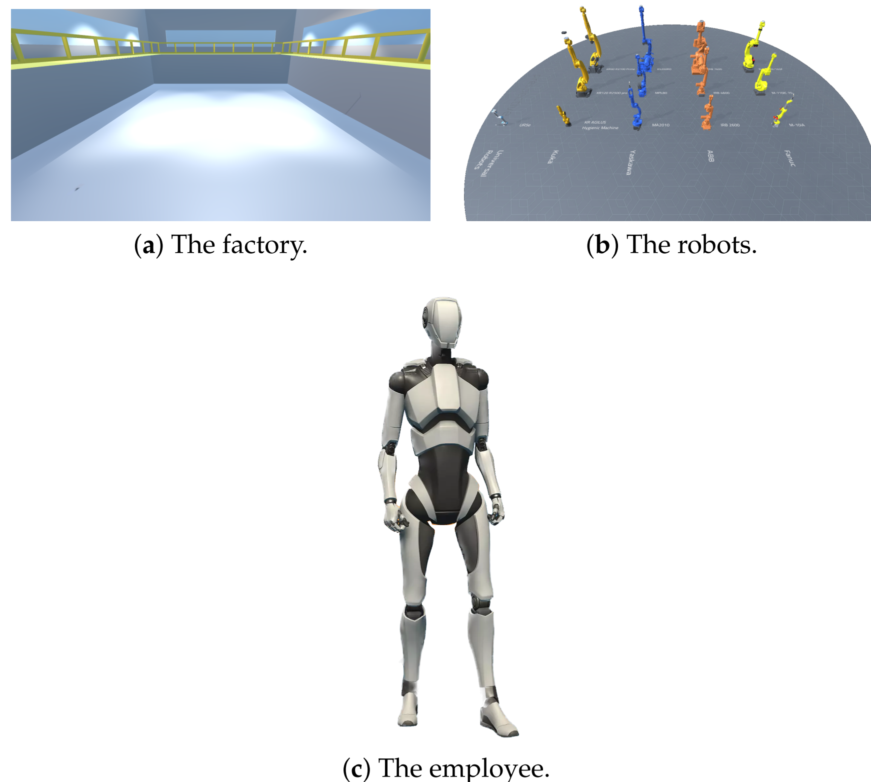
Once on the Unity side, the action has been completed, feedback is sent to the JaCaMo assistant agent that, in turn, sends back a positive answer to Dialogflow, and a message

is displayed to the user via the chat. Finally, the user may decide to stop or start another iteration by adding a brand new object to the scene.
The response time, on average, is less than 1 s to see a reaction in the virtual environment and around 1.1 s to have the answer in the chat. From a user experience point of view, the delay is not visible.

The entire framework will work on every computer that can run Unity fluently. Unity recommends Intel i5 CPU or better with at least 8 GB of RAM, which are also the minimum hardware requirements for VEsNA to work efficiently. For the experiments, we worked with a i7-8565U CPU and 20 GB of RAM.

5. Using VEsNA in the Factory Automation Domain

As a concrete example of use, let us consider the scenario introduced in Section 1. Let us suppose that VEsNA's user is the owner of a small-medium factory where extended-reach welding robots (robots that can move their only arm in almost all directions but cannot wander through the factory, see Figure 6b) must be positioned in an optimal way to carry some automation work out. When VEsNA is first run, an empty model of the factory in Unity is available to the user, ready to be modified (Figure 6a).



(**a**) The factory. (**b**) The robots.

(**c**) The employee.

Figure 6. Model in Unity.

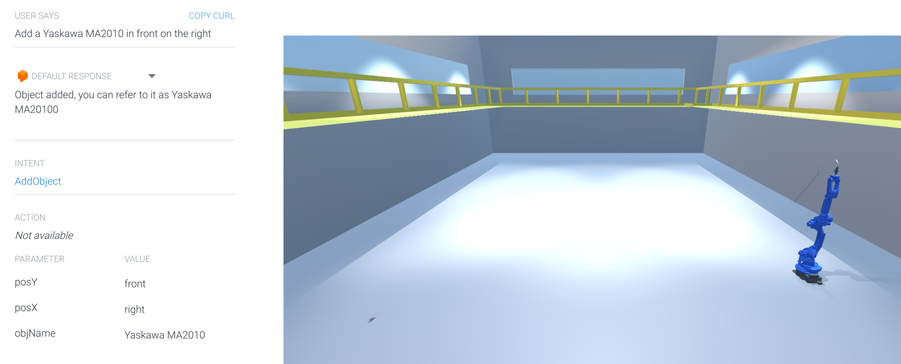
In this case, the user is interested in simulating, in Unity, the outcome of putting robots and employees in different positions to devise the optimal one for robots and to make sure that employees can safely work with them. However, the user might not have the programming skills to use Unity. This is when VEsNA comes into play to let the user position objects in the Unity scene by using natural language. Let us suppose that the user wants to position a Yaskawa MA2010 robot (https://www.robots.com/robots/motoman-ma2010, accessed on 19 January 2023) in the front-right global position. The user may type *"Add a Yaskawa MA2010 in front on the right"* in the VEsNA chat.

The sentence in the chat (Figure 7a) automatically creates the scene in Figure 7b. The machinery behind the result displayed in Figure 7b is the following. The Dialogflow agent

brings the *"Add a Yaskawa MA2010 in front on the right"* back to the intent *AddObject*. An entity **Object** that has parameters

1. **Name**, the name of the object to be positioned, the robot in this case;
2. **PosX**, in global positioning, is the horizontal position, in relative positioning contains the relative position (*left of*, *right of*, *behind*, *in front of*);
3. **PosY**, in global positioning, is the forward position, in relative positioning contains the name of the object the user refers to.

In the sentence the Dialogflow agent finds the name *"Yaskawa MA2010"*, the posX *"right"* and the posY *"front"* and sends a request to the link provided in the fulfillment (where the JaCaMo agents awaits).



(**a**) The Dialogflow chat. (**b**) The graphical result.

Figure 7. Example of object addition with global coordinates.

Dial4Jaca receives the information from the Dialogflow agent and adds the following belief to the assistant agent:

```
1  request(
2      ‘‘undefined’’,
3      ‘‘5b485464-f275-42ab-853e-59514b115359-cf898478’’,
4      ‘‘AddObject’’,
5      [
6          param(‘‘posX’’,‘‘right’’),
7          param(‘‘posY’’,‘‘front’’),
8          param(‘‘objName’’,‘‘Yaskawa MA2010’’)
9      ],
10     ...
```

The *assistant agent* receives the message, verifies that it matches with the triggering event of the following plan (after some syntactic reworking), checks that the plan's context is verified, and executes the plan's body:

```
5  +!answer(RequestedBy, ResponseId, IntentName, Params, Contexts)
6  :   (IntentName == ‘‘AddObject’’)
7  <-  !getParameters(Params);
8      addObject(Result);
9      reply(Result).
```

The assistant agent performs three simple steps:

1. gets the information on the position of the object;
2. calls the `addObject` function defined by the CArtAgO artifact that provides an interface for Unity; the variable *Result* contains the result of the action provided by the Unity listener;
3. sends *Result* back to Dialogflow, thanks to the *reply* action in the plan's body.

The `addObject` function provided by the artifact sends an HTTP request to the 8081 port on the same machine and waits for an answer (we remind readers that on port 8081, Unity is listening for messages). The HTTP request is assembled as follows

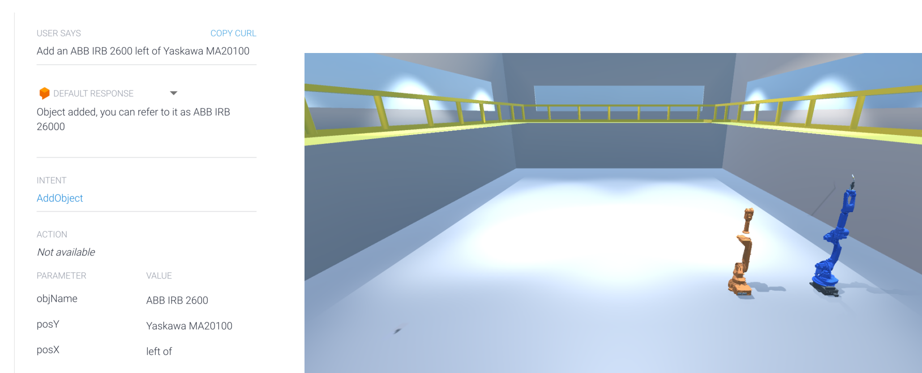
$$\texttt{http://localhost:8081/}\underbrace{\texttt{Yaskawa\%20MA2010}}_{\texttt{obj name}}/\underbrace{\texttt{right}}_{\texttt{posX}}/\underbrace{\texttt{front}}_{\texttt{posY}}$$

On the Unity side, the script attached to the factory has a listener on that port and receives this piece of information. It then converts the location from a couple of strings (in our example $(right, front)$) into a 3D vector of coordinates. This procedure is described in Figure 4. In this specific case, the requested position is at the front of the right, and it is computed as a combination of *front* and *right* starting from the center. When the Unity listener has a position, it checks that it is not already occupied by something else and eventually adds the object to the scene.

Let us now assume that the user wants to continue the interaction with VEsNA. For instance, the user could ask to add a new robot to the left of the one inside the scene. Since there is already another robot inside the scene, the user can add objects and position them by reference. In this case, the user could write, *"Add an ABB IRB 2600 left of Yaskawa MA20100"*. The request sent from the *assistant agent* to Unity will be

$$\texttt{http://localhost:8081/}\underbrace{\texttt{ABB\%20IRB\%202600}}_{\texttt{obj name}}/\underbrace{\texttt{left\%20of}}_{\texttt{posX}}/\underbrace{\texttt{Yaskawa\%20MA20100}}_{\texttt{posY}}$$

The `posX` parameter, in this case, tells where to put the new object with respect to the one referenced in `posY`. The result obtained after this request is shown in Figure 8b:



(**a**) The Dialogflow chat. (**b**) The graphical result.

Figure 8. Example of object addition with relative coordinates.

Let us now suppose that the user changes his/her mind and decides to remove the robot positioned at the beginning. The user could type *"Remove the Yaskawa MA20100"* and the result would be the one shown in Figure 9, left, where the blue arm is no longer part of the scene. The Jason plan used by the *assistant agent* in this case will be

```
10  +!answer(RequestedBy, ResponseId, IntentName, Params, Contexts)
11  :   (IntentName == ‘‘RemoveObject’’)
12  <-  !getParameters(Params);
13      removeObject(Result);
14      reply(Result).
```

The *removeObject* request from the *assistant agent* to the Unity listener would turn out to be:

$$\texttt{http://localhost:8081/}\underbrace{\texttt{remove}}_{\texttt{instr}}/\underbrace{\texttt{Yaskawa\%20MA20100}}_{\texttt{obj_name}}$$

The first piece tells Unity that the operation to be performed is removal, and the second one is the *reference name* of the object to be removed.

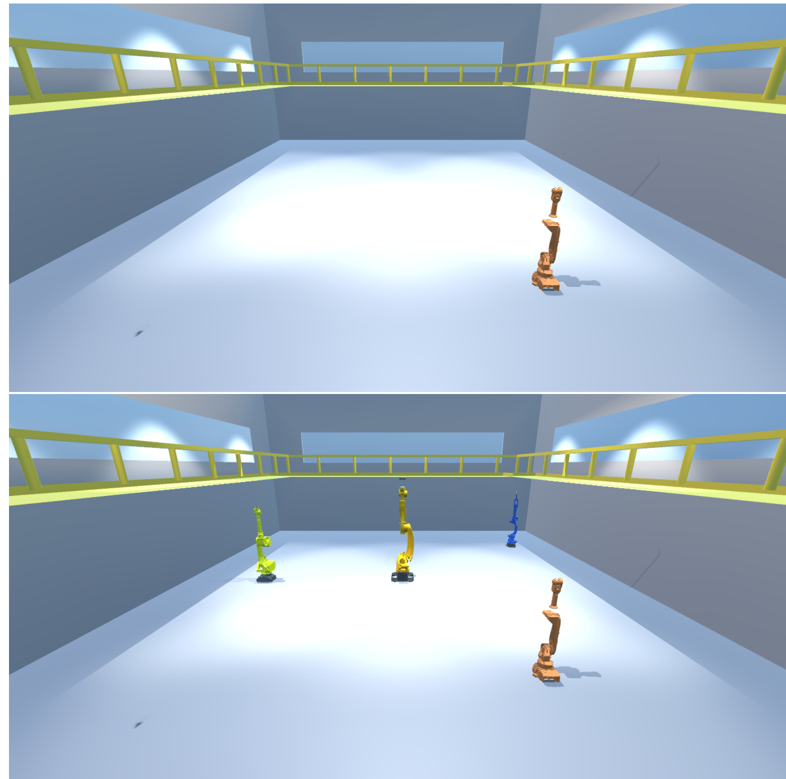


Figure 9. Example of object removal (**above**) and result of further interactions (**below**).

After a few iterations, all the robots would be positioned in the factory as the user desires, and the result might look like the scene shown in Figure 9, right.

Finally, let us suppose that the user wants to check the practical usability of the created configuration. In order to do it, the user will add an employee in the factory by writing "*Add an employee in the centre*". As for the other objects, the employee can be inserted using either the global or the relative reference system, and it will appear in the scene. The user can now make the employee move step by step, simulating the real motion and checking the safety and convenience of the configuration. In this case, the *assistant agent* uses the following Jason plan:

```
15  +!answer(RequestedBy, ResponseId, IntentName, Params, Contexts)
16      :   (IntentName == ‘‘MoveActor’’)
17      <-  !getParameters(Params);
18          moveActor(Result);
19          reply(Result).
```

If, for example, the user writes "*Make employee1 move on the left*" in the chatbox, the *assistant agent* will identify the *Employee object* in the sentence that has two fields:

1. **Employee Name:** the reference name of the employee to move;
2. **Direction:** the direction to follow when moving.

Once the two pieces of information are identified, the assistant agent executes this request on the Unity listener:

$$\texttt{http://localhost:8081/}\underbrace{\texttt{move}}_{\texttt{instr}}/\underbrace{\texttt{Employee1}}_{\texttt{obj_name}}/\underbrace{\texttt{right}}_{\texttt{dir}}$$

As for the removal, the instruction contains not only the *reference name* and the *direction* but also the instruction `move` as the first argument. The result is shown in Figure 10.

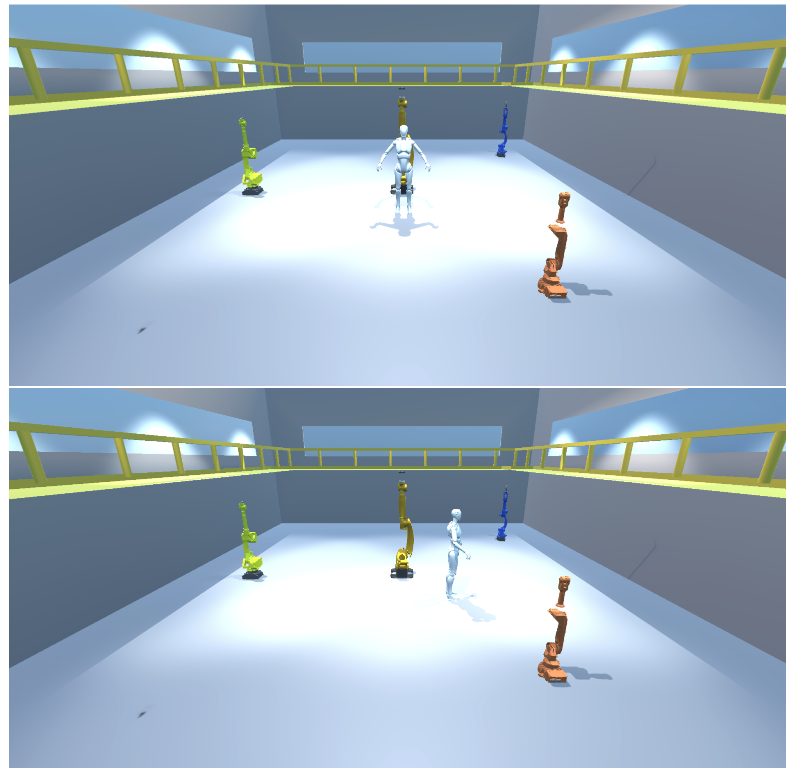


Figure 10. Example of employee addition (**above**) and employee movement (**below**).

VEsNA stays awake and responsive until users say that they have completed their work.

6. Conclusions and Future Work

We presented VEsNA, a general-purpose framework to allow users with no programming skills to populate simulations in Unity through natural language, heavily exploiting agent-based technologies that might help in reasoning on scenarios, verifying the feasibility of choices, and providing explanations. We showed how starting from natural language sentences, VEsNA can automatically handle the creation, addition, and removal of objects and employees and the motion of employees into a simulated scene in Unity. Specifically, this is obtained in VEsNA through the combination of three different frameworks: Dialogflow, JaCaMo, and Unity. These frameworks support natural language analysis, scene construction, and graphic representation, respectively. To the best of our knowledge, VEsNA is the first framework that combines all these three aspects together.

For future developments, we are exploring additional uses for the objects in the simulated scene. In particular, instead of having static objects, we might have active objects as agents. This would help make the simulation more realistic, engaging, and amenable to a more precise risk analysis. In the factory automation scenario, besides employees that will be represented as cognitive agents in the factory to experiment with the movements and the interactions with objects, robots could be implemented as agents as well, with some degree of autonomy (Many standards define the level of autonomy of artifacts and robotic systems; see, for example, the SAE Levels of Driving Automation, https://www.sae.org/blog/sae-j3016-update, accessed on 19 January 2023, that goes from Level 0 (no automation) to Level 5 (full automation): some robots in the factory might reach high automation levels, and hence their implementation as intelligent agents would be very appropriate). Given that robots would not need to be driven by mentalistic/cognitive features as the simulated employees, we might implement them directly using the language supported by the graphical environment tool, for example, C# or UnityScript for Unity. The autonomy of the simulated robots and the autonomy of the simulated employees would allow the developer of the simulation to explore possibly unexpected and harmful

combinations of autonomous behaviors. This is exactly why we believe that VEsNA may be useful: to discover threats before they take place.

We are also exploring the possibility of integrating other game engines such as Unreal (https://www.unrealengine.com/, accessed on 19 January 2023) and Godot (https://godotengine.org, accessed on 19 January 2023) in VEsNA. Both support virtual reality natively as Unity. Particular attention is given to Godot, which is open-source and lighter than Unreal and Unity and can be a suitable alternative for a desktop application with a small scene. The actual exploitation of virtual reality in the factory automation domain, thanks to VEsNA, is a goal that will require some more time to be achieved. Thus far, we took the possibility to "enter the graphical rendering of the factory thanks to tools for virtual reality" as a VEsNA hard requirement, and indeed we selected graphical tools that do support it, but we have not tested this feature yet. Virtual reality would guarantee employees not only a better and more immersive experience but would also allow them to interact directly with the robots to learn how to use them in a safe and controlled situation when, for example, they are not working or they are working at a low autonomy level.

One relevant aspect we will take into account, as soon as the virtual reality component will be better developed and tested, is to consider a way to reduce the power consumption of virtual reality headsets, following, for example, the approach presented by Z. Yang et al. [68].

Being general-purpose, VEsNA is not limited to the factory automation domain. Indeed, one of our original reasons for the development of VEsNA was to create a tool for interactive, agent-based theatrical story-telling along the lines of [69–74]. In that domain, the presence of cognitive, goal-driven, nonplaying characters in the Unity scene able to perform some actions on their own provides a strong motivation for the integration of JaCaMo and Unity.

Finally, from a more practical perspective, we are also considering the exploration of additional frameworks to improve the VEsNA usability and accessibility. As already mentioned, other options are available on the sentence analysis side, such as Rasa. Rasa is a valid alternative to Dialogflow because it is open-source and can be run locally, differently from Dialogflow, which instead requires to be executed in the cloud. In the first VEsNA release, we opted for Dialogflow for its simplicity and because we had already used it. Further, Dialogflow was natively supported by Dial4JaCa. However, if used in scenarios where privacy might be an issue, the possibility of having all software running locally could be a necessary feature. Rasa solves these issues, and we will definitely substitute Dialogflow in the forthcoming VEsNA releases.

Author Contributions: Conceptualization, A.G. and V.M.; methodology, A.G. and V.M.; software, A.G.; validation, A.G.; writing—original draft preparation, A.G.; writing—review and editing, V.M.. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

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

References

1. Google. DialogFlow. Available online https://cloud.google.com/dialogflow/ (accessed on 19 January 2023).
2. Austin, J.L. *How to Do Things with Words*; William James Lectures, Oxford University Press: Oxford, UK, 1962.
3. Searle, J.R. *Expression and Meaning: Studies in the Theory of Speech Acts*; Cambridge University Press: Cambridge, UK, 1979. [CrossRef]
4. 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]
5. Boissier, O.; Bordini, R.H.; Hübner, J.; Ricci, A. *Multi-Agent Oriented Programming: Programming Multi-Agent Systems Using JaCaMo*; MIT Press: Cambridge, MA, USA, 2020.
6. Boissier, O.; Bordini, R.H.; Hübner, J.H.; Ricci, A.; Santi, A. JaCaMo Project. Available online: http://jacamo.sourceforge.net/ (accessed on 19 January 2023).

7. Bordini, R.H.; Hübner, J.F.; Wooldridge, M. *Programming Multi-Agent Systems in AgentSpeak Using Jason*; John Wiley & Sons: Hoboken, NJ, USA, 2007.
8. Ricci, A.; Viroli, M.; Omicini, A. CArtAgO: A framework for prototyping artifact-based environments in MAS. In Proceedings of the International Workshop on Environments for Multi-Agent Systems, Hakodate, Japan, 8 May 2006; Springer: Berlin/Heidelberg, Germany, 2006; pp. 67–86.
9. Ricci, A.; Piunti, M.; Viroli, M. Environment programming in multi-agent systems: An artifact-based perspective. *Auton. Agents Multi Agent Syst.* **2011**, *23*, 158–192. [CrossRef]
10. Hübner, 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]
11. Rao, A.S.; Georgeff, M.P. BDI agents: From theory to practice. In Proceedings of the First International Conference on Multiagent Systems (ICMAS), San Francisco, CA, USA, 12–14 June 1995; Volume 95, pp. 312–319.
12. Georgeff, M.; Pell, B.; Pollack, M.; Tambe, M.; Wooldridge, M. The Belief-Desire-Intention model of agency. In Proceedings of the International Workshop on Agent Theories, Architectures, and Languages, Paris, France, 4–7 July 1998; Springer: Berlin/Heidelberg, Germany, 1998; pp. 1–10. [CrossRef]
13. Rao, A.S. AgentSpeak (L): BDI agents speak out in a logical computable language. In Proceedings of the European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands, 22 January 1996; Springer: Berlin/Heidelberg, Germany, 1996; pp. 42–55. [CrossRef]
14. Sterling, L.; Shapiro, E. *The Art of Prolog—Advanced Programming Techniques*, 2nd ed.; MIT Press: Cambridge, MA, USA, 1994.
15. Engelmann, D.; Oliveira, J.D.; Borges, O.T.; Krausburg, T.; Vivan, M.; Panisson, A.R.; Bordini, R.H. Dial4Jaca. Available online: https://github.com/smart-pucrs/Dial4JaCa (accessed on 19 January 2023).
16. 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.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2021; Volume 12946, pp. 77–88. [CrossRef]
17. Wooldridge, M.J. *Reasoning about Rational Agents*; Intelligent Robots and Autonomous Agents; MIT Press: Cambridge, MA, USA, 2000.
18. Ricci, A.; Bordini, R.H.; Hübner, J.F.; Collier, R.W. AgentSpeak(ER): An Extension of AgentSpeak(L) improving Encapsulation and Reasoning about Goals. In Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS, 2018, Stockholm, Sweden, 10–15 July 2018; André, E., Koenig, S., Dastani, M., Sukthankar, G., Eds.; International Foundation for Autonomous Agents and Multiagent Systems: Richland, SC, USA, 2018; pp. 2054–2056.
19. de Oliveira Gabriel, V.; Panisson, A.R.; Bordini, R.H.; Adamatti, D.F.; Billa, C.Z. Reasoning in BDI agents using Toulmin's argumentation model. *Theor. Comput. Sci.* **2020**, *805*, 76–91. [CrossRef]
20. Bordini, R.H.; Fisher, M.; Pardavila, C.; Wooldridge, M.J. Model checking AgentSpeak. In Proceedings of the Second International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS 2003), Melbourne, VIC, Australia, 14–18 July 2003; pp. 409–416. [CrossRef]
21. Ancona, D.; Drossopoulou, S.; Mascardi, V. Automatic Generation of Self-monitoring MASs from Multiparty Global Session Types in Jason. In Proceedings of the Declarative Agent Languages and Technologies X—10th International Workshop, DALT 2012, Valencia, Spain, 4 June 2012; Baldoni, M., Dennis, L.A., Mascardi, V., Vasconcelos, W.W., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2012; Volume 7784, pp. 76–95. [CrossRef]
22. Ferrando, A.; Dennis, L.A.; Ancona, D.; Fisher, M.; Mascardi, V. Recognising Assumption Violations in Autonomous Systems Verification. In Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, 10–15 July 2018; André, E., Koenig, S., Dastani, M., Sukthankar, G., Eds.; International Foundation for Autonomous Agents and Multiagent Systems: Richland, SC, USA, 2018; pp. 1933–1935.
23. 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 AREA 2022, the Second Workshop on Agents and Robots for Reliable Engineered Autonomy, Vienna, Austria, 23–25 July 2022.
24. 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.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2021; Volume 13073, pp. 3–17. [CrossRef]
25. Ferreira, C.E.A.; Panisson, A.R.; Engelmann, D.C.; Vieira, R.; Mascardi, V.; Bordini, R.H. Explaining Semantic Reasoning using Argumentation. In Proceedings of the 20th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS), L'Aquila, Italy, 13–15 July 2022.
26. Unity Technologies. Unity. Available online: https://unity.com/ (accessed on 19 January 2023).

27. Biagetti, A.; Ferrando, A.; Mascardi, V. The DigForSim Agent Based Simulator of People Movements in Crime Scenes. In Proceedings of the Advances in Practical Applications of Agents, Multi-Agent Systems, and Trustworthiness. The PAAMS Collection—18th International Conference, PAAMS 2020, L'Aquila, Italy, 7–9 October 2020; Proceedings; Demazeau, Y., Holvoet, T., Corchado, J.M., Costantini, S., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2020; Volume 12092, pp. 42–54. [CrossRef]
28. Jerald, J.; Giokaris, P.; Woodall, D.; Hartbolt, A.; Chandak, A.; Kuntz, S. Developing virtual reality applications with Unity. In Proceedings of the 2014 IEEE Virtual Reality (VR 2014), Minneapolis, MN, USA, 29 March–2 April 2014; IEEE Computer Society: Piscataway, NJ, USA, 2014; pp. 1–3. [CrossRef]
29. Bakar, F.A.; Cheung, C.; Yunusa-Kaltungo, A.; Mohandes, S.R.; Lou, E. "The State of Immersive Technology Application for Construction Safety Training": A Systematic Literature Review. In Proceedings of the the International Post-Graduate Research Conference (IPGRC), Salford, UK, 8–10 April 2022; pp. 1–16.
30. Becker-Asano, C.; Ruzzoli, F.; Hölscher, C.; Nebel, B. A Multi-agent System based on Unity 4 for Virtual Perception and Wayfinding. *Transp. Res. Procedia* **2014**, *2*, 452–455. In Proceedings of the Conference on Pedestrian and Evacuation Dynamics 2014 (PED 2014), Delft, The Netherlands, 22–24 October 2014. [CrossRef]
31. Motamedi, A.; Wang, Z.; Yabuki, N.; Fukuda, T.; Michikawa, T. Signage visibility analysis and optimization system using BIM-enabled virtual reality (VR) environments. *Adv. Eng. Inform.* **2017**, *32*, 248–262. [CrossRef]
32. Xie, J.; Yang, Z.; Wang, X.; Zeng, Q.; Li, J.; Li, B. A Virtual Reality Collaborative Planning Simulator and Its Method for Three Machines in a Fully Mechanized Coal Mining Face. *Arab. J. Sci. Eng.* **2018**, *43*. [CrossRef]
33. Yıldız, B.; Çağdaş, G. Fuzzy logic in agent-based modeling of user movement in urban space: Definition and application to a case study of a square. *Build. Environ.* **2020**, *169*, 106597. [CrossRef]
34. Karami, S.; Taleai, M. An innovative three-dimensional approach for visibility assessment of highway signs based on the simulation of traffic flow. *J. Spat. Sci.* **2022**, *67*, 203–218. [CrossRef]
35. Juliani, A.; Berges, V.; Vckay, E.; Gao, Y.; Henry, H.; Mattar, M.; Lange, D. Unity: A General Platform for Intelligent Agents. *arXiv* **2018**, arXiv:1809.02627.
36. Bogdanovych, A.; Esteva, M.; Simoff, S.J.; Sierra, C.; Berger, H. A Methodology for Developing Multiagent Systems as 3D Electronic Institutions. In Proceedings of the Agent-Oriented Software Engineering VIII, 8th International Workshop (AOSE 2007), Honolulu, HI, USA, 14 May 2007; Revised Selected Papers; Luck, M., Padgham, L., Eds; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2007; Volume 4951, pp. 103–117. [CrossRef]
37. Bogdanovych, A.; Esteva, M.; Simoff, S.J.; Sierra, C.; Berger, H. A methodology for 3D electronic institutions. In Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2007), Honolulu, HI, USA, 14–18 May 2007; Durfee, E.H., Yokoo, M., Huhns, M.N., Shehory, O., Eds.; IFAAMAS: Istanbul, Turkey, 2007; p. 57. [CrossRef]
38. Bogdanovych, A.; Simoff, S.J.; Esteva, M. Virtual Institutions: Normative Environments Facilitating Imitation Learning in Virtual Agents. In Proceedings of the Intelligent Virtual Agents, 8th International Conference, IVA 2008, Tokyo, Japan, 1–3 September 2008; Proceedings; Prendinger, H., Lester, J.C., Ishizuka, M., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2008; Volume 5208, pp. 456–464. [CrossRef]
39. Bogdanovych, A.; Simoff, S.J.; Esteva, M. Normative Virtual Environments—Integrating Physical and Virtual Under the One Umbrella. In Proceedings of the ICSOFT 2008—Proceedings of the Third International Conference on Software and Data Technologies, Volume PL/DPS/KE, Porto, Portugal, 5–8 July 2008; Cordeiro, J., Shishkov, B., Ranchordas, A., Helfert, M., Eds; INSTICC Press: Lisbon, Portugal, 2008; pp. 233–236.
40. Bogdanovych, A.; Rodríguez-Aguilar, J.A.; Simoff, S.J.; Cohen, A.; Sierra, C. Developing Virtual Heritage Applications as Normative Multiagent Systems. In Proceedings of the Agent-Oriented Software Engineering X—10th International Workshop, AOSE 2009, Budapest, Hungary, 11–12 May 2009; Revised Selected Papers; Gleizes, M., Gómez-Sanz, J.J., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2009; Volume 6038, pp. 140–154. [CrossRef]
41. Bogdanovych, A.; Rodríguez-Aguilar, J.A.; Simoff, S.J.; Cohen, A. Authentic Interactive Reenactment of Cultural Heritage with 3D Virtual Worlds and Artificial Intelligence. *Appl. Artif. Intell.* **2010**, *24*, 617–647. [CrossRef]
42. Tarau, P.; Bosschere, K.D.; Dahl, V.; Rochefort, S. LogiMOO: An Extensible Multi-user Virtual World with Natural Language Control. *J. Log. Program.* **1999**, *38*, 331–353. [CrossRef]
43. Strugala, D.; Walczak, K. Virtual Reality and Logic Programming as Assistance in Architectural Design. In Proceedings of the Augmented Reality, Virtual Reality, and Computer Graphics—6th International Conference, AVR 2019, Santa Maria al Bagno, Italy, 24–27 June 2019; Paolis, L.T.D., Bourdot, P., Eds.; Proceedings, Part I; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2019; Volume 11613, pp. 158–174. [CrossRef]
44. Angilica, D.; Ianni, G.; Pacenza, F. Declarative AI design in Unity using Answer Set Programming. In Proceedings of the IEEE Conference on Games (CoG 2022), Beijing, China, 21–24 August 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 417–424. [CrossRef]
45. Lifschitz, V. Action Languages, Answer Sets, and Planning. In *The Logic Programming Paradigm—A 25-Year Perspective*; Apt, K.R., Marek, V.W., Truszczynski, M., Warren, D.S., Eds.; Artificial Intelligence; Springer: Berlin/Heidelberg, Germany, 1999; pp. 357–373. [CrossRef]
46. Lifschitz, V. *Answer Set Programming*; Springer: Berlin/Heidelberg, Germany, 2019. [CrossRef]

47. Costantini, S.; Gasperis, G.D.; Migliarini, P. Constraint-Procedural Logic Generated Environments for Deep Q-learning Agent training and benchmarking. In Proceedings of the 37th Italian Conference on Computational Logic, Bologna, Italy, 29 June–1 July 2022; Calegari, R., Ciatto, G., Omicini, A., Eds.; 2022; Volume 3204, CEUR Workshop Proceedings, pp. 268–278. Availalbe online CEUR-WS.org (accessed on 19 January 2023).
48. Csuhaj-Varjú, E. Grammar Systems: A Multi-Agent Framework for Natural Language Generation. In *Mathematical Aspects of Natural and Formal Languages*; Paun, G., Ed.; World Scientific Series in Computer Science; World Scientific: Singapore, 1994; Volume 43, pp. 63–78. [CrossRef]
49. Aref, M.M. A multi-agent system for natural language understanding. In Proceedings of the IEMC '03—Managing Technologically Driven Organizations: The Human Side of Innovation and Change (IEEE Cat. No.03CH37502), Cambridge, MA, USA, 30 September–4 October 2003; pp. 36–40. [CrossRef]
50. Yoon, V.Y.; Rubenstein-Montano, B.; Wilson, T.; Lowry, S. Natural Language Interface for a Multi Agent System. In Proceedings of the 10th Americas Conference on Information Systems (AMCIS 2004), New York, NY, USA, 6–8 August 2004; Association for Information Systems: Atlanta, GA, USA, 2004; p. 215.
51. Casillas, L.A.; Daradoumis, T. Constructing a Multi-agent System for Discovering the Meaning over Natural-Language Collaborative Conversations. In *Intelligent Collaborative e-Learning Systems and Applications*; Daradoumis, T., Caballé, S., Marquès, J.M., Xhafa, F., Eds.; Springer: Berlin/Heidelberg, Germany, 2009; Volume 246, Studies in Computational Intelligence; pp. 99–112. [CrossRef]
52. Trott, S.; Appriou, A.; Feldman, J.; Janin, A. Natural Language Understanding and Communication for Multi-Agent Systems. In Proceedings of the 2015 AAAI Fall Symposia, Arlington, VA, USA, 12–14 November 2015; AAAI Press: Palo Alto, CA, USA, 2015; pp. 137–141.
53. Doubleday, S.; Trott, S.; Feldman, J. Processing Natural Language About Ongoing Actions. *arXiv* **2016**, arXiv:1607.06875.
54. Busetta, P.; Calanca, P.; Robol, M. *Applying BDI to Serious Games: The PRESTO Experience*; Technical Report; University of Trento: Trento, Italy, 2016.
55. Poli, N. Game Engines and MAS: BDI & Artifacts in Unity. Master's Thesis, Alma Mater Studiorum Universita di Bologna, Bologna, Italy, 2018.
56. Denti, E.; Omicini, A.; Ricci, A. Multi-paradigm Java-Prolog integration in tuProlog. *Sci. Comput. Program.* **2005**, *57*, 217–250. [CrossRef]
57. Mariani, S.; Omicini, A. Game Engines to Model MAS: A Research Roadmap. In Proceedings of the 17th Workshop "From Objects to Agents" Co-Located with 18th European Agent Systems Summer School (EASSS 2016), Catania, Italy, 29–30 July 2016; Santoro, C., Messina, F., Benedetti, M.D., Eds.; 2016; Volume 1664, CEUR Workshop Proceedings; pp. 106–111. Available online: CEUR-WS.org (accessed on 19 January 2023).
58. Marín-Lora, C.; Chover, M.; Sotoca, J.M.; García, L.A. A game engine to make games as multi-agent systems. *Adv. Eng. Softw.* **2020**, *140*, 102732. [CrossRef]
59. Sudkhot, P.; Sombattheera, C. A Crowd Simulation in Large Space Urban. In Proceedings of the 2018 International Conference on Information Technology (InCIT), Khon Kaen, Thailand, 24–26 October 2018; pp. 1–8. [CrossRef]
60. Paschal, C.H.; Shiang, C.W.; Wai, S.K.; bin Khairuddin, M.A. Developing Fire Evacuation Simulation Through Emotion-based BDI Methodology. *JOIV Int. J. Inform. Vis.* **2022**, *6*, 45–52. [CrossRef]
61. Benkhedda, S.; Bendella, F. FASim: A 3D Serious Game for the First Aid Emergency. *Simul. Gaming* **2019**, *50*, 690–710. [CrossRef]
62. Matoso, O.A.; Lampert, L.; Hübner, J.F.; Conceição, M.; Bernardes, S.P.; Amaral, C.J.; Zatelli, M.R.; de Lima, M.L. Agent Programming for Industrial Applications: Some Advantages and Drawbacks. *arXiv* **2020**, arXiv:2006.05613.
63. Wai, S.K.; WaiShiang, C.; bin Khairuddin, M.A.; Bujang, Y.R.B.; Hidayat, R.; Paschal, C.H. Autonomous Agents in 3D Crowd Simulation Through BDI Architecture. *JOIV Int. J. Inform. Vis.* **2021**, *5*, 1–7. [CrossRef]
64. Brännström, A.; Nieves, J.C. A Framework for Developing Interactive Intelligent Systems in Unity. In Proceedings of the Engineering Multi-Agent Systems (EMAS 2022), Virtual Event, 9–10 May 2022; Amit Chopra, J.D., Zalila-Wenkstern, R., Eds.; Springer: Berlin/Heidelberg, Germany, 2022.
65. Bechhofer, S.; van Harmelen, F.; Hendler, J.; Horrocks, I.; McGuinness, D.; Patel-Schneijder, P.; Stein, L.A. OWL Web Ontology Language Reference. Recommendation, World Wide Web Consortium (W3C). 2004. Available online: http://www.w3.org/TR/owl-ref/ (accessed on 19 January 2023).
66. Moreira, Á.F.; Vieira, R.; Bordini, R.H.; Hübner, J.F. Agent-Oriented Programming with Underlying Ontological Reasoning. In Proceedings of the Declarative Agent Languages and Technologies III, Third International Workshop, DALT 2005, Utrecht, The Netherlands, 25 July 2005; Baldoni, M., Endriss, U., Omicini, A., Torroni, P., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3904, pp. 155–170. Selected and Revised Papers. ._10. [CrossRef]
67. Mascardi, V.; Ancona, D.; Barbieri, M.; Bordini, R.H.; Ricci, A. CooL-AgentSpeak: Endowing AgentSpeak-DL agents with plan exchange and ontology services. *Web Intell. Agent Syst.* **2014**, *12*, 83–107. [CrossRef]
68. Yang, Z.; Qian, Y.; Zou, J.; Lee, C.L.; Lin, C.L.; Wu, S.T. Reducing the Power Consumption of VR Displays with a Field Sequential Color LCD. *Appl. Sci.* **2023**, *13*, 2635. [CrossRef]
69. Spierling, U.; Grasbon, D.; Braun, N.; Iurgel, I. Setting the scene: Playing digital director in interactive storytelling and creation. *Comput. Graph.* **2002**, *26*, 31–44. [CrossRef]

70. Brookes, J.; Warburton, M.; Alghadier, M.; Mon-Williams, M.; Mushtaq, F. Studying human behavior with virtual reality: The Unity Experiment Framework. *Behav. Res. Methods* **2020**, *52*, 455–463. [CrossRef] [PubMed]
71. Damiano, R.; Lombardo, V. Value-Driven Characters for Storytelling and Drama. In Proceedings of the AI*IA 2009: Emergent Perspectives in Artificial Intelligence, Reggio Emilia, Italy, 9–12 December 2009; Serra, R., Cucchiara, R., Eds.; Springer: Berlin/Heidelberg, Germany, 2009; pp. 436–445. [CrossRef]
72. Peinado, F.; Cavazza, M.; Pizzi, D. Revisiting Character-Based Affective Storytelling under a Narrative BDI Framework. In Proceedings of the Interactive Storytelling, Erfurt, Germany, 26–29 November 2008; Spierling, U., Szilas, N., Eds.; Springer: Berlin/Heidelberg, Germany, 2008; pp. 83–88. [CrossRef]
73. Rank, S.; Hoffmann, S.; Struck, H.G.; Spierling, U.; Petta, P. Creativity in configuring affective agents for interactive storytelling. In Proceedings of the International Conference on Computational Creativity, Montpellier, France, 27 August 2012; p. 165.
74. Berov, L. Character focused narrative models for computational storytelling. In Proceedings of the Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference, Little Cottonwood Canyon, UT, USA, 5–9 October 2017.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.