FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

INTRA SPACE Agent

An Agent-Based Architecture for an Artistic Real-Time Installation

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Tom Tucek
Matrikelnummer 01325775

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Projektass. (FWF) Dipl.-Ing. Christian Freude, BSc

Wien, 29. November 2018

Tom Tucek	Michael Wimmer

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

INTRA SPACE Agent

An Agent-Based Architecture for an Artistic Real-Time Installation

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Tom Tucek
Registration Number 01325775

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Projektass. (FWF) Dipl.-Ing. Christian Freude, BSc

Vienna, 29th November, 2018

Tom Tucek	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Tom Tucek
Kanalstrasse 8/2; 1220 Vienna; Austria

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. November 2018

Tom Tucek

Danksagung

Ich möchte hiermit allen Projektmitgliedern von INTRA SPACE danken, welche mir erlaubten, mich an ihrem Streben nach Wissen zu beteiligen. Mein besonderer Dank gilt dem Team innerhalb des Raumes in welchem alles stattfand – Christina Jauernik, Simon Oberhammer, Martin Perktold und Christian Freude. Letzterer, welcher mich nicht nur während der Entwicklung unterstützte, sondern auch mein Schreiben an dieser Arbeit mit Geduld, konstruktiver Kritik und gütigem Beistand förderte, stellt einen der Hauptgründe dar, dass dieses Werk fertiggestellt wurde.
Ich möchte weiters sowohl Professor Wolfgang Tschapeller und Esther Balfe danken, wessen Wirken an diesem Projekt mich zu neuen Sichtweisen im Rahmen der Wissenschaft führte. Weiterer Dank gilt Professor Michael Wimmer, welcher mir die Gelegenheit bat, an diesem Projekt zu arbeiten und diese Arbeit an seinem Institut zu schreiben.
Danke an Professor Michael Thielscher, welcher unserem Team dabei half, mit Agenten Entwicklung anzufangen und uns den Weg geleitete. Ich bin besonders dankbar für die Unterstützung durch Professor Paolo Petta, welcher uns alle ermutigte und inspirierte, neue Wege zu finden, selbst wenn unser Team nicht weiter wusste.
Schließlich möchte ich all meinen Freunden, meiner Familie und meinem Hund danken – für all ihren konstanten Beistand und ihre Ermutigung im Laufe meiner Arbeit. Spezieller Dank gilt meinem Vater, für seine stetige Bereitschaft mich in allem zu unterstützen was ich unternehme.

Acknowledgements

I would like to express my gratitude to the entire team of project INTRA SPACE, who allowed me to join them in their pursuit of knowledge. Special thanks go out to the team stationed inside of the development room, consisting of Christina Jauernik, Simon Oberhammer, Martin Perktold, and Christian Freude. The latter, who did not only aid me during project development, but also supported me in writing this work with both wisdom and patience, giving constructive criticism and benevolent assistance, was a major cause for this thesis to be finished.
Further thanks to both Professor Wolfgang Tschapeller and Esther Balfe, whose work on this project opened my eyes to new ways of thinking in an academic setting. I would also like to thank Professor Michael Wimmer, who allowed me to work on this project and write this thesis at his institute.
Thanks to Professor Michael Thielscher, for helping the project team getting started with agent development and guiding us on our way. I am particularly grateful for the assistance given by Professor Paolo Petta, for encouraging and inspiring our team in times we had trouble finding ways to advance.
Finally, I wish to thank all of my friends, my family and my dog, for their support and encouragement throughout my work. Special thanks to my father, for his constant willingness to assist me in everything I do.

Kurzfassung

Diese Arbeit behandelt und beschreibt die Entwicklung und Einbettung eines Agenten-Systems in eine künstlerische Einrichtung. Das System ist dafür verantwortlich, virtuelle Figuren auf einer Leinwand zu kontrollieren, um somit Besuchern der Einrichtung die Interaktion mit jenen Figuren zu ermöglichen. Das Verhalten der Agenten basiert auf zuvor vom Projekt-Team gestalteten Stories und soll darauf optimiert sein, sowohl von Besuchern gut wahrgenommen zu werden, als auch dem Projekt Team zu erlauben, an der Interaktion zwischen Menschen und Nicht-Menschen zu forschen.
Die Implementierung des Agenten Systems wurde mittels Jason wahrgenommen, eines Java-basierten Interpreters der Agenten-Programmiersprache AgentSpeak. Es wurden unterschiedliche Agenten-Szenarien über den Verlauf des Projektes entwickelt und mittels verschiedener Herangehensweisen implementiert. Ein iterativer und dynamischer Entwicklungsprozess wurde verfolgt, während regelmäßige Meetings der Projektmitglieder stattfanden, bei welchen Fortschritt und Ideen mit Unterstützung von Visualisierungen diskutiert wurden. Das Verhalten der Agenten war von vielen Problemen geprägt, da die Agenten entweder zu sehr von Reaktionen auf Besucherverhalten abhängig waren, oder sich stattdessen zu stark auf eigene Ziele konzentrierten, worunter die Qualität der Interaktion mit Besuchern der Einrichtung litt. Um diese Probleme zu lösen, wurden unterschiedlichste Ansätze verfolgt, diskutiert und dokumentiert.
In der finalen Version der Einrichtung wurden Agenten mit indeterministischem und emergentem Verhalten eingesetzt. Weiters haben Agenten den Fokus auf Besucherverhalten mit dem Verfolgen der eigenen Ziele balanciert, was zur Folge hatte, dass Besucher die Agenten als gesellschaftliche Präsenz wahrnehmen konnten, während die Interaktion auf sowohl natürliche, als auch neuartige Weise stattfinden konnte.

Abstract

This work describes the processes involved in developing and embedding an agent system into an artistic real-time installation. The agent system would be responsible for controlling virtual figures on a screen, which interact with users of the installation. It was necessary to develop agents which displayed behavior pre-defined in stories designed by the project team, as well as to ensure that such agents acted in a way that was both well received by visitors, while also stimulating interaction in a way that allowed the project team to conduct research on the interactions between humans and nonhumans. The agent system was implemented using Jason, a Java-based interpreter of the agent-programming language AgentSpeak. Over the course of the project, various agent scenarios were developed, with differing ways of implementation. An iterative process was used for development and regular meetings with project members were instated, to discuss progress and ideas, while utilizing visualizations to aid communication. Behavior of developed agents was plagued by various problems, from being too reliant on reactions towards user behavior, to not interacting enough with active users. Various approaches to such problems were tried out, discussed, and documented.
During the final installation, agents with indeterministic and emergent behavior were employed. Furthermore, agents were focused on both pursuing their own goals as well as constantly paying attention to visitor behavior. This allowed users to realize agents as a social presence and interact with them in a way that was both novel and natural.

Contents

CHAPTER 1

Introduction

This work describes the process and challenges of developing an agent-based system for an artistic real-time installation called INTRA SPACE. The installation, which was being developed as an artistic research project by the academy of fine arts in Vienna in cooperation with the TU Wien, had the goal of exploring and deconstructing interactions between humans and so-called nonhumans. The main idea of the installation was to offer Visitors the chance to interact with a virtual figure on a wide screen in front of them. The virtual figures would guide Visitors through certain procedures, mimic them, interact with them, as well as follow their own goals.
The virtual figures were to be controlled by an independently running agent-system, which would have access to information regarding the Visitors' behavior, and should thus determine the actions of one or more characters on the screen. Furthermore, the agents' course of action had been predefined by various stories, written by the research group. Converting these stories into AgentSpeak (AS) using Jason was the first main goal of the task. Caused by the constantly developing environment of the project, development of the agent system faced various dynamic goals as well. The research question of this work is thus "How can an agent system be developed and embedded into an installation with dynamic requirements?". This document describes the project's vision and set-up, followed by a detailed description of the process of developing, implementing, and embedding an agent system into the artistic real-time installation of the project.
This work is generally structured chronologically, as implementations are presented in the order they were developed. However, alternatives and other additional information are sometimes based more on hindsight and reflection. After the installation is described and explained, an overview of related works is presented as well. The aforementioned research question and resulting sub-questions are presented with multiple possible approaches, as well as detailed descriptions of the paths which were actually chosen and implemented. Concluding remarks include reflections on the project, problems, possible solutions and approaches, discussion, and further work.

1.1 About Project INTRA SPACE

Project INTRA SPACE was an artistic research project by the academy of fine arts in Vienna, which ran from April 2015 until June 2017, with involvement of the author of this work starting in August 2016. It was funded by the Fonds zur Förderung der wissenschaftlichen Forschung (Austrian Science Fund) (FWF).

> "*INTRA SPACE: The reformulation of architectural space as a dialogical aesthetic* explores how interactions between, across and beyond humans and nonhumans can be experimentally embodied, aesthetically reformulated and theoretically challenged in their spatial, temporal and transversally entangled spheres." [JO]

Visitors were able to experience the finished product either by themselves, or watch performers interact with it. Although the installation was considered mobile and has been rebuilt at another location for presentation purposes, its development, testing, and presentation during the author's involvement were conducted within a single room. The layout of the room was changed multiple times during development, with the final layout being presented as a schematic overview in Figure 1.1, as well as a picture, taken from the screen's side opposing the motion-capture space, in Figure 1.2.
The layout seen in Figure 1.1 can be separated into a motion-capture space, a back-projection space, as well as two work spaces. Computers running necessary software or used for development were placed in the work spaces. The two most notable computers were called the "Render Node" and the "Captury Node". First of which was connected to the projector and responsible for agents appearing on screen, as well as hosting the agent system. The second PC called "Captury Node" was connected to the cameras tracking the Visitor within the motion-capture space. This machine was responsible for processing live picture data fed to it via Ethernet, discerning relevant information, such as the tracked person's position and movements, and finally sending the pre-processed data to the "Render Node".
The installation before the introduction of Artificial Intelligence (AI) was comparable to [BSM+14a], with the agent on the screen mirroring the Visitor's movements in real time. After a short "donning" phase (explained further in Section 3.3), the Visitor's movements were entirely recreated by the agentfigure on the screen (see Figure 1.3). As seen in Figure 1.2 and Figure 1.3, agentfigures were not rendered in a photorealistic way during the final phases of the project, but instead were chosen to be represented by monochromatic, sexually ambiguous models. There were several models in use, as can be seen in Figure 1.4, and a common characteristic was the high polygon count for the figures faces and hands, while their bodies were made of a noticeably lesser amount of polygons.
The goal of adding an agent system to the project was for the agentfigure to "emancipate" and disconnect from the Visitor, realizing its own motions and behavior, thus allowing, even provoking, more interaction between the Visitor and the figure. The initial vision

consisted of the agentfigure copying the Visitor's movements for a random, but limited amount of time, before slowly starting to act on its own. The Visitor was supposed to gradually realize the emancipation of the figure, and move on from one-way interaction by being replicated, to two-way interaction by gesture-based dialogue. The agent system containing the required AI initially had to meet the following requirements:

- Communicate with the already installed system by sending and receiving information via a defined interface
- Run isolated and independently from the rendering engine, be fault tolerant and stable
- Take control over the agentfigures' actions at appropriate times
- Incorporate pre-written stories into interactions with Visitors
- Make decisions based on the information it is presented in real time (information may be incomplete or faulty)

During development additional requirements emerged, such as non-deterministic behavior and Visitor guidance, which are further explored in later chapters.
Development of the agent system took place in a period of about six months, using iterations and continuous feedback from project staff, experts, and Visitors. Implementation was based on independent stories, which were to be told by the agentfigures' actions. Stories would sometimes feature branching paths depending on Visitor behavior, or would not follow a time line at all, but rather be purely reactive in nature. Early agent stories were mostly based either on animalistic behavior, such as that of a bat or an octopus, or on goals to be reached – for example getting the Visitor to perform a certain action, such as waving towards the agent. Some of the stories are explained in detail in Chapter 4.
Pre-recorded motions from the motion-capture software The Captury Live [cap] were used as basic building blocks for agent behaviour. These motions could either be of continuous nature, such as walking or standing animation cycles, or isolated actions, such as hand gestures. Certain actions would leave the agent in a different state then before. For example, sitting down on the floor or getting up, would switch between the states of standing and sitting, thus enabling the agent to perform different actions from before. Because of graphic depictions of such features, stories for the agent system eventually transformed into state machines – a topic which is further discussed in Subsection 4.1.4.

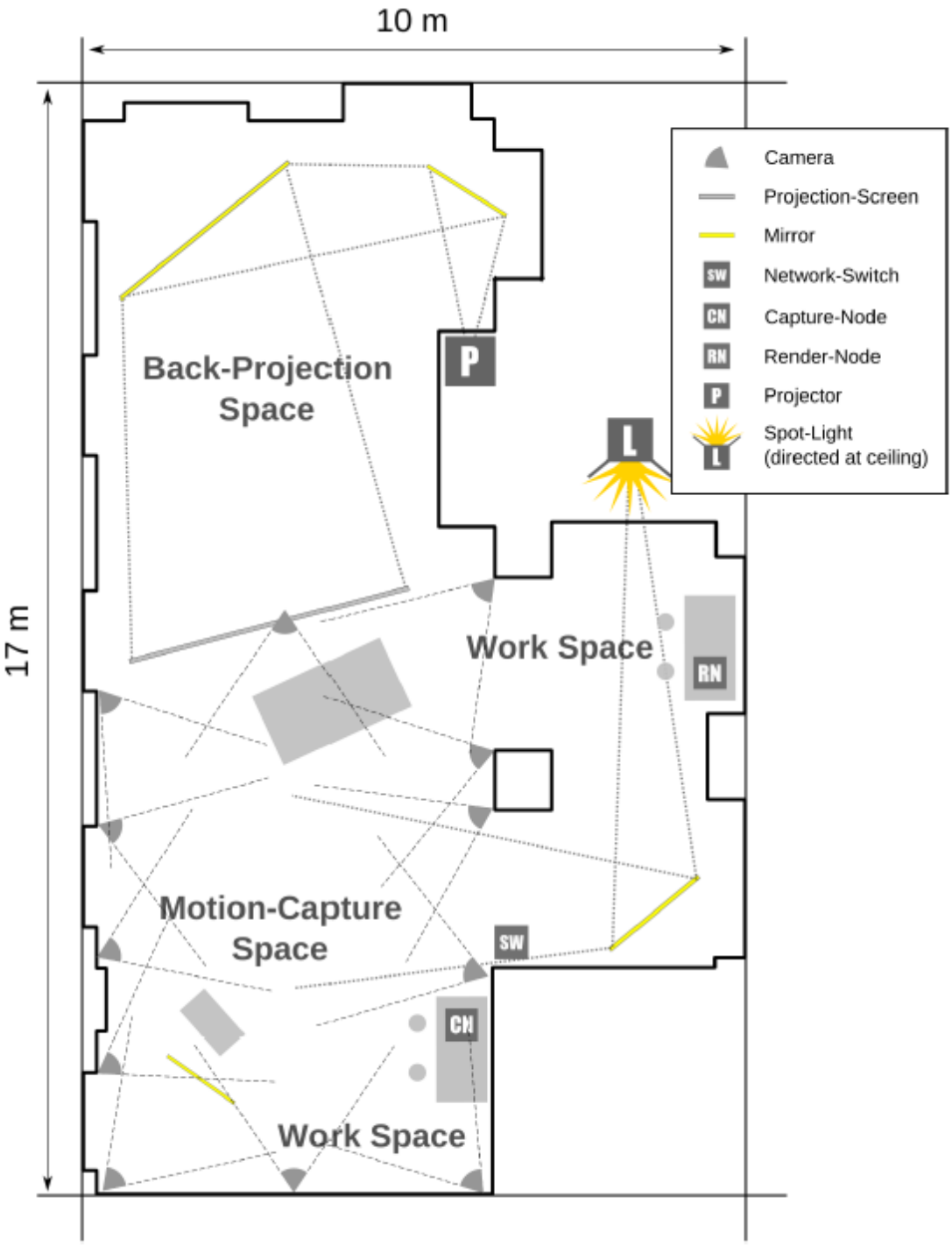

Figure 1.1: Schematic plan of final setup. (Taken from the technical report about INTRA SPACE, by Christian Freude.)

Figure 1.2: Picture of the installation. (Picture taken by Günter Richard Wett.)

Figure 1.3: Picture of the agentfigure mirroring the Visitor's movements. (Picture taken by Günter Richard Wett.)

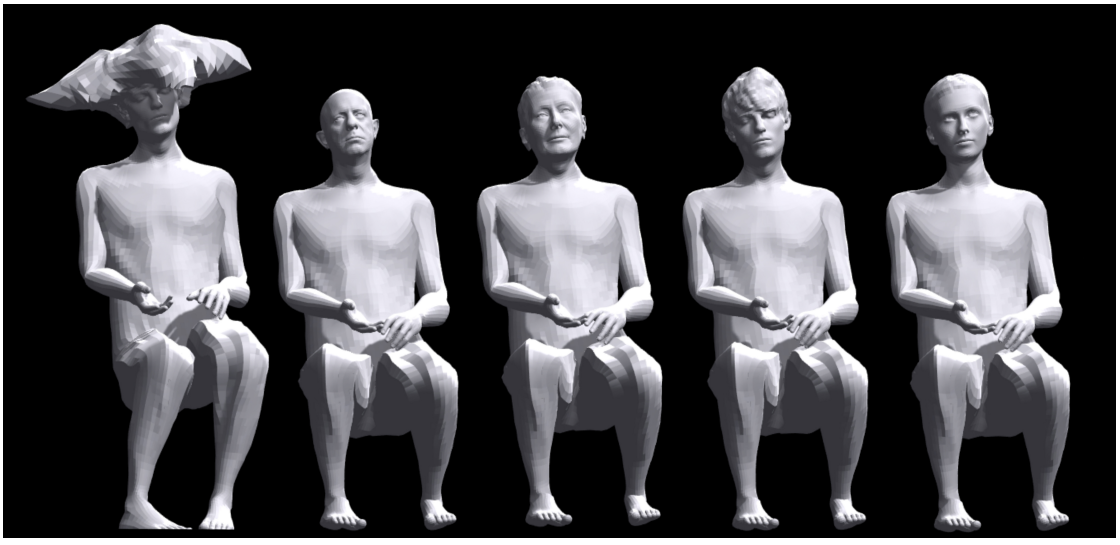

Figure 1.4: Various models used in the project. From left to right: Bob with hat, Old man, Vivienne, Bob, Carla.

CHAPTER 2

Related Work

Interactive agents on screens can be found in current research as an approach to entertainment, education, and training. While entertainment and art overlap in many areas, not many such projects have explored the more philosophical side of such an interaction as in INTRA SPACE. But as the main focus of this work is describing the implementation of the implemented agent system, this chapter reviews similar use-cases and installations, as well as general research regarding the topic of interactions between users and agent systems.

2.1 User Interaction and Depiction of Agents

Fundamental research on user interactions with virtual agents in both virtual and mixed environments has been conducted in [CMC+03], where the difference between such environments is highlighted:

> "User interaction with virtual agents generally takes place in virtual environments in which there is clear separation between the virtual actors and the user, due to the fact that in most cases, the user is in some way external to the virtual world. In Mixed-Reality Interactive Storytelling, the user's video image is captured in real time and inserted into a virtual world populated by autonomous synthetic actors with which the user interacts. The user in turn watches the composite world projected on a large screen, following a 'magic mirror' metaphor." [CMC+03, p. 1]

The same magic mirror metaphor can be applied to the project described in this work, as Visitors' image data is captured and information gained from it is used to render images onto the screen. However, different from such mixed-reality applications, the mirror does not reflect the world of the user and instead serves as an interface to the world of the

virtual agent.
Relevant research can also be found in the field of human-robot interaction, as robots serve as agents, controlled by AI as well. While robots have a physical body and agents of this project do not, many problems apply to both alike, such as appearing human or human-like to users. In [SCB+12], it is described when and how humans view robots as robots or not, whereas [SCFH98] describes human-robot interaction with agents using artificial emotions including learning and self-adaption features, similar to the agents in this project. Emotions are exhibited by visuals, lights, music, movement, behavior, etc. – methods that have been tried to represent INTRA SPACE's agents' emotions as well. In [SCFH98], humans interact with the robot without wearing any sensors, as the robot uses ultrasound sensor and cameras to recognize user actions. Furthermore, the use of the word kansei (sense, sensibility) is suggested, as it describes a field of major research in Japanese robotics – how do robots express emotions?
Similar research can be found in [PPP14], which provides answers to the questions of what makes humans perceive agents as a social presence, and how can it be improved. They conducted a case study, in which a multiplayer board game was played together with a robot. The results of this study led them to establish 5 guidelines, which were a physical body, believable verbal and nonverbal behavior, an emotion system, social memory, and the simulation of social roles. Accordingly, agents controlling robots must adhere to those rules, to be realized as a social presence by humans. The importance of a physical body is especially interesting in the context of this work, as the agents of this project do not possess one to the extent that is considered necessary by [PPP14], but are merely projections on a physical screen.
Depiction of lifelike virtual humans is a topic of research for various applications, as is described in [MBB12]. Such depictions find use in movies, games, interactive dramas, therapy, training, and marketing (see also [GRA+02]). Realistic rendering oftentimes causes a negative reaction among viewers, caused by the famous uncanny valley phenomena. [MBB12] also goes into detail on motion capture and eye tracking, and presents an evaluation on the eleven different rendering styles which have been tried out over the course of their research.
Other topics relevant to the project and AI development, even if not directly the focus of this work, include research on motion capture of humans, such as [MHK06], which dictates four major steps which were used in the project of this work as well – initialization, tracking, pose estimation, and recognition. Tracking human gestures is a field of study that ties into various applications within modern computer science, as [WKSE11] argues that future human-computer interaction will enable more natural, intuitive communication resembling human-human communication by employing gesture tracking.

2.2 Similar Installations

FlurMax – an installation very similar to the one described in this work, containing an interactive virtual agent in a hallway to entertain Visitors, is described in [JK03]. Another example for conversational agents can be found in web-based Eliza from [LB17]. Such

agents focus on conversation in natural human language, whereas the agents described in this work are, with a few exceptions, only able to understand and communicate via gestures.
More focused on the artistic side and thus more similar installations can be found in [BSM+14b], in which the effects of human-virtual agent body interaction are explored by conducting an experiment on user experience. The results show that the most obvious dimension to users is the so called dimension of "coupling", which measures the level of connection between the user and the virtual agent, and is considered necessary for users to feel engaged.

> "Coupling is the continuous mutual influence between two individuals, and has a dynamic specific to the dyad. It possesses the capability to resist disturbance, and compensates by evolving the interaction. Disturbances come from both the environment and from within the individuals, depending on how they perceive the interaction." [BSM+14b, p. 1]

In the experiment, the virtual on-screen agent mirrors the user's movements and actions, while the wizard of Oz (a person hidden from user) is able to manipulate agent actions directly, to cause disturbances within the interaction. The study claims that a 0.5 second delay on mirroring – meaning that 0.5 seconds would pass before the virtual agent enacts the same movement as the user did before – was found out to be the best value to cause the agent to seem less automated and more natural or intelligent.
A follow-up experiment is described in [BJNDL15], which focuses once again on coupling between humans and virtual characters in an artistic context of imitation. It features a very similar set-up, in which the user and the agent figure mirror one another. The whole experience is defined as a "game", in which the user can "fail", by losing the virtual figures interest, caused by a lack of interaction. This second installation is even more similar to the one described in this work, as it features agent behavior, goals, and AI in order to shape the interaction between the agent and the user. It is however notable, that the interaction with the virtual agent in INTRA SPACE was explicitly stated not to be seen as a game and thus differentiates itself from the research found in [BJNDL15].
In [PL11], research was conducted on motion-based bodily interactions with virtual characters and methodology based on the findings is proposed. The experiments used a similar set-up as well, and furthermore also used pre-recorded animations in combination with real-time captured animations to bring the agents to life.

2.3 Jason and Alternatives

Jason [BH+04] [BHW07] is used as a platform to develop agent systems in various different contexts. Because of its portability by being implemented completely in Java, as well as being based on the agent programming language AS [Rao96] which is in turn based on the Belief-Desire-Intention (BDI) model, agents in Jason can fulfill the role of an agent system for a myriad of requirements. The platform works as an interpreter for

an extended version of AS, a BDI agent-oriented logic programming language. Because it allows Multi-agent systems (MASs) to be distributed over a network, its primary usage lies within distributed multi agent systems. It offers speech-act based inter-agent communication, as well as annotations on plans. Users can customize features such as trust functions, selection functions, and internal actions. The environment, in which the agents act, can also be implemented and customized in Java. It is considered easy to learn and has good scalability, user friendly GUI features, good performance, and high stability. Finally, it is free and open source. More details on Jason, as well as an explanation why it was chosen for the project, can be found in Chapter Section 3.4.
An example for relevant agent systems implementations using Jason can be found in [RCP11], in which Jason is used for an agent platform to act within the online game Second Life and has to rely on unreliable sensor data. The game Second Life is generally often used as a platform for virtual agents and agent systems to act within (see also [BRASC10]). Another example is the system described in [BBH+13], in which Jason is used in conjunction with Moise for agent organization and Common ARTifact infrastructure for AGents Open environments (CArtAgO) for shared environments. CArtAgO [RVO06] is a framework designed to extend MASs with artifact based working environments. CArtAgO in combination with Jason is furthermore also used in [RPV11], which implement artifacts as abstract entities in an environment, for agents to interact with.
As there is already a large amount of different agent platforms available to develop agent systems with, choosing the right one for a project can be a difficult task. Furthermore, the wrong choice of an agent platform can lead to unnecessary work or unsatisfying results. In [KB15], a vast amount of agent platform options is presented, compared, and reviewed, to help making informed decisions on which platform to use. Similarly, [OFS99] presents an overview of what kind of research can be used for which applications in regards to MASs.
An alternative to using Jason can be found in JACK [Win05], which is considered to be the leading commercial BDI-agent toolkit, extending Java intuitively and thus being easily portable, same as Jason. Another Java-based alternative would be JADE [BPR99], which is considered the most popular platform in the industry and academic community. JADE focuses on distributed applications. NetLogo [TW04] offers a multi agent modeling environment, designed after the Logo programming language, and is considered the most popular in education and research, as it is designed to be used without beforehand knowledge about programming.
Concurrent MetateM [BFG+95] offers a unique language based on temporal logic for modeling reactive systems. It was suggested as an alternative during the development process of INTRA SPACE, and although the language itself ended up not being used, various ideas and concepts of the language were adopted and used to develop the project in the end.
Other approaches have agents learning from scratch to solve problems or act on their desires. Examples include the use of back-propagation neural networks to create Intelligent Agents (IAs) in intelligent virtual environments, such as in [JZ07], in which a self-learning

AI uses such neural networks to navigate a maze-like game.
In [SM12], a biologically inspired decision making system is implemented, which uses emotions, such as happiness, sadness, and fear to help the AI to learn how to act from scratch. Similarly, in [GC98], artificial animals using neural networks are implemented using genetics, breeding, and a biochemical system. They interact with humans and each other, and results show that creatures appear to learn and display emergent social behavior. More research on artificial life can be found in [Mae95], which “models life as it could be, so as to understand life as we know it".

2.4 Story-telling

Software agents can be used in various ways, such as a mechanism to help computer users with work and information overload, but also purely for arts and entertainment, as interactive artificial agents are already a big part of the modern entertainment industry. Telling stories using agent systems is a research topic that ties closely to this work, as the virtual agents try to communicate with Visitor's based on predefined stories. [Bre10] presents a dynamic story based on a MAS, in which beliefs, sentiments, and goals change all the time. They employ plot graphs using events, temporal and causal links, beliefs, etc. to visualize stories and argue that generation of dynamic plots has not been studied sufficiently. [RS06] aims for believable agents for interactive storytelling, constructing a coherent narrative combined with user agency, by employing a drama manager (an automated story director) to orchestrate and guide agents within their system. A summarizing work regarding interactive narrative research can be found in [RB12], which describes its uses in entertainment, education, and training. They describe interactive narrative as a digital interactive experience with user influence, which creates a dramatic storyline. However, users must believe that their actions can significantly alter the direction or outcome of the story, meaning they must receive feedback to their actions and be able to comprehend the consequences, as opposed to being shown scripted stories which do not involve them. This problem was also encountered during this work's project (see Subsection 4.1.4).

CHAPTER 3

The System

As described in Section 1.1 and seen in Figure 1.1, the physical set-up can be divided into multiple spaces. In this chapter, the spaces' separate entities and their connections are further described and explained. In Figure 3.1, the flow of data and information within the system is shown. The main focus lies on the agent system which is running independently inside the Render Node, containing a simulated environment and the individual agents. Wrappers and interfaces for communication between entities are not shown in the depiction.

3.1 Overview

12 cameras are placed around the motion-capture space to record the Visitors' movements from various angles, enabling the motion-capture software The Captury Live [cap] to reconstruct a model corresponding to each Visitor. The model's skeletal data can then be saved as a recording, or sent directly to the render node, which is responsible for drawing the agentfigures. The render node utilizes various ways to analyze the received data – such as the Full Body Interaction Framework (FUBI) [Kis], which is able to recognize pre-defined gestures like waving or sitting. Data on recognized gestures is then processed by the rendering engine and shared with the running instance of the Java Virtual Machine, containing the Jason processes. Data communication between Java and the rendering engine was handled via User Datagram Protocol (UDP), by sending packets of information encoded in Binary JavaScript Object Notation (JSON) (BSON), which is a binary data interchange format based on JSON.
The Java thread receiving the packets processes the information and feeds it into the simulated environment, which is part of the Jason architecture. Agents within the environment are able to perceive it and changes within it, thus being able to react accordingly to new information. Jason agents are able to control the agentfigures within the rendering engine by sending them commands using Jason's custom-definable actions.

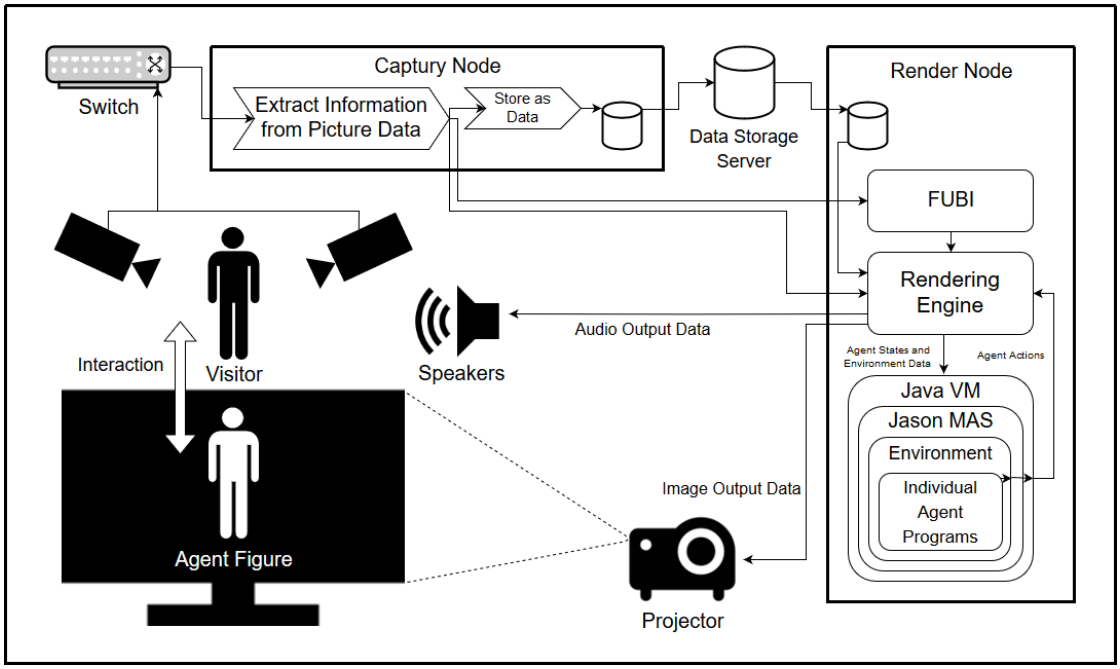

Figure 3.1: Simplified schematic overview of information flow within the installation.

Because of this implementation, it could be said that the actual agents within the Jason platform are not in absolute control of the on-screen agentfigures visible to Visitors. A fitting allegory might be puppet masters controlling their puppets using strings, without being able to see exactly what their puppets could see. As such, it is theoretically possible for the agentfigures to fail performing commands issued to them by the Jason agents – a problem which had to be considered during development.
The environment is generally considered to be an empty space, causing the simulated environment within the Jason platform to be rarely used. However, interactions with artifacts, such as tangible objects, sources of darkness and light, or cameras, are also possible in some agent stories. In such cases, the artifacts are often-times part of the simulated environment, although related calculations ultimately proved to be more efficiently done directly within the rendering engine.
The rendering engine is responsible for performing calculations, animation, and drawing the agentfigures onto the screen. Using orthographic projection and a custom-made projection-mapping component to nullify alignment errors, the picture was drawn onto the screen using a professional projector. Because of physical constraints, the projectors image was reflected by two large-scale mirrors before reaching the screen (see Figure 1.1). Finally, the large screen, suspended mid-air, functioned as a window for Visitors to see into the world of the virtual agentfigures. While Visitors got their information about the agents from the screen, the agents got their information about the Visitors from the cameras surrounding the Visitors, thus completing the circle of information flow and interaction.

3.2 The Room

Most of the project's later stages were developed and executed within a single room. The layout of said room can be seen in Figure 1.1. The room featured white, blank walls and a grey floor, its architecture including large pillars and arcs to support the ceiling and upper floors. The room's decoration was minimalistic and simple, with notable parts being the large mirrors, some of which were used to refract the light from the projector onto the screen. The room's small windows were shut during visits, as the ever-changing lighting conditions of sunlight could easily disrupt the tracking software and thus lead to unwanted movements or behavior of the agents. Instead, the room was illuminated by a large spot light, which was also refracted by a mirror suspended in mid-air, which directed it at the ceiling and illuminated the whole motion-capture space (see Figure 1.1). Visitors would enter the room from the other side of the room, unable to see the work spaces and motion-capture space before walking past the screen. The first thing they saw instead, would be the mirrors, the reflections of the agentfigure in them, and the screen from behind, which would also show the agentfigure. The Visitors had to walk through the beams of light emitted from the projector and reflected by the mirrors and would often stop to observe the mirrors on the sides and on the ceiling, before entering the spaces behind the screen.
The motion-capture space was laid out with black flooring, for the purpose of showing Visitors what portion of the room was focused on by the cameras, equalling the portion of the room they could freely move in while being tracked. Roughly in the middle of the flooring was a spot marked with a subtle X-symbol, which served the purpose of showing Visitors where to stand during the donning phase (see Section 3.3), because the motion-capture software required new persons to stand within a predefined position in the room at the start, so as to accurately locate and capture them. Leaving the black flooring would cause some cameras to lose the Visitor's image, thus increasing the likelihood of problems in capturing their motion data. During the final installation, a "bed" akin to an extremely low table covered in pelt and a larger piece of felt fabric, functioning as a potential blanket, were sometimes placed into the motion-capture space prior to the arrival of Visitors, which can be seen in Figure 3.2. As the final installation used the sleeping agent (see Subsection 4.1.4), these objects could be used as a part of the interaction with said agent.

3.3 Donning

Regardless of which version of the project is being used, the Visitor has to go through a set of pre-defined motions first, to enable the motion-capture software to estimate body proportions. This process was dubbed 'donning', as the Visitor could see a visualized skeleton being laid over his own body, akin to clothing or armor, on the Captury Node screen (see Figure 3.3). Such skeletons had the option to be saved within the software and used again at a later date. This functionality proved to be insufficient however, as the data used to find the corresponding person to each skeleton was based solely on color

Figure 3.2: View onto the motion-capture space and the screen (Picture taken by Günter Richard Wett)

data delivered by the cameras, thus causing slight changes in color, such as different clothing or even different times of the day, to cause mismatches. It was thus decided, that every Visitor had to start with the donning phase before they could interact with the virtual agentfigure on the screen. During development, it was tried to have the agents on the screen lead Visitors through this phase. This lead to multiple problems, such as Visitors realizing the agentfigure as autonomous before emancipation has occurred. Furthermore, as the agentfigures' communicative abilities were limited to gestures, it proved hard to understand to first-time Visitors, which in turn lead to suboptimal results during the donning phase. The project team realized the importance of this phase being executed as well as possible, as improper results lead to erroneous tracking and thus increased the amount of unwanted behavior by the agent. Thus it was decided to explain the donning phase to Visitors using on-screen text and audio-based instructions.
The donning process itself consists of multiple suggested motions and postures, which are supposed to be performed and sometimes repeated by the Visitors, to allow the motion-capture software to make accurate estimates of their body proportions. The first pose consists of upright standing, with both arms held about 30 centimeters in front of the chest. Shoulders, elbows, and hands would be roughly on the same level under optimal conditions. After assuming such a pose, Visitors are asked to move their

upper bodies and turn around on the spot, before having to move their lower bodies and perform stretches and similar motions. Results of the donning process vary greatly and are dependent on various factors, including lighting conditions, the color of worn clothing, time since the last system calibration was performed, as well as the speed in which the aforementioned motions were performed. Most of these factors could be contained by various methods, but unpredictable Visitor behavior always represented a risk for problems occurring in the process of donning.

Figure 3.3: Some Visitors with their assigned skeletons drawn onto them.

3.4 Jason

Jason is an open source software developed by Jomi F. Hübner and Rafael H. Bordini [BH+04] [BHW07]. It offers a development platform for MASs using an extended version of AS, which is an agent-oriented programming language designed by Anand Rao in 1996 [Rao96] and based on the BDI model. This chapter offers a short introduction to the various concepts related to Jason, and shows how AS is implemented using the Jason MAS.

As was explained in Section 2.3, Jason is developed in Java and allows for a high degree of customization, although some critical areas are impossible to change without changes in the source code. It offers a platform for both developing and executing agent systems,

either locally or employing a network, and is further extendible by other agent-system frameworks, such as CArtAgO [RVO06] or JADE [BPR99].
Jason was chosen because of low complexity and being easy to learn. Furthermore, it fit the project's requirements and allowed customization in critical areas. It was also recommended by experts in the field of AI development. As is further explained in Section 5.2, in hindsight, other platforms and approaches might have worked as well, or even better, for the task that was at hand.

3.4.1 Agents

Agents in the context of AI, often also referred to as IAs, describe autonomous entities, which are able to perceive, as well as interact with their environment. More specifically, IAs perceive the environment through sensors and act through actuators (see Figure 3.4). IAs are rational, which means they must act in ways that maximize their performance and progression towards their goals, based on their currently held knowledge at any time [RN16]. Depending on the type of IA, it might use a history of everything it perceived so far, or only beliefs it holds at the moment, to make a decision.
IAs are implemented using agent programs, which are responsible for defining which actions are to be taken after new information is perceived. Agent programs can be implemented by following various design patterns. Jason employs goal-based agents, which use pre-defined plans to reach their active goals. Agent programs are then embedded within a certain architecture, which supplies sensors and actuators. Jason offers this architecture in the form of an environment, which is implemented by a customizable Java class. Usually, the architecture functions as the interface to the physical world, in which sensors and actuators are used. In the case of this project however, agents only exist and act within a virtual world, with some sensors providing data from the physical world, while others are targeted at the virtual world. The agents are thus not directly able to interact with the physical world, while perceiving both spaces. For the sake of the abstract model shown in Figure 3.4 however, there is no distinction necessary between the physical and the virtual space. Agents are thus defined as the combination of an agent program and an architecture [RN16].

3.4.2 Multi-agent Systems

Agent systems, more often MASs, are systems consisting of multiple IAs. A MAS is able to host multiple IAs acting independently of each other, which are also being able to communicate and interact with each other, as well as with the environment. Thus MASs offer a distributed system for IAs to work within, allowing for solutions to complex problems which might not be possible by using centralized approaches. Especially if the problem itself is distributed in nature, using a MAS can lead to efficient and easy-to-understand solutions [FW99].

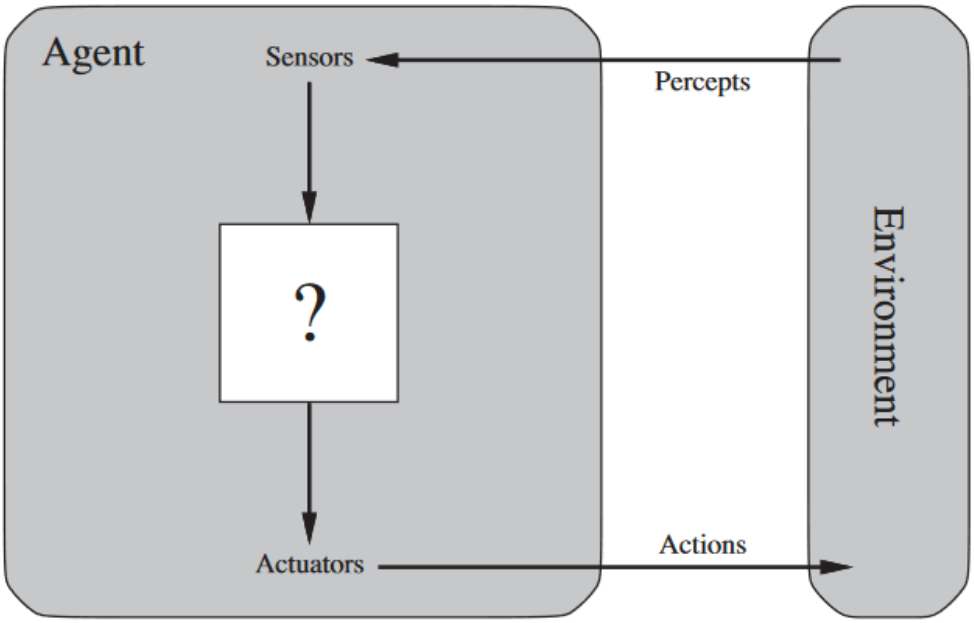

Figure 3.4: Abstract model for agent-environment interaction. (Source: [RN16, p. 35].)

3.4.3 Belief-desire-intention Model

The BDI model is based on a psychological model by philosopher Michael E. Bratman, who was also involved in developing the software model. The BDI model separates the concepts of beliefs, desires, and intention within the IAs, to allow them to both choose what to do, as well as follow through with their chosen plans. The plans to be executed by the IAs however are usually not created by the agents themselves, but are designed and implemented by their developers [Rao96].
Beliefs describe all information an IA has about is environment. They are either perceptions about the environment, introduced via sensor data, or mental notes established by the IAs themselves – as a result of other beliefs, events, or their own actions. Beliefs do not have to be true and can change in the future. In addition, they can also possess a percentage-based truth value to represent the likeliness of being accurate. Adding or removing beliefs from an IA causes respective events to trigger, which in turn may modify other beliefs or trigger plans to be executed.
Desires represent states that individual IAs want to accomplish, regardless of whether they are currently acting towards fulfillment of such states or not. As soon as the IA starts pursuing a desire, it has acquired a new goal. Contrary to desires, goals must be mutually exclusive. This means that IAs may need to make decisions on which single desire to act on, if multiple possible but mutually exclusive desires are present.
Goals are reached by following plans, which consist of multiple actions, which in turn may be represented by other plans. Furthermore, a single goal could also be reachable by multiple plans, which could all be possible to follow at a single point in time. In

such cases, the agent is expected to choose only one applicable plan. In the case of the Jason MAS, this can be done in two ways – either by making only one goal viable to be pursued at a time by defining mutually exclusive contexts (pre-conditions), or by using custom selection functions defined within the Java-based environment. If neither are implemented, the default selection function chooses the first applicable plan instead, which may not be the most cost-efficient one. Once the IA has actively started executing such a plan, the desire which caused the plan to be followed is referred to as an intention. Intentions are thus defined as desires an IA has committed to.

3.4.4 AgentSpeak

AS, formerly called AgentSpeak(L), is a logic-based, agent-oriented programming language, which was originally designed as an abstract language without actual implementation, for the purpose of gaining insight and understanding of the BDI model [Rao96]. The language offers a way of equipping agents with knowledge about certain pre-defined plans, also referred to as know-how, so that agents can make use of those plans to fulfill their goals or react to changes in their environment. AS, as well as Jason, were designed for cooperative agents, focusing on allowing IAs to communicate and interact with each other easily. A brief explanation on AS syntax, as it is used in Jason, can be found in the next subsection.

3.4.5 Jason Syntax

Jason offers an actual implementation of the AS language, while also adding additional functionality – for example in the form of various customizable components within the agent and environment Java classes. Agents within AS are defined by their initial beliefs and their plans. A simple example for this can be seen below.

```
alive.

+alive
<-  .print("Hello World").
```

The first line specifies an initial belief, which the IA adds to its belief base as soon as it starts being active. The belief is saved as a literal, similar to traditional logic programming languages such as Prolog. Such literals often take the form of predicates, to express properties of certain objects. The literal in this example however, is a so-called atom, which displays no properties except for existing within the belief base. The period following the initial belief functions as a syntactic separator, similar to semicolons in Java.
The second line displays a very simple example of a plan. The first part, `+alive`, defines when the plan is to be executed. The plus symbol represents the addition of a new belief, thus the plan will be executed whenever the belief `alive` is added to the belief base. The arrow operator separates the plan's head from its body, where instructions can be

found. In this case, there is only one instruction, which is an action to be executed as soon as the plan is started. As can be seen in the example, the plan will print out the words `Hello World` into the console, whenever it is triggered. If the agent were to be started now, it would first add a new belief in the form of `alive` to its belief-base, which would cause an event to trigger. This event is represented by `+alive`, which happens to have an applicable plan assigned to it, which is then executed. It is important to note, that the content of the belief is insignificant in this case. Instead of `alive`, other words or expressions could be used in the same way, as long as they start with a lower-case character, which causes them to be recognized as atoms. An example for a more complex plan can be seen below.

```
+alive
    <-  !eat;
        .print("I have finished eating").

+!eat : not eaten(_)
    <-  eat_food;
        +eaten(1);
        !!eat.

+!eat : eaten(X) & X < 5
    <-  eat_food;
        -+eaten(X+1);
        !!eat.

+!eat : eaten(X) & X >= 5 .
```

The first plan looks somewhat similar to the one in the first example. It is triggered with the addition of the belief `alive` and finishes by printing out a message. In between, there is an additional step, which starts with an exclamation mark and ends with a semicolon. The latter separates multiple steps of a single plan, as opposed to the period, which concludes plans or initial beliefs, whereas the exclamation mark precedes goals. In this case, a new goal is added, which is `eat`. Only after the goal has been fulfilled, the plan continues being executed, and the message is being printed out on the console. The newly added goal `eat` has three plans associated with its addition, which are marked by `+!eat`. Furthermore, all of those plans have an additional component in their plan head, which is placed in between a colon and the aforementioned arrow operator. This part is called the context, which works like a pre-condition for the plan to be chosen. It must offer a binary result, based on which the plan either becomes viable to be chosen or not. The first context reads `not eaten(_)`, which means the plan can be chosen as long as there is no belief called `eaten` within the IA's belief base, with any value associated to it, as the underscore represents a wildcard. The second context is satisfied as long as the value held within the belief `eaten` is smaller than five, whereas the third

context will become satisfied as soon as the same value equals five or higher. This code thus represents an example of three mutually exclusive plans for the same goal, as was discussed in Subsection 3.4.3.
The first plan for `+!eat` executes the action `eat_food` as its first step. This action has to be implemented within the Java-based environment class and return a success flag, for the plan to be able to continue. After the action has finished, a new belief is added in the form of `+eaten(1)`, in which the plus represents the addition of the belief that follows after it. The belief adds the attribute of `eaten` to the value of `1`, which could also be an atom, such as `apple` for example. As stated before, the addition of a belief also triggers an event, however there is no plan associated with `+eaten(1)` or `+eaten(_)`, thus no plan is executed. The last step once again causes the agent to pursue the goal of `eat`. The double exclamation mark causes the current plan to continue without waiting for the newly added plan to finish. In this case, it serves only as recursion optimization, as it does not leave the finished `+!eat` plans on the stack.
The second plan is mostly the same, with a difference in the second line. Expressions starting with upper-case characters in AS function as variables, in which other literals can be stored. In this example, `X` is unified with every available value for the attribute of `eaten`, causing it to take the value of `1` the first time it is called. `X` is used again in the expression `-+eaten(X+1)`, which causes all beliefs of `eaten` to be removed from the belief-base first, before adding a new `eaten` belief with the value of `X+1`. The third plan for `+!eat` does not have a body at all, which means that the goal is fulfilled as soon as this plan is chosen. It thus serves as a termination condition for the recursive functionality of the `!eat` goal.
In conclusion, the example code above would cause the agent to `eat_food` five times, before stopping and printing out a message. For this to happen, it would need to start by adding the belief `alive` to its belief-base, for example in the same way as in the first code example.

3.5 Input Data

Input data refers to all data sent from the rendering engine to the Jason MAS. The kinds of information that are transmitted this way can be seen in 3.1. All input data is processed by the Java thread receiving the information via UDP. The thread then introduces the pre-processed data into the simulated environment. If data of the same kind was already available, it is updated instead. For example, positional data of all active agents and Visitors is updated periodically. In some cases, the data is instead only removed from the environment, because it is no longer up-to-date. Furthermore, information in the environment can be made visible to all agents or only to specifically chosen ones. A supervisor agent, which does not have a visual representation on the screen and only runs on the Jason platform, is responsible to orchestrate the agents respective to the currently active scenario, telling them when to start and when to stop their performances. As such, some of the data is only made available to the supervisor agent.

Information content	Reference within agent source code
Visitors' gestures recognized by FUBI or by the rendering engine	actorstate
Visitors' position within the physical room	actorposition; actordistancetoscreen
Visitors' movement speed	bodymotion
Donning status (percentage; 100% means donning is finished)	actorscaling
Information regarding the active camera	camerapos; cameraborder
Positions of objects or artifacts	objectposition
Current scenario (story)	scenario
Agents' current action (for example: idle, live, mid-animation)	agentstate
Agents' position within the virtual room	agentposition

Table 3.1: Information transmitted from the rendering engine to the Jason MAS

All data is sent from the rendering engine, where it was processed from input coming from The Captury, calculated using the physics engine or script functions, or forwarded from FUBI. Use of FUBI was minimized in the later stages of the project, as it oftentimes proved not reliable enough. Instead, custom scripts within the rendering engine were developed to detect specific gestures and send relevant information to the agent system. During development, some calculations were made within the MAS, by the agents themselves, which proved to be inefficient and oftentimes produced code that was very hard to read and comprehend afterwards. Thus it was changed later and calculations were almost exclusively performed within the rendering engine or within the Java-based environment.
During development, one of the problems that became clear was latency caused by the communication between the rendering engine and the MAS environment. As both were usually executed on the same machine, network latency did not pose a problem. However, processing of received data on the side of the MAS was handled by multiple queues, as well as Jason's inherent reasoning cycle, which is responsible for detecting changes in the perceived environment, thus updating the currently held beliefs of all agents, and subsequently triggering all relevant events linked to any changes in beliefs. As it turned out, it was possible for this system to be overloaded with input data, which in turn caused it to lag behind on execution and ultimately caused it to not be able to react in time to any changes. It became apparent, that it was important to control the amount of information sent from the rendering engine to the MAS, so as to ensure flawless execution and timely reactions. Because of this, the rate at which constantly changing information was sent out from the rendering engine had to be limited to a frequency which was less than the frame rate of the rendering engine, whereas one-time changes were always sent out immediately.

3.6 Output Data

Output data refers to all data sent from the MAS to the rendering engine, which consists mostly of actions the agents wish to take, but also includes meta-information like currently active scenarios and signals regarding the state of the agent system in general, such as whether it is ready to be used. To send out orders to the rendering engine, Jason agents use custom actions, which are defined within the environment class, which is written in Java. The environment class uses threads to communicate with the rendering engine, sending BSON-encoded packets via UDP.
Commands issued by the Jason agents are either related to a pre-recorded animation, a desired state, or both. Examples for animations include gestures like waving or shrugging, as well as general movements like jumping or swooping. States are divided into the three main states, of which only one can be active at a time and which include "idle", "Live", and "animating", as well as meta states, which can be active in combination with the main states. "Idle" is the default state, in which the agent stands upright and plays a subtle animation loop to appear unengaged. The Live state causes the agent to mirror the Visitor's movements. Switching into Live is not always easy however, for reasons which are further explained further below. "Animating" is a common state for all animations, which is used whenever the agent is currently in the middle of executing a pre-recorded animation. Meta states are oftentimes coupled with a certain animation, of which they usually come as a natural result – for example "lying down" causes agents to switch to a "sleeping" state. Furthermore, some states and commands are exclusive to certain scenarios, such as the bat scenario, which included states for the agent appearing upside down on the ceiling of the virtual room within the screen, as well as commands to switch from the ceiling to the floor or vice versa.
Because agents are not directly in control of their actions, as was described in Section 3.1, but rather send commands towards the agentfigures within the rendering engine, they need to know whether the action was executed and furthermore, if the action caused any changes within the environment. AS and Jason generally define every agent action as a process that might be successful or unsuccessful. This implies that agents get feedback whether their actions succeeded or not. If custom actions are implemented using Java, these actions must terminate with return-values indicating success or failure, so that the agent can act accordingly. If a single action takes a while to complete, the agent pauses its currently followed plan and waits for a response on whether the action has succeeded or not. However, this proved to be problematic in the scope of this project for many reasons. One of which was the fact, that agent actions had to be interruptible by the Jason agents themselves, in case the situation changed and the agent had to react. In some cases, single animations spanned timeframes of more than ten seconds, in which agents might have not been able to react to changes. Jason's framework offers solutions to this problem as well, such as concurrent plans and reactions to new events being able to cancel and discard other currently pursued goals. Such an approach proved to be in need of detailed error-handling, as agents would otherwise cancel their goals too often without replacing them, causing them to lose all active behavior in many cases.

Furthermore, it was oftentimes unforeseeable how long certain actions would take. For certain actions, such as synchronizing with the Visitor or interacting with an artifact within the virtual room, the virtual agentfigure would have to move to a certain position first, before being able to execute the action. Depending on where the agentfigure was located before the command was issued, as well as other circumstances, the time needed to complete the action could vary greatly. This makes it harder for the Jason agents to plan beforehand on when to issue follow-up steps or whether or not they are in a position to react to certain events.
Instead, Jason agents in this project were caused to behave differently. External actions, which would be sent out to the rendering engine, always return successes as soon as the packet is sent out. Thus, the agents command flow will not be interrupted and agents can continue on with their plans. However, Jason agents are made to doubt whether their actions are actually successful or not, thus making the agents check the status of their respective visual agents, to determine if the actions have actually been performed after they have sent out the order. For example, whenever the agent would perform a SwitchToLive, it commands the visual agent to synchronize with the Visitor and to start copying the Visitor's movements. However, it is not clear beforehand how long the virtual agentfigure will take until it starts mirroring the Visitor, as it has to move to the same position within the mirrored reality before being able to start. The Jason agent is nevertheless able to continue its plan after it sent out the command. It is capable of checking the state of the visual agent, whether it is Live or not – meaning whether it has successfully started mirroring the Visitor, and thus determine when to continue with other parts of the plan that depend on that state. Yet if the Jason agent wants to interrupt the process of the visual agent synchronizing with the Visitor, it can do so at any time as well, by sending out a new command.
This behavior could effectively also be performed by multi-threading agents, which would pause one plan while waiting for an action to finish, while concurrently following other goals, which would take over in case the action needed to be stopped. Such a design was also employed whenever the agent needed to react to changes in its environment, as changes were also able to cause the agent to pursue a new goal. For the story-based main goals of this work however, this proved complicated, as splitting them up into multiple goals was not an easy task and proved too complex within the scope of this project.
This decision also made it possible to perform smooth transitions from one pre-recorded animation to the next, as Jason agents were allowed to queue up multiple animations while the visual agent was still mid-animation. If Jason agents had to wait for a response from the animation engine to confirm their command has been executed successfully, there would have been a brief window, after the visual agent finished the animation and before a new command was sent out by the Jason agent, in which there were no clear instructions on what the agent within the rendering engine was supposed to do. In this project, the visual agents defaulted to their idle state in such cases, in which they stood upright and performed a looping idle animation. Even if only a fraction of a second passed in between the end of one animation and starting the next one, the idle state would be clearly visible to spectators of the visual agent on the screen, often in the

form of quick and erratic movement. To prevent this from happening, Jason agents were given the ability to queue up commands for execution by the visual agents instead. This allows animations to chain into each other without triggering the default idle state, thus creating more natural looking agent movement.
Furthermore, communication between agents, to share beliefs, intentions, or other information is directly supported in Jason. Agents were thus able to warn other agents if they intended to perform certain actions, tell them what to do, or request information or cooperation. This was commonly used while developing multi-agent scenarios including artifacts, so as to not cause both agents to interact with the same artifact if that was not possible. It was also used as a shortcut to add specific new perceptions to other agents, for example if one of the visual agents waved at the other, the waving agent also directly communicated with the other agent at the same time, instead of letting the second agent perceive it via the environment. This is counter to the design philosophy of the agent system, but proved to be adequate. A possible scenario, in which differences may occur, would be if the second agent was looking in the opposite direction and would not be able to see the first agent waving at them. However, cases like this could also be handled by polling information about positions and directions of the agentfigures directly by the agents themselves.

CHAPTER 4

Implementation

This chapter goes into detail about the process of developing and implementing the agent system using Jason. First, an initial approach is described and explained, which was developed and tested right after work on the agent system has started. Some specific agent stories are highlighted, as well as their results explained. Afterwards, the final approach, which was implemented after multiple discussions within the project team and consulting with experts, is introduced and explained. Finally, this chapter also offers insight into how communication worked within the core team of the project, using visualizations to plot and discuss various agent stories. In this context, stories refer to textual or visual representations of what is expected of an agent, whereas scenarios refer to the implementations of said stories in the form of Jason agents within the MAS.

4.1 Initial Approach

During the initial approach, various stories that had been written and designed previously by the project team were to be implemented into a MAS using Jason. As described in Section 1.1, at the start of development, the rendering engine was able to draw an agentfigure onto the screen, which would proceed to mirror any movements of Visitors tracked by the motion-capture system. The Jason agent was then supposed to infuse the visual agent with a will of its own, so that it could emancipate itself away from being a mere reflection of the Visitor. The agents' behavior, movements, animations, goals, and reactions all differed depending on the active scenario.
Two of the first stories developed by the project team were the octopus agent and the bat agent. The way in which those stories were brought to paper can be seen in Figure 4.1. As can be seen in the intention part of the figure story, additional features like a visible environment within the virtual space, as well as the use of sound to enhance the experience were originally planned, but not implemented completely within the final product. Before those first two stories were implemented however, it was decided to develop a simpler

story for purposes of setting up the environment properly and getting a general look at how agent development in Jason worked within the scope of the project. This simple story became known as the waving agents, featuring two active agents with the goal of causing at least one of the Visitors to wave their hand towards the screen.
Throughout development of the waving agents, it became clear that their behavior could be divided into somewhat independent phases. Phase zero usually consists of the donning phase (as described in Section 3.3), in which the Visitors' body proportions are estimated by the motion-capture system, to allow the software to track visitors' movements throughout the designated motion-capture space. After donning is completed, the so called Live part follows, in which the agentfigure would synchronize with the Visitor's position on the other side of the screen and start mirroring the Visitor's movements.

FIGURE

Description

Name	Octopus
Speciality	flexible, precise
Mesh	
Texture	XX(fluid, instable, glow)
Camera	Perspective
View	Front
Size	1:1

Intention

Behavior	Octopus is constantly repositioning its body into, under, between (invisible) niches, gaps, constraints; measuring, folding the body, fitting and pushing into molds. Fig.
Environment	black sheets, water
Sound	invisible resistance (?); water

Figure 4.1: Octopus agent story, written by Christina Jauernik.

4.1.1 Waving Agents

As was mentioned above, the waving agent scenario features two agents and requires two Visitors. Each Visitor is assigned their agentfigure, which becomes apparent to the Visitors, as soon as the respective agents figures start mirroring their movements. The end-goal of the scenario consists of getting at least one of the Visitors to wave at the screen, towards the agentfigures. The origin of the scenario lies within already implemented functionality of the FUBI framework, which made it possible to recognize waving as a gesture performed by motion-captured individuals. Furthermore it was argued that IAs necessarily needed an ultimate goal, so as to be able to act within the Jason MAS – which is why the goal of getting the Visitors to wave was defined as such. In the first phase, after the Live part, emancipation occurred and the waving agents start pursuing their scenario-specific goals. During this stage of development, the trigger for

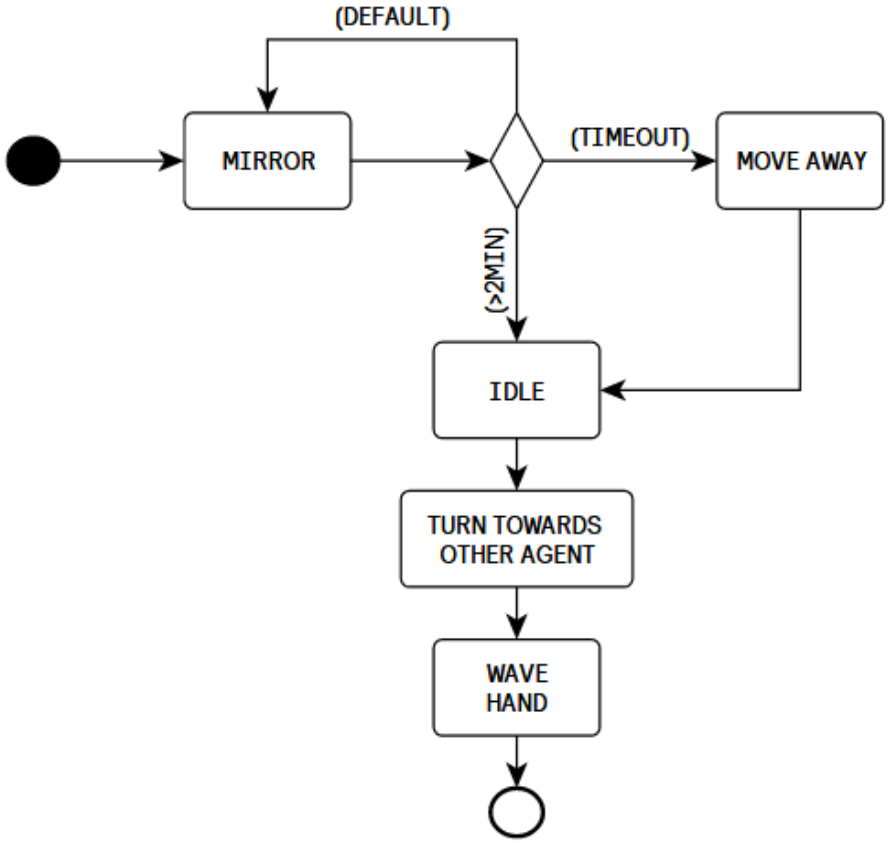

Figure 4.2: Waving agent model, phase 1, digitalized by Simon Oberhammer.

agents to stop being Live and start moving on their own is purely based on time. At first it was decided to have this behavior start exactly two minutes after donning was completed, although this time span was later reduced to a random amount in between 50 and 70 seconds. As soon as both agents are finished with Live, the first agentfigure turns towards the second agentfigure and waves at them. If there is not enough space in between the two agents to make the animated gesture certain to be visible, the first agent decides to move away from the other agent on its own, so as to create enough space. After the first agentfigure has successfully executed the waving animation, phase one ends and phase two starts. A simple model depicting phase one, using notation similar to that of UML activity diagrams, was created by the project members and can be seen in Figure 4.2. The visualization only depicts the process of the first agent's activities

and simplifies the way the agent checked for distance between the two agentfigures.
Within phase two, the two agentfigures continue by waving their hands at each other, before turning towards the camera and waving towards the Visitors as well. In case at least one of the Visitors waves back, the figures express joy by clapping their hands, before switching back to Live, mirroring the Visitors' actions indefinitely. If the Visitors do not wave back, the two agents try three more times by waving towards the camera, before giving up and fading away from sight. The scenario was later extended by an additional goal of getting both Visitors to wave back towards the agentfigures.
The scenario generally worked well after several iterations. However, problems with the FUBI framework arose early, causing it to be unable to recognize waving Visitors consistently. Furthermore, the binary result of either celebration or failure made the scenario very similar to a game in nature. This went against the idea of the installation, as the idea was to distance the experience from games and to promote more natural interaction between Visitors and the agentfigures. Development on the waving agents was thus stopped, and development on the previously mentioned animal-based agents begun.

4.1.2 Octopus Agent

The octopus agent story was one of the first ones to be implemented as an agent scenario. The general concept can be understood from Figure 4.1, which offers descriptions of both the octopus agentfigure, as well as a its behavior and environment. The agent itself can be considered more of an experimental agent, based on the idea of the magic mirror, which was embodied by the screen. The octopus agent is supposed to interact with its respective Visitor's body reflection, for example by squeezing the agentfigure's body into the empty space between the Visitor's arms and chest, or similar openings. Different from a magic mirror installation however, the Visitor's body was not visible on the screen, but only the agentfigure.
The idea of the octopus agent was ultimately not pursued or developed very far, as it proved too complicated to record fitting animations and use them appropriately. Additionally, the concept itself was very confusing to Visitors. Combined with suboptimal agentfigure movements and decisions, Visitors did not properly understand the mode of interaction. Thus the whole scenario was deemed unsuitable for the given project scope.

4.1.3 Bat Agent

The bat agent story focused on an agent with bat-like, animalistic behavior. The core concept of the bat is the ability of the agentfigure to switch between standing on the floor and upside down on the ceiling. For example, the bat agent would start off by sleeping upside down on the top of the screen, before waking up and eventually moving down to the floor to find itself on the same level as the Visitor.
The agent scenario is focused on various kinds of interactions and different outcomes depending on the Visitors' behavior. It was the most extensively developed agent scenario up to this point, with a large number of iterations, as well as many concepts which

were tried out before being removed from the agent scenario again. Different from the waving agents, it is designed for one Visitor at a time, although multiple bat agents could be employed for multiple Visitors. In the case of multiple bat agents being active at the same time however, they will not interact with each other – the only inter-agent communication that occurs is to prevent the agents from overlapping and occupying the same spot of the screen.
Another noteworthy feature of the bat agent story is that agents are able to show more open hostility in interactions with Visitors, for example by attacking or swooping towards them. Such animations are hard to interpret by Visitors however, as the missing dimension of the orthogonal projection, in addition to the impossibility of physical interaction between agentfigures and Visitors made it hard to tell what the agents are aiming at.
Bat agents were the first agent scenario to use emotional scales, such as for example the level of timidity ranging from zero to one hundred. The value of those scale variables would change depending on actions of the Visitor, and would influence how the bat agentfigure would react to certain events. The waving agents had a similar concept, in which their emotions would simply change from one state to the other – such as either being happy, sad, or neutral. In later agent scenarios, such numerical values are further used to impact other dimensions of the interaction, such as the speed of the agentfigure's animations or the angle in which it would face the Visitor.
Furthermore, bat agents have the ability to deploy additional agentfigures and give them commands. These agentfigures have various roles, such as instructing the Visitor during the donning phase, while the bat agent figure was sleeping upside down on the ceiling, or appearing next to the sleeping bat in a similar manner, such as to appear like a group of sleeping bats.
The bat agent scenario was in development significantly longer than previous agent scenarios. However, it was ultimately stopped and abandoned in favor of a different scenario, because the bat agentfigure's behavior was deemed too hard to understand and the upside-down, pre-recorded animations did not turn out the way the project team envisioned them. Additionally, the bat agent's behavior was deemed too reliant on reactions to Visitor actions. The goal was instead to develop an agent that interacted with the Visitor, but would also provoke interaction in case the Visitor did not act first. This meant that the agent story had to encompass both reactive parts to Visitor input, as well as pro-active parts for the agent to pursue their own goals. The agent story introduced in the next chapter focuses more on autonomy and pro-active behavior of the agentfigure.
A small part of the bat agent program's source code can be seen below. It shows the reaction of the bat agent to a certain Visitor action under certain circumstances. Most of the bat agent program consisted of similar reactions to Visitor input.

```
// visitors putting their hands together while agent figure
// remains on the ground causes bat-like agent figure to go
// back to the ceiling
```

```
+actorstate(ACTOR, STATE) : my_actor(ACTOR) & not animating
& phase(3) & STATE == handstogether & onGround
    <-  +animating;            // currently mid-animation
        -onGround;             // not on ground level
        switchToIdle;          // stop moving, go idle
        !waitForIdle(250,0);
        switchToCeiling;       // agent figure to ceiling
        !waitForIdle(2000,0);
        switchToLive;          // mirror visitor movements
        -animating.            // no longer mid-animation
```

4.1.4 Sleeper Agent

The sleeper agent scenario went through the highest amount of iterations and was eventually used in the final installation of the project. The agent story's main idea was based on a figure that gets exhausted through movement and wants to go to sleep, to recover, before being able to interact with the Visitor again. Similar to the bat agent story, it was designed for one Visitor at a time, while utilizing multiple instances of the same agent in the case of multiple concurrent Visitors. As can be seen in Figure 4.3, the actual agent story and the implementation both took the form of a finite-state machine after several iterations of development.
After three minutes of mirroring the Visitor in the Live state, the agentfigure will emancipate and distance itself, seek the invisible virtual bed within the virtual space, and lie down to sleep. If the Visitor joins the agentfigure in lying down – either on the ground or on the provided bed (as explained in Section 3.2), the agent would warm up to the visitor, by increasing an internal value called sympathy. This value would increase whenever the agentfigure was asleep and the Visitor was either detected as lying down, based on specific bones of the body being within a certain vertical range of each other, or as long as the Visitor was close enough to the screen. In the same way, if the Visitor moved away too far from the screen and was not lying down, it would cause the sympathy value to decrease.
The goal or milestone of the scenario during the iteration displayed in Figure 4.3, is to arrive at the dream sequence after the Visitor lies down with the agentfigure for long enough to reach the upper limit of the sympathy value. This sequence would play pre-recorded audio and offer an ending to the experience, after which interaction with the agent is no longer possible. This was the first time a state with no way to interact and no way to change it was introduced, as it was originally planned to continue the story after this point-of-no-return with a second milestone in mind, similar to the waving agents which change their goal from having at least one Visitor waving towards the screen to having both Visitors waving their hands at the same time. This second milestone was however never implemented before the whole scenario for the sleeper agent was reworked.
Other noteworthy states are also part of the scenario, which implemented various experimental features, such as changing cameras or environments. During the deep-sleep state, which occurs after the sympathy value is raised high enough, the camera changes to

a perspective camera, placed inside of the agentfigure's head – thus granting the Visitor sight of what the agentfigure would see. Because the virtual space the agentfigure resides in is completely empty most of the time, the first thing to appear on the screen in this state is usually nothing but black background. As the agent is Live during this phase, mirroring its respective Visitor's movements, it is possible for the Visitor to see the agentfigure's hands or whole body from the perspective of the agentfigure, if they move accordingly. Furthermore, if more than one Visitor is present and has an agent assigned, it was also possible to see the other agentfigure in this state. Differently placed cameras were also experimented with, such as one placed within the agentfigure's hand, or one placed within its knee. Further experimental features include visible environments within the virtual space, such as one filled with mirrors in which the agentfigures' reflections can be seen, although the mirrors cannot directly be interacted with.
The scenario furthermore includes varying animations for the same activities, such as lying down, sleeping, and getting up again. Various parameters, such as animation speed or rotation around the agent's own axis are also changed depending on the situation or the current sympathy value. As mentioned before when explaining the dream sequence, this scenario also incorporates sound, for the most part in the form of spoken language, but also in the form of unintelligible whispering for atmospheric purposes.
Because the story of the sleeper agent was mapped out as a finite-state machine (see Figure 4.3), the implementation follows this model as well. Each state of the state machine is implemented as a separate plan, which is triggered whenever the goal of pursuing the respective state as a goal is added. These goals are added via various means, such as finishing a single state's plan, as a reaction to Visitor behavior, or by reaching certain sympathy values. This can also be seen in the agent program code shown below.

```
+sympathy(S) : max_negative_sympathy(MNS) & S <= MNS
    <-  !!switchToPhase(11);
    .drop_desire(adjust_sympathy).

+!phase(11) : true
    <-  environment(narcotic);
        setTimedZoom(0,1);
        setXRotation(0, 10);
        playanimation(narcotic_sleep_loop, 1, 1, 1);
        ?my_actor(ACTOR);
        ?actorposition(ACTOR,X,Y,Z);
        -+narcotic_position(X, Y, Z).
```

Commands starting with question marks symbolize test goals, which attempt to retrieve information from the belief base. In this case they are used to initialize variables which are later on stored as a separate belief.
It is important to note, that this implementation and usage of Jason – in the form of a state machine – is not the way the Jason platform was intended to be used. It does not

follow the principles of agent programming, such as utilizing delegated goals or agents balancing being goal-driven and reactive. Delegation of goals implies telling an IA what to achieve, but not how to achieve it, so that the IA can pick the appropriate plan to achieve its goal by itself. Because this implementation offered almost no alternative plans for the various states, there was no flexibility available to the IA. Furthermore, the various state-machine-based goals are oftentimes not true goals in the sense they are supposed to be in the context of an agent platform, as they do not describe a concrete state of affairs to be brought about. On the other hand, it could be argued that reaching a certain state within the state machine is also a state of affairs, although a more abstract one. Further guidelines and concepts of Jason and AS which are not represented well in this scenario include responsiveness, as agents would oftentimes completely ignore Visitor actions of all kind during certain states, as well as inter-agent communication and cooperation, which is after all one of the major features and selling points of the Jason MAS.

The results of this implementation are accordingly unsatisfactory. As a polar opposite to the bat agent's behavior, which consists of almost only reactive parts, this incarnation of the sleeping agent almost never directly reacts to the Visitor's actions. Interaction is always very one-sided, with the agent performing in front of the Visitor, occasionally verifying the Visitor's position and gestures. Starting with the donning phase, the Visitor is instructed by the agentfigure, before the agent goes Live and allows the Visitor to experience the agentfigure mirroring their movements. Afterwards, the Jason agent starts telling its story and interaction is further reduced, as the agent is animated most of the time and the Visitor's role is reduced to that of a spectator, with very limited and unclear options for interaction.

The story itself can unfold in a couple of ways, akin to branching paths that all lead to the same end, which is represented by the dream sequence. However, because there is no real interaction between the Visitor and the agentfigure, it was considered by many Visitors to be more similar to watching a movie or a lecture, than to interpersonal interaction. Repeated Visitors also noticed how the story will always lead to the same end with not too much variance, as some of the optional states are quite hidden, because of specific pre-conditions which are unclear to onlookers – such as standing up again at specific point in time after lying down. Project members also expressed their wishes for more variance, less predictability, more possible interactions, and most importantly, indeterministic behavior instead of deterministic one. This meant, that randomness should play a larger role within the agent scenario, and that the agent should not always choose the same way to act or react under certain conditions.

After consulting with experts on how to continue, it was decided to scrap the current agent program and develop a new one from scratch. Because the general concept of the sleeper agent was kept, animation files and features within the rendering engine could continue to be used, thus reducing the amount of effort the fresh start entailed. However, as the agent program should no longer be based on a finite-state machine, development as well as graph-based visualization of the agent stories had to change drastically.

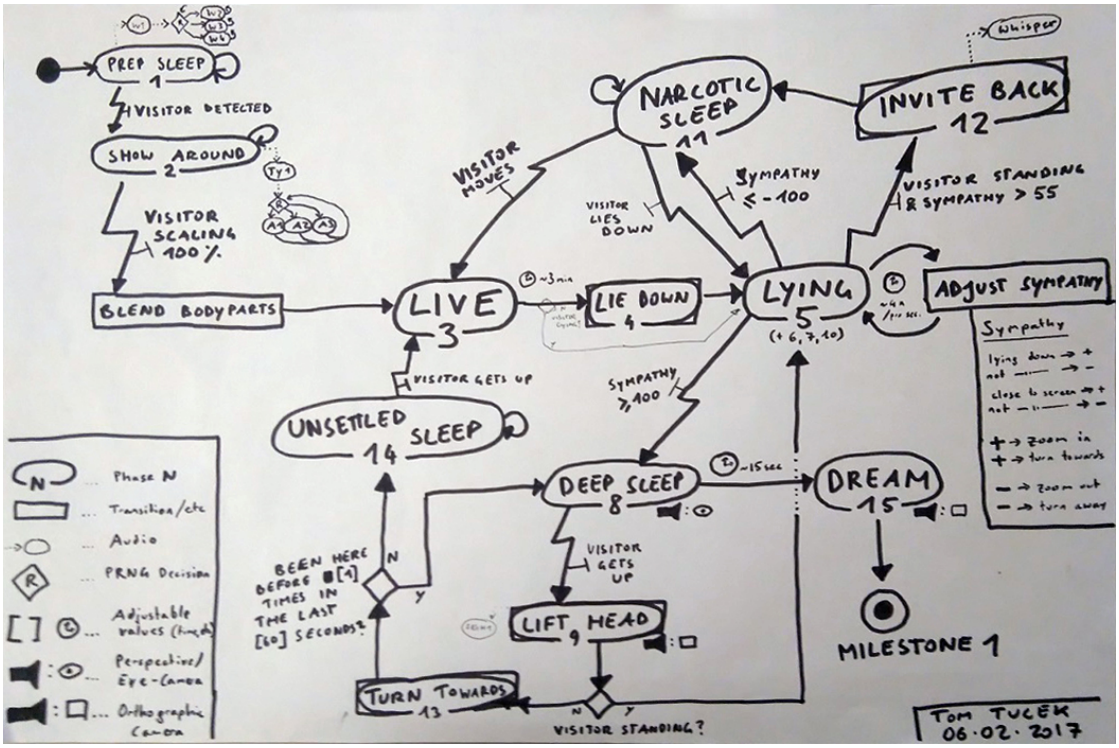

Figure 4.3: Sleeping agent state machine, hand-drawn during discussion.

4.2 Final Approach

During the consulting phase with experts from the field of agent development, it was suggested to take a look at other agent development platforms, which might be better suited to the requirements of the project. Explicitly, Concurrent MetateM [BFG+95] was recommended, which is a multi-agent language similar to Jason, but uses temporal logic to generate agent behavior. It would allow agents to act in less obviously formulaic ways, as plans would be automatically and incrementally constructed based on temporal logic rules within the agent program. This would allow agents more flexibility for varying actions under the same circumstances as well.
However, because of the late stage during the project it was decided to not switch to MetateM, as embedding it into the current environment, learning the appropriate usage of the language and the different way of thinking in regards to temporal logic would take too long for the project to be finished in time. Instead, a new way of implementing agents in Jason was inspired by MetateM, which was referred to as the island-system within the project team. It was based on an approach to cause more indeterministic and emergent behavior amongst agents, as was requested before. Agents are supposed to decide on their next actions on their own, instead of only reacting to the Visitor, but actions of the Visitor would also influence which actions agents could take next, in addition to direct reactions to certain Visitor behavior.
The general concept is based on a graph similar to a state machine, but with no direct transitions leading from one state to another. The graph also displays states in the

form or ovals, which are considered the name-giving islands within a vast ocean of white background to be navigated by the agent. Islands have pre-conditions attached to them, which must be fulfilled in order for the island to be accessible. If the agent wants to get to a certain island, it might have to cross other islands first, so as to fulfill the goal island's pre-conditions. Furthermore, if the agent has no specific goal, it considers all islands accessible to it at the time and decides indeterministically – at random – where to go. This approach is closer than the state-machine-based approach to the way Jason is intended to be used, but some elements, such as the random selection amongst available islands, can be considered violations of Jason's principles.
Various conditions were defined using logical rules in Jason itself, which allows use of rules in the same style as the Prolog programming language. Such rules, examples of which can be seen in the code below, are used as preconditions for the islands and were named based on the way the pre-conditions were visualized, as can be seen in Figure 4.4.

```
// position - left side
b1 :- my_actor(ACTOR) & actorposition(ACTOR, X, Y, Z) & X < -0.5.
// right side
b2 :- my_actor(ACTOR) & actorposition(ACTOR, X, Y, Z) & X > 0.5.
```

The agent story was once again called "sleeping agent", with its general behavior once again based on an agent that had to go to sleep after becoming tired. The agentfigure generally mirrors the Visitor's movements and accumulates fatigue based on the intensity of the movement. Fatigue is represented by an energy value within the agent program code and is decreasing if the agentfigure is in motion, while increasing if the agentfigure is resting. If the Visitor's movement got too insignificant, the agentfigure starts moving on its own, performing actions like strolling around the room or sitting down on an invisible chair within the virtual room. Such actions would cause the agent to lose energy as well. If the Visitor's movement gets more intense again, the agent returns back to the Live state. Compared to the game-like installation described in [BJNDL15], in which users had to mirror the virtual agent's movements and vice versa, otherwise the agent would lose interest and the game was considered lost, in this project's installation the agent mirrors the Visitor's movements and loses interest if the movement is considered insufficient in terms of activity level, which is calculated using the distance travelled by the Visitor's individual limbs over time.
If the agent's energy gets too low for it to continue moving, it will seek rest by going to sleep. While sleeping, the Visitor's interactions are once again limited. However, the agentfigure's sleep phase did not last as long as in the previous installments of this agent story. While the agentfigure is lying asleep in its virtual, invisible bed, the Visitor has the option to wait and observe, or to lie down as well, either on the floor or on their own bed, once again provided within the physical room as explained in Section 3.2. Because the agent will go to sleep multiple times during a single Visitor's experience, the Visitor has multiple chances to realize this opportunity. If they lie down, the agent will switch to Live, mirroring the Visitor's movements while lying down, as well as change the camera, also similar to the previous installation, to a perspective camera located within the agents hand. If the Visitor decides to get up again, the camera returns to

normal again. Furthermore, if the Visitor lies down first, the agentfigure would follow suit, allowing it to recover energy even if it was not tired yet.
The island-based agent follows three different kinds of goals. The first one is a constant goal – it causes the agent to always seek movement. If the agent is mirroring the Visitor, and they move their bodies enough by themselves, the goal is satisfied and the agent does not have to move away from the Live state. The second kind of goal is triggered by Visitor actions, such as intense movement after the agentfigure stopped mirroring the Visitor causing the agent to re-synchronize with the Visitor. The last kind of goal is triggered by agent-internal conditions, such as the wish to rest as soon energy has reached a certain threshold.
Implementation of the island-system was achieved using Jason's annotation system, to give various plans differing chances of being executed, if multiple ones were possible to be selected. Annotations were read by the custom selector function in Java, which utilized random values to pick out of the list of weighted available options. The code below shows an example for two plans with the same goal and the same context, but with different bodies and annotations, which contain the chance of them being picked by the selector function. Accordingly, the first plan has a chance of 30, which is six times the chance of the second plan being chosen.

```
@idling[chance(30)]
+!moveAround : movement(M) & M <= 0.5
    <-  .wait(200);
        !moveAround.

@move[chance(5)]
+!moveAround : movement(M) & M <= 0.5
    <-  .random(R);
        ?bound_left(BL);
        ?bound_right(BR);
        moveTo(BL + (BR-BL)*R, 0);
        !waitForIdle(1000,250);
        !moveAround.
```

Below, a code snippet for the customized selector function can be seen. It was implemented within a class called RandomOption, which extends the Jason standard Agent class and had to be referenced within the Jason agent code of the supervisor agent whenever a new agent using this selector function was to be created. An example for this kind of agent creation can be seen further below.

```
/**
* Custom selector function chooses one of the options available at
   random.
* Reads annotations for weighted randomness, increases chance of
   being drawn by same factor.
```

```
*/
public class RandomOption extends Agent {
  public Option selectOption(List<Option> options) {
    List<Option> chanced = new ArrayList<>(options);
    for(Option o : options) {
      int chance = 1;
      try {
        Literal l = o.getPlan().getLabel().getAnnot("chance");
        chance = Integer.parseInt(l.getTerm(0).toString());
      }
      catch(Exception e) {}
      for(int i = 0; i < chance; i++)
      chanced.add(o);
    }
    double r = Math.random() * chanced.size();
    return chanced.get((int)r);
  }
}
```

Below, the example for creating an agent within Jason, using the `RandomOption` class explained above.

```
.create_agent(sleeper_agent1, "170329_sleeper6.asl",
 [agentClass("RandomOption")]);
```

It is important to note again, that while this approach to agent development is arguably closer to the original way Jason was intended to be used, it still does not completely align with the principles of Jason and agent-oriented programming. Compared to the previous state-machine-based approach, this way includes both the pro-active pursue of goals, as well as reactiveness to a changing environment, especially in the form of Visitor behavior. However, inter-agent communication and cooperation once again fell short, as the developed scenario focused more on the interaction between one agent and one Visitor, instead of incorporating multiple agents.
The result of this installment is an agent with very unpredictable and oftentimes hard-to-understand behavior. Visitors are oftentimes baffled and confused by what the agents are doing at any given time – as are the members of the project team. Although the agent program does not offer an excessively wide amount of options, the agents display a considerable amount of emergent behavior, which was not premeditated by the developer beforehand. Some ideas for improvements include toning down certain screen effects, like zooming, as well as decreasing the frequency of random actions undertaken by the agentfigure. Additionally, if more options were provided for the agent to choose from what to do at any given time, the scope of emergent behavior could be further broadened. This incarnation of the sleeping agent scenario contains several other, somewhat hidden, features, such as for example a credit roll being displayed on the screen, if a Visitor stands still and displays no intense movements for at least three minutes. Because Visitors

generally seem to enjoy the Live part of the interaction with the agentfigure the most, a way for Visitors to stop agents from acting on their own and go back to Live, mirroring the Visitor, was introduced. As mentioned before, whenever the Visitor shows a high enough activity level, the agent would return to them. The simple agent code for this reaction can be seen below.

```
@react_fast_go_live
+activity_level(AL) & AL > 2 & auto & not sleeping
    <-  -auto;
        switchToLive.
```

Some Visitors realized this very quickly and started calling the agentfigure back by waving their arms around energetically, whenever the agentfigure started autonomous actions. Arguably, this can be considered a very unique kind of interaction between the human and the non-human in form of the agentfigure, which was to be explored according to the original project statement.

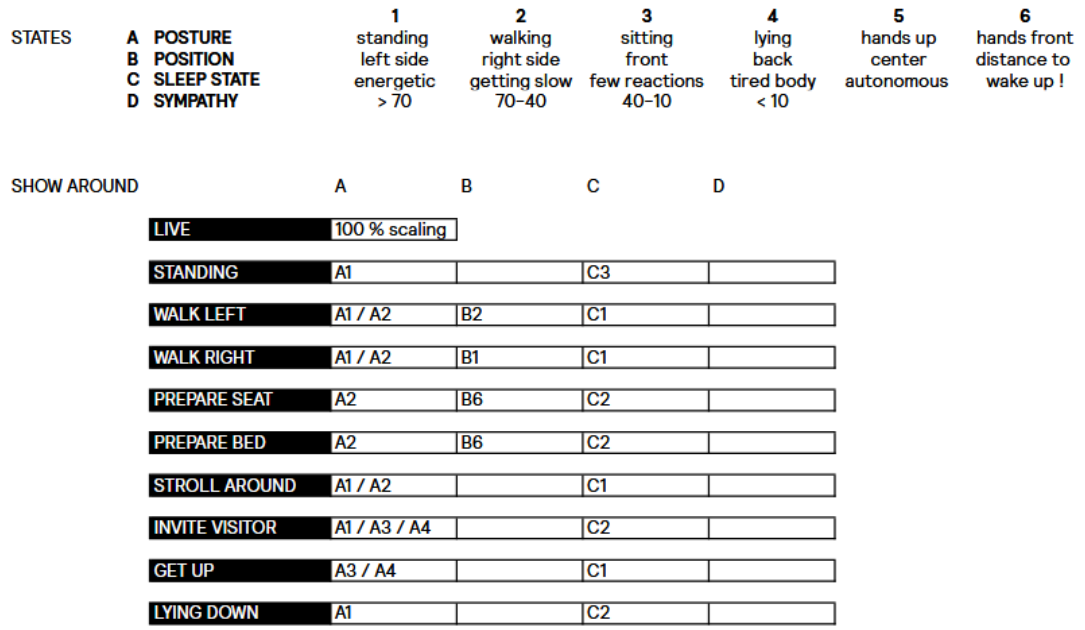

Figure 4.4: Sleeping agent island diagram, by Simon Oberhammer.

4.3 Visualization and Modelling

One of the major problems the project team was facing during development of the agent system was communicating and visualizing agent behavior in a way for every project team member to understand and agree on. The project owner had specific visions of how

agents were supposed to act and the development team had to understand and implement those visions. As such, communication of these ideas and visions was of vital importance to the project's success.
At the start, agent stories and descriptions were written down as text, as can be seen in Figure 4.1. Such descriptions included some concrete technical data which could be directly used to adjust parameters within the rendering engine. However, the written descriptions of agent behavior were sometimes rather ambiguous and thus hard to understand or translate into agent program code. What was described was the way the agent's behavior should appear to the Visitor, but not what caused it to act in the way it should. In hindsight, it would have been possible to develop agents from this starting point as well, but it would leave room for a lot of interpretation and creativity by the developers.
Instead, it was decided to hold small project-staff meetings to discuss and agree on possible agent behavior together. Drawing on paper to visualize thoughts and processes to other project members proved to be very successful, as it served both as a means of supporting verbal communication, as well as a form of documentation, which could be used for the process of implementation afterwards. A makeshift modeling concept based on the activity diagram of UML was employed, as it was able to tell the general gist of the story within a single look, while also encompassing all the fine det ls if necessary. Such drawn maps would sometimes be digitalized as seen in Figure 4.2.
These makeshift visualizations of agent behavior were much easier to convert into agent program code, as they provided insight into possible ways how the interaction between the Visitors and the virtual agentfigures was envisioned to be like. Similar to user stories in traditional development, the diagram allowed development to focus on a certain chain of events, but at the same time made it clear, if alternative paths were supposed to exist. As development went on and various agent stories were discarded in favor of new ones, the diagrams got more complex as well – an example can be seen in Figure 4.3. At the same time, the visualized agent behavior became less abstract and more akin to a finite-state machine. As was described in Subsection 4.1.4, converting the state machine into agent program code was rather straightforward, as every state would be implemented as a single plan attached to the goal of finishing that state. As the resulting code was no longer following some of the principles of agent programming, Jason's intended usage and best practice, it was decided to start the development of the agent again from scratch. The new iteration was supposed to move away from the finite-state machine and instead employ a model closer to the original vision of an agent system, while also enabling the desired indeterministic behavior of agents.
After the paradigm shift towards the island-system, agent behavior was at first visualized in a similar fashion to the state machines again. The new diagrams showed unconnected states on a blank background, similar to isolated islands floating in an ocean – hence the name of the system. As the agent story developed, such a visualization got confusing to read very quickly, as the addition of pre-conditions, in some cases even post-conditions, made it hard to comprehend the intended agent behavior as a whole. The graphs were thus exchanged for separate matrices for both pre-conditions and agent behavior within

certain contexts, a digitalized form of which can be seen in Figure 4.4. This way of presenting agent stories was also possible to convert into agent program code without too much effort. By using logical rules based on the aforementioned matrices as pre-conditions within plan contexts (context referring to the condition for a plan being eligible at a certain point in time, as explained in Subsection 3.4.5), readability was improved and plan heads could be adopted directly from the matrix-based description of agent behavior.
To summarize – it was a challenge to find a fitting solution to the communication problem that worked well for all parties involved. Text-based descriptions of agent stories were preferred by non-developers, but proved hard to implement. On the other hand, representations closer to actual agent program logic were hard to comprehend but easy to convert into code. As was mentioned at the start of the chapter, it might have been possible to work with text descriptions as well, especially if they were to focus on the experience from the standpoint of a Visitor, akin to user stories. Multiple stories, representing various possible interactions with the agentfigure, might have helped to develop the agent system in a more traditional way. Furthermore, it is important to note that the chosen way of visualization has an actual impact on the development and way of thinking of agent behavior, as can be seen in the example of agent program code devolving into a finite-state machine over the course of this project.

CHAPTER 5

Discussion and Conclusion

This chapter offers a short summary, discussion and reflection in regards to the development of an interactive agent system in the context of the INTRA SPACE project, as well as some concluding remarks. However first off, methods of testing and evaluating the agent system, as well as problems and solutions in general are presented and discussed. In this context, *testing* judges whether or not the agents perform what is expected of them, whereas *evaluation* focuses on the quality achieved in regards to the original research project goal – the exploration of interaction between humans and virtual agents.

5.1 Testing and Evaluation

In general, both testing and evaluating the developed agent system were rather limited in scope and could not be performed in a way necessary for a project of this kind. One of the main reasons for this was the fact that the complete installation needed to be running for practical testing to be conducted, which was not always the case. As this made active testing and debugging difficult – especially at the early stages of development – a simple, custom-made Java application was used to simulate UDP responses from the rendering engine, to enable quickly available ways of both testing and debugging the Jason agent programs, as well as the Jason environment and network communication Java classes. In the later stages of the project, the installation was in use more frequently and the number of Visitors increased, thus enabling pragmatic testing of the agent system more regularly.
Another problem was the lack of error messages within the Jason platform. Whenever agents stopped working, either by crashing, by not responding anymore, or by displaying erroneous behavior, the underlying reason was not easy to find or comprehend. This problem was solved for the most part by utilizing extensive logging via the Jason agent platform, which showed all agent logs in real-time within a separate window.
Evaluation of the system was also problematic, as the subjects testing the system and

experiencing the interaction with the agentfigure were project team members for the most part. Being aware of both agent behavior and goals made it possible to test for errors, but made it more difficult to evaluate the form of interaction between the Visitors and the virtual agentfigures. Proper evaluation was thus limited to cases in which outside Visitors interacted with the agent system. Because of the finite number of Visitors and the ambiguity of feedback, the task of evaluating the system regularly turned out to be quite challenging.
Results of the system's evaluation at the end of the project, based upon the feedback and comments of Visitors, imply a high variance in the way the installation was experienced. Many Visitors claimed to have realized the agentfigure as a social presence and were able to have meaningful interactions, whereas other Visitors saw the interaction rather as a form of entertainment, akin to a game or narrative experience. Consequently, it could be said that the project goal of exploring, formulating, and deconstructing the ways of interaction between humans and non-humans in the form of virtual agents were met, albeit in ways which were probably not originally envisioned.

5.2 Problems, Challenges, and Solutions

To summarize, the general challenges discussed in this work revolve around the following points:

- Developing and embedding an agent system within a dynamic environment
- Finding an agent story and behavior that works well for the given goal of exploring interaction
- Proper Implementation of IAs, following the principles of general agent programming, BDI, AS, and Jason
- Communication within the project team

The first point, equaling the research question of this work, is answered by the processes described over the course of Chapter 4. Development consisted of an iterative process, which was seemingly reset whenever an agent story was scrapped in favor of a new one. However, experience from previous installments would carry over, thus enabling the creation of more sophisticated agent scenarios with every iteration. The iterative process and regular discussions within the project team, as well as the inclusion of experts in relevant fields during such meetings, enabled a fruitful way of developing the required functionality within a dynamic requirement. It is important to mention that other agent system languages and frameworks might have suited the task better, if the final requirements were apparent at the start. However, such a massive change later on in the project was no longer possible due to resource constraints.
In regards to fitting agent stories and behavior, a fair number of points can be made, as to what constitutes as enabling the desired forms of interaction between agents and Visitors.

First and foremost, during the final stages of the project it became apparent that the instantaneous feedback during the Live phase seemed to engage Visitors the most, as it was clear what was happening as long as the agent mirrored their movements. Whenever the agent took control over the figure, feedback was often too slow to be understood as such. It was thus considered to be important to incorporate quick reactions to Visitor interaction within the agent stories. If the agent only acts in a predetermined way and does not really interact with Visitor, its behavior becomes very obvious and predictable to repeated Visitors and onlookers, thus minimizing interest, engagement, and interaction. To remedy this problem, indeterminism and the almost always existing possibility of instant interaction were introduced into the agents' behavior.
The problem of proper implementation ties into the other problems discussed in this work, as following the principles and guidelines of the BDI architecture, AS, and Jason more properly would have certainly produced different results than the ones achieved over the course of this project. On the other hand, the purposeful breaking of some rules enabled the implementation of the island-system, as described in Section 4.2. However, other core concepts, such as inter-agent cooperation, were still not followed appropriately and might have enhanced the interaction between Visitors and virtual agentfigures even further. The way custom action feedback was handled, as described in Section 3.6, can also be seen as a violation of Jason's best practice, and finding a way to properly receive feedback while still being able to produce the necessary results would have helped in producing more reliable agent programs. Different approaches, using either different agent platforms or other implementations of AI, such as neural networks or extensive scripts, might have also been worth pursuing and would have offered differing solutions to the ones described in this work.
The last of the points listed above – communication within the project team – played a significant role and had an impact, either directly or indirectly, on all areas of development. It was important to communicate both the imagined functionality and behavior of the agents, but also what was possible within the scope of the project in regards to cost requirements of various implementations. In order to maximize mutual understanding of what was expected and what was possible, graphs and visualization were utilized – both analog and digital. Communication of agent stories, plans, and behavior worked well using graphs, especially when communicating face to face while creating them. The resulting diagrams furthermore enabled a faster development and testing process. However, they also impacted the way agent behavior was thought about and implemented, as can be seen by the negative example of finite-state-machine-based agents described in Subsection 4.1.4.
Further problems and challenges which appeared during the course of this project, which are not directly related to the process of agent development, include the donning stage explained in Section 3.3, which went through several iterations and experiments together with the agent system, before arriving at the solution of both audio-based and visual instructions at the same time. Although the installation was planned to be a completely autonomous one at the start, this goal was never reached, as it was considered crucial to explain the donning process to new Visitors in addition to constant maintenance of

the installation. Difficulties in regards to motion capture and gesture recognition also impacted agent development, as they limited the scope of gestures and interaction the agents were able to perceive. A solution to parts of this problem was to move away from using the FUBI framework and implement custom scripts from scratch with similar functionality within the rendering engine.

5.3 Conclusion

In conclusion, it can be said that the developed agent system did not reach the level which was expected of it during the initial vision of the project in many aspects. However, it provided the required functionality of agents to be communicated with and was successfully embedded into the already existing installation. Some Visitors of the installation expressed surprise, intrigue, confusion, and many other unique experiences after interacting with the agents. Some were fascinated by the agents mirroring their actions, others wanted to touch and interact with the agents more deeply, while some even enjoyed watching the agents act on their own. Many Visitors were furthermore also eager to suggest various applicable fields, in which a similar installation, requiring no special equipment on behalf of the Visitor, could be used, such as entertainment, education, or therapy.

Appendix

Source code for some of the agents described in this work can be found in this appendix. Note that some parts of the source code have been omitted – for example if they were a common base among all agents, like plans used to wait for certain states.

Waving Agent 1

wave_agent1.asl

```
// === INITIAL BELIEFS ===

attitude(normal).  // describes the current mental attitude of the agent, eg normal,
sad, happy, etc.

// === RULES ===

// returns true, if the distance (=difference) between A and B is less than X
distance_under(A,B,X) :- (A-B) < X & (A-B) > (-X).

// === INITIAL GOALS ===

// === PLANS ===

+phase(X) : true <- !phase(X). -phase(X) : true <- .drop_desire(phase(X)).

// plan for phase 1:  go live for ~60 seconds, then wait for agent B to do the same.
// Then wave 3 times at agent B. (waiting ~20 secs each time).
// Afterwards, proceed to phase 2.
+!phase(1) :  true <- !liveFor(50, 20); !waitForAgentB; !exchangeInformation;
?other_agent(OTHER_AGENT); !waveTo(OTHER_AGENT); !waitForIdle; !liveFor(50, 10);
!waveTo(OTHER_AGENT); !waitForIdle; !liveFor(50, 10); !waveTo(OTHER_AGENT);
!waitForIdle; switchToLive; !waitForLive; -agentB_ready[_]; +phase(2); -phase(1).

// plan for phase 2:  go idle for a second, then wave to the camera.
// then wait for a reaction from the actor (see !waitForSingleWave)
+!phase(2) :  true <- .print("Entered phase 2."); .send(wave_agent2, achieve,
initphase(2)); switchToIdle; !waitForIdle; -+wavesToCamera(0); !waveToCamera(1);
!waitForIdle; !waitForSingleWave.

// wave into the camera (position assumed to be at 0,0,-3) using the given animation id
+!waveToCamera(ID) : true <- ?wavesToCamera(N); -+wavesToCamera(N+1); ?my_agent(AGENT);
?agentposition(AGENT, X, Y, Z); if(camerapos(CX,CZ)) { moveTo(X,Z,CX,CZ); } else {
moveTo(X,Z,0,-3); } !waitForIdle(500,500); playanimation(wave,ID).

// if an actor waves back, clap, let both agents wave into the camera
// and wait for both actors to wave back (!waitForDoubleWave)
+!waitForSingleWave :  actorWaved(_, _) <- playanimation(clap,1); .send(wave_agent2,
achieve, clap); !waitForIdle(500,1000); .send(wave_agent2, achieve, waveToCamera(1));
-+wavesToCamera(0); !waveToCamera(1); !waitForIdle; !waitForDoubleWave.
```

```
// if the waiting time reaches 5 seconds, talk to b, teach to wave and then try again
+!waitForSingleWave :  timeout_sw(T) & T > 50 & wavesToCamera(N) & N <=3 <- !talkToB;
!waitForIdle(500,1000); !teachToWave; !waitForIdle(500,1000); !waveToCamera(1);
!waitForIdle; -timeout_sw(_); !waitForSingleWave.

// waiting process for a reaction from an actor
+!waitForSingleWave :  timeout_sw(T) & wavesToCamera(N) & N <=3 <- -+timeout_sw(T+1);
.wait(100); !waitForSingleWave.

+!waitForSingleWave :  wavesToCamera(N) & N <= 3 & not timeout_sw(_) & wavesToCamera(N)
& N <=3 <- +timeout_sw(1); .wait(100); !waitForSingleWave.

// if the agent gets no reactions after the 4th wave, he gets sad and stops trying
+!waitForSingleWave :  wavesToCamera(N) & N > 3 <- .print("I'm sad now.  No one waves
back to me..."); -+attitude(sad); switchToIdle; !waitForIdle; fade; -phase(2).

// if both actors wave back -> success (phase 2 complete)
+!waitForDoubleWave :  my_actor(ACTOR) & other_actor(ACTORB) & actorWaved(ACTOR, _) &
actorWaved(ACTORB, _) <- .print("finished phase 2"); !celebrate; -phase(2).

// if the waiting time reaches 5 seconds (50 * 100 ms), wave angrier
+!waitForDoubleWave :  timeout_dw(T) & T > 50 & wavesToCamera(N) & N <=3 <-
.send(wave_agent2, achieve, waveToCamera(2)); !waveToCamera(2); !waitForIdle;
-timeout_dw(_); !waitForDoubleWave.

// waiting process for a reaction from both actors
+!waitForDoubleWave :  timeout_dw(T) & wavesToCamera(N) & N <=3 <- -+timeout_dw(T+1);
.wait(100); !waitForDoubleWave.

+!waitForDoubleWave :  wavesToCamera(N) & N <= 3 & not timeout_dw(_) & wavesToCamera(N)
& N <=3 <- +timeout_dw(1); .wait(100); !waitForDoubleWave.

// if the agent gets no reactions after the 4th wave, he gets sad and stops trying
+!waitForDoubleWave :  wavesToCamera(N) & N > 3 <- .print("I'm sad now.  They dont wave
back at us..."); -+attitude(sad); .send(wave_agent2, achieve, sad); fade; -phase(2).

+!talkToB : true <- .send(wave_agent2, achieve, talkToA); ?my_agent(AGENT);
?other_agent(AGENTB); ?agentposition(AGENT, X, Y, Z); ?agentposition(AGENTB, XB, YB,
ZB); moveTo(X,Z, XB,ZB); !waitForIdle; playanimation(talk, 1).

+!teachToWave :  true <- .send(wave_agent2, achieve, lookAtA); ?my_agent(AGENT);
?other_agent(AGENTB); ?agentposition(AGENT, X, Y, Z); ?agentposition(AGENTB, XB, YB,
ZB); moveTo(X,Z, XB,ZB); !waitForIdle; playanimation(teachToWave, 1); .wait(2000);
.send(wave_agent2, achieve, teachToWave).

// exchanges information about actors and agents with the other wave_agent
+!exchangeInformation :  my_actor(ACTOR) & my_agent(AGENT) <- .send(wave_agent2, tell,
other_actor(ACTOR)); .send(wave_agent2, tell, other_agent(AGENT)); .send(wave_agent2,
askOne, my_agent(_), AgentB); +agentb(AgentB); ?agentb(my_agent(OTHER_AGENT));
+other_agent(OTHER_AGENT); -agentb(_); .send(wave_agent2, askOne, my_actor(_), ActorB);
+actorb(ActorB); ?actorb(my_actor(OTHER_ACTOR)); +other_actor(OTHER_ACTOR); -actorb(_).

// waits until belief agentB_ready is achieved (when B has been live for ~60 sec)
+!waitForAgentB : not agentB_ready <- .wait(250); !waitForAgentB.

+!waitForAgentB : agentB_ready .

// tries to perform a waving animation towards the other agent.
// if the agents are too close together and 3 seconds pass without them separating,
// the agent decides to move away from the other agent before the waving animation
+!waveTo(TARGET) : other_agent(TARGET) & my_agent(AGENT) & agentposition(AGENT, XA, YA,
ZA) & agentposition(TARGET, XB, YB, ZB) & not distance_under(XA,XB,1.1) <-
-wave_try(_); moveTo(XA, ZA, XB, ZB); !waitForIdle(2000,1000); waveTo(TARGET, 1).

+!waveTo(TARGET) : other_agent(TARGET) & wave_try(N) & N >= 3 <- -+wave_try(N+1);
?agentposition(TARGET, XB, YB, ZB); .print("Im moving away..."); ?my_agent(AGENT);
```

```
?agentposition(AGENT, XA, YA, ZA); if(XA == XB) { if(cameraborder(CB1, CB2) & XA+2 >=
CB2-0.5) { moveTo(XA-2, ZA); } else { moveTo(XA+2, ZA); } } else { if(cameraborder(CB1,
CB2) & (XA+(XA-XB) >= CB2-0.5 | XA+(XA-XB) <= CB1+0.5)) { moveTo(XA-(XA-XB), ZA); }
else { moveTo(XA+(XA-XB), ZA); } } !waitForIdle(2000,1000); !waveTo(TARGET).

+!waveTo(TARGET) : other_agent(TARGET) & wave_try(N) & N < 3 <- -+wave_try(N+1);
.wait(1000); !waveTo(TARGET).

+!waveTo(TARGET) : other_agent(TARGET) <- .print("I'm too close to wave at ", TARGET);
+wave_try(1); .wait(1000); !waveTo(TARGET).

// if not in a phase, and actorscaling is finished, enter phase 1
+actorscaling(ACTOR,VALUE) : my_actor(ACTOR) & VALUE == 1 & not phase(_) <- +phase(1).

// if the other actor starts waving while in phase 1, celebrate and skip to phase 2.
+actorstate(ACTOR, STATE) : other_actor(ACTOR) & STATE == wave & agentB_ready &
phase(1) <- !celebrate; -phase(1); +phase(2); .drop_desire(phase(1)).

// logs waves received from actors for 3 seconds, phase 2 only
+actorstate(ACTOR, STATE) : STATE == wave & phase(2) <-
.drop_desire(countDownActorWaved(ACTOR)); +actorWaved(ACTOR);
!!countDownActorWaved(ACTOR).

+!countDownActorWaved(ACTOR) : true <- .wait(3000); -actorWaved(ACTOR).

// change attitude to happy, play celebration animation and tell B to do the same
+!celebrate :  true <- .print("yaay"); -+attitude(happy); .send(wave_agent2, achieve,
celebrate); playanimation(cheer,1); !waitForIdle; switchToLive; !waitForLive.
```

Waving Agent 2

wave_agent2.asl

```
// === INITIAL BELIEFS ===

wavesFromA(0).  // counts the number of perceived waves from agent A
attitude(normal).  // describes the current mental attitude of the agent, eg normal,
sad, happy, etc.

// === RULES === :- (A-B) < X & (A-B) > (-X).

// === INITIAL GOALS ===

// === PLANS ===

+phase(X) : true <- !phase(X). -phase(X) : true <- .drop_desire(phase(X)).

// plan for phase 1:  go live for ~60 sec, tell A that I'm ready.  rest is reactive.
+!phase(1) :  true <- -+wavesFromA(0); !liveFor(50,20); .send(wave_agent1, tell,
agentB_ready).

// plan for phase 2:  (completely controlled by agent 1, agent 2 only has to react)
+!phase(2) :  true <- .print("Entered phase 2."); .drop_desire(react);
.drop_desire(wave_back).

// if not in a phase, and actorscaling is finished, enter phase 1
+actorscaling(ACTOR,VALUE) : my_actor(ACTOR) & VALUE == 1 & not phase(_) <- +phase(1).

// whenever agent A waves at me (agent B), increase the counter (phase1 only)
+wave(A,B) : my_agent(B) & other_agent(A) & phase(1) <- ?wavesFromA(COUNT);
-+wavesFromA(COUNT + 1).

// if the counter is less then 2, do nothing
+wavesFromA(COUNT) : COUNT < 2 <- .print("Agent A waved at me ", COUNT, " times.").

// if the counter equals 2, look at the other agent
```

```
+wavesFromA(COUNT) : COUNT == 2 <- .print("Agent A waved at me ", COUNT, " times.
(looking)"); !react.  +!react :  phase(1) <- ?other_agent(AGENTA);
?agentposition(AGENTA, XA, YA, ZA); ?my_agent(AGENTB); ?agentposition(AGENTB, XB, YB,
ZB); lookAt(XA, YA * 2, ZA, true); moveTo(XB, ZB, XA, ZA); !waitForIdle(500,1000);
playanimation(headscratch,1); !waitForIdle(500,500); switchToLive.

// if the counter is 3 or more, wave back at agent A
+wavesFromA(COUNT) : COUNT >= 3 <- .print("Agent A waved at me ", COUNT, " times.
(waving back)"); !waveBack.

+!waveBack :  phase(1) <- lookAt(0,0,0,false); ?other_agent(AGENTA);
?agentposition(AGENTA, X, Y, Z); !waveTo(AGENTA); !waitForIdle(500,500); switchToLive.

// tries to perform a waving animation towards the other agent.
// if the agents are too close together on the screen and 3 seconds pass,
// agent decides to move away from the other agent and wave afterwards.
+!waveTo(TARGET) : other_agent(TARGET) & my_agent(AGENT) & agentposition(AGENT, XA, YA,
ZA) & agentposition(TARGET, XB, YB, ZB) & not distance_under(XA,XB,1.1) <-
-wave_try(_); moveTo(XA, ZA, XB, ZB); !waitForIdle(2000,1000); waveTo(TARGET, 1).

+!waveTo(TARGET) : other_agent(TARGET) & wave_try(N) & N >= 3 <- -+wave_try(N+1);
?agentposition(TARGET, XB, YB, ZB); .print("Im moving away..."); ?my_agent(AGENT);
?agentposition(AGENT, XA, YA, ZA); if(XA == XB) { moveTo(XA+1, ZA); } else {
moveTo(XA+(XA-XB), ZA); } !waitForIdle(2000,1000); !waveTo(TARGET).

+!waveTo(TARGET) : other_agent(TARGET) & wave_try(N) & N < 3 <- -+wave_try(N+1);
.wait(1000); !waveTo(TARGET).

+!waveTo(TARGET) : other_agent(TARGET) <- .print("I'm too close to wave at ", TARGET);
+wave_try(1); .wait(1000); !waveTo(TARGET).

+!initphase(2) :  not phase(2) <- +phase(2); -phase(1).

+!initphase(2):  phase(2) <- .print("Agent A wants me to init phase 2, but I am already
in phase 2 ???").

+!celebrate :  true <- .print("yaay"); -+attitude(happy); playanimation(cheer,1);
!waitForIdle; switchToLive; !waitForLive.

+!clap :  true <- playanimation(clap,1); !waitForIdle; switchToLive.

+!sad :  true <- .print("I'm sad now.  They dont wave back at us..."); -+attitude(sad);
fade; -phase(2).

+!waveToCamera(ID) : true <- ?my_agent(AGENT); ?agentposition(AGENT, X, Y, Z);
if(camerapos(CX,CZ)) { moveTo(X,Z,CX,CZ); } else { moveTo(X,Z,0,-3); } !waitForIdle;
playanimation(wave,ID); !waitForIdle; switchToLive.

+!lookAtA : true <- ?my_agent(AGENTB); ?other_agent(AGENTA); ?agentposition(AGENTA, XA,
YA, ZA); ?agentposition(AGENTB, XB, YB, ZB); moveTo(XB,ZB, XA,ZA).

+!talkToA : true <- ?my_agent(AGENTB); ?other_agent(AGENTA); ?agentposition(AGENTA, XA,
YA, ZA); ?agentposition(AGENTB, XB, YB, ZB); moveTo(XB,ZB, XA,ZA);
!waitForIdle(1500,500); playanimation(talk, 1).

+!teachToWave :  true <- ?my_agent(AGENTB); ?other_agent(AGENTA);
?agentposition(AGENTA, XA, YA, ZA); ?agentposition(AGENTB, XB, YB, ZB); moveTo(XB,ZB,
XA,ZA); !waitForIdle(1500,500); playanimation(teachToWave, 1).
```

Bat Agent

bat_agent_v3.asl

```
// === INITIAL BELIEFS ===

phase(1).  animating.
```

```
// === INITIAL GOALS ===

// === PLANS ===

+phase(X) : true <- !phase(X). -phase(X) : true <- .drop_desire(phase(X)).

+actorscaling(ACTOR, VALUE) : my_actor(ACTOR) & phase(1) & VALUE > 0 & VALUE < 1 &
initiated <- -phase(1); +phase(2).

+actorscaling(ACTOR, VALUE) : my_actor(ACTOR) & phase(2) & VALUE == 1 <- -phase(2);
+phase(3).

+!phase(1) :  true <- .print("init phase 1 (waiting for visitor)");
.create_agent(shadow_agent10, "shadowAgent.asl"); .send(shadow_agent10, achieve,
phase(1)); if(camerapos(CX,CZ)) { moveTo(-1,0,CX,CZ); } else { moveTo(-1,0,0,-3); }
!waitForIdle(1000,100); +initiated; !!waitForVisitor.

+!phase(2) :  true <- .print("init phase 2 (donning)"); .send(shadow_agent10, achieve,
phase(2)).

+!phase(3) :  true <- .drop_desire(waitForVisitor); .print("init phase 3 (live)");
.send(shadow_agent10, achieve, die); switchToIdle; !waitForIdle(1000,250);
switchToLive; -animating.

// Print black holes on arms and feet, scaling with distance to screen
+actordistancetoscreen(ACTOR, DISTANCE) : my_actor(ACTOR) & phase(3) <-
printSprite(black1, 100, lefthand, 0.5, 0.5, DISTANCE/400).

// Handstogether causes the bat to come down to the ground.
// it will stay there for a maximum of 30s or until hands are put together again.
// after going to the ground, it will ignore the command to go down again for 30s
+actorstate(ACTOR, STATE) : my_actor(ACTOR) & not animating & phase(3) & STATE ==
handstogether & not onGround & not stayOnCeiling <- +animating; +onGround;
switchToIdle; !waitForIdle(250,0); switchToGround; !waitForIdle(2000,0);
switchToCamera(CameraLeftForeArm, 3); .wait(3000); switchToLive; -animating;
+stayOnCeiling; !!getBackToCeiling(30).

// handstogether while on ground causes bat to go back to ceiling
+actorstate(ACTOR, STATE) : my_actor(ACTOR) & not animating & phase(3) & STATE ==
handstogether & onGround <- +animating; -onGround; switchToIdle; !waitForIdle(250,0);
switchToCeiling; !waitForIdle(2000,0); switchToCamera(CameraFront, 3); .wait(3000);
switchToLive; -animating.

// causes the bat to go back to ceiling after TIME seconds
+!getBackToCeiling(TIME) : true <- .wait(TIME * 1000); if(not animating & onGround) {
+animating; -onGround; switchToIdle; !waitForIdle(250,0); switchToCeiling;
!waitForIdle(2000,500); switchToCamera(CameraFront, 3); .wait(3000); switchToLive;
-animating; } .

// after coming down, the bat will not react to commands to come down for 60 seconds
+stayOnCeiling :  true <- .wait(60000); -stayOnCeiling.

// plays waiting animations ("sleeping cocoon") in pseudo random order
+!waitForVisitor :  true <- .random(RANDOM); if(RANDOM > 0.66) {
if(last_animation(NAME, TIMES) & NAME == hanging_cocoon1) { if(TIMES < 2) {
-+last_animation(hanging_cocoon1, TIMES+1); playanimation(hanging_cocoon, 1); } } else
{ -+last_animation(hanging_cocoon1, 1); playanimation(hanging_cocoon, 1); } } else {
if(RANDOM > 0.33) { if(last_animation(NAME, TIMES) & NAME == hanging_cocoon2) {
if(TIMES < 2) { -+last_animation(hanging_cocoon2, TIMES+1); playanimation(hanging, 2);
} } else { -+last_animation(hanging_cocoon2, 1); playanimation(hanging, 2); } } else {
if(last_animation(NAME, TIMES) & NAME == hanging_cocoon3) { if(TIMES < 2) {
-+last_animation(hanging_cocoon3, TIMES+1); playanimation(hanging, 3); } } else {
-+last_animation(hanging_cocoon3, 1); playanimation(hanging, 3); } } } .wait(11000);
!waitForVisitor.
```

Offset Agent (Used by Sleeper Agent)

170307_sleeperOffset.asl

```
+!die :  true <- .my_name(NAME); killOffset; .kill_agent(NAME).

+!run :  true <- .print("starting up!"); .random(RAND); .wait(1000 + RAND * 3000);
switchToSwapping; +swapping; playanimation(prepare_for_sleep, 1, 1, 1); +running.

+!walkOut :  running & swapping <- .drop_desire(walkAround); switchToSwapping;
-swapping; .random(RAND); if(RAND > 0.5) { moveTo(-20,0); } else { moveTo(20,0); } .

+!walkOut :  running & not swapping <- .drop_desire(walkAround); .random(RAND); if(RAND
> 0.5) { moveTo(-20,0); } else { moveTo(20,0); } .

+!walkOut :  not running <- .wait({+running}); !walkOut.

+!walkIn :  true <- .random(RAND); moveTo(-2 + RAND*4, 0).

+!walkAround :  not running <- .wait({+running}); !walkAround.

+!walkAround :  running <- .random(RAND); moveTo(-2 + RAND*4, 1); .random(RAND2);
.wait(3000 + RAND2 * 5000); !walkAround.
```

Sleeper Agent

170329_sleeper6.asl

```
t_action_interval(500).  // time between actions

t_welcome1(3000).  t_thankyou1(2000).  t_welcome2(18000).  t_welcome3(12000).
t_welcome4(7000).
t_random_min(4000).  // time in between spoken sentences (minimum)
t_random_max(12000).  // time in between spoken sentences (maximum)
speed_dontex(1.4).  // general duration multiplier for donning textures
t_dontex001(7000).  t_dontex002(6500).  t_dontex1(5000).  t_dontex2(3000).
t_dontex3(6500).  t_dontex4(6500).  t_dontex5(5000).  t_dontex6(9000).
t_dontex7(20000).  t_dontex8(7000).  t_dontex9(7000).  t_dontex10(7000).
t_dontex11(5000).  t_dontex12(1000000).

sympathy(50).  energy(150).  movement(0).  activity_level(0).
min_movement_walking(0.5).

bodymotion_multi(1).  // multiplier for bodymotion value added to activity level
movement_multi(1).  // multiplier for movement value added to activity level
activity_level_th1(10).  // activity level threshold for small energy reduction
activity_level_th2(30).  // activity level threshold for strong energy reduction
small_energy_cost(0.3).  strong_energy_cost(1.0).

bound_left(-2).  // bounds to the left side (world coordinates)
bound_right(2).  // bounds to the right side (world coordinates)
wait_time(250).  // waiting time between movement adjustments in ms
delta_movement(0.1).  // delta for movement value, deducted with every adjustment
t_min_live(2000).  t_before_emancipation(180000).
bm_min_start(4).  // bodymotion value to start emancipation
m_min_start(6).  // movement value to start emancipation
max_distance_to_objects(1.2).

t_deep_sleep(120000).  // duration of deep sleep phase
t_before_credits(180).  // time in seconds to stand still before credits roll
// === RULES ===

:- (A-B) < X & (A-B) > (-X).

// standing
```

```
a1 :- not a2 & not a3 & not a4.
// walking
a2 :- moving.
// sitting
a3 :- my_actor(ACTOR) & actorstate(ACTOR, waistlow) & not actorstate(ACTOR, waistdown).
// lying
a4 :- my_actor(ACTOR) & actorstate(ACTOR, waistdown).
// hands up
a5 :- my_actor(ACTOR) & actorstate(ACTOR, handsup).
// hands front
a6 :- my_actor(ACTOR) & actorstate(ACTOR, handsfront).
// position - left side
b1 :- my_actor(ACTOR) & actorposition(ACTOR, X, Y, Z) & X < -0.5.
// right side
b2 :- my_actor(ACTOR) & actorposition(ACTOR, X, Y, Z) & X > 0.5.
// front side
b3 :- my_actor(ACTOR) & actorposition(ACTOR, X, Y, Z) & Z > 0.5.
// back side
b4 :- my_actor(ACTOR) & actorposition(ACTOR, X, Y, Z) & Z < -0.5.
// center
b5 :- not b1 & not b2 & not b3 & not b4.
// close to bed
b6 :- objectpositon(bedlie, X, Y, _, _) & max_distance_to_objects(MD) & my_agent(AGENT)
& agentposition(AGENT, AX, _, AY) & distance_under(AX, Y, MD) & distance_under(AY, Y,
MD).
// close to seat
b7 :- objectpositon(bedsit, X, Y, _, _) & max_distance_to_objects(MD) & my_agent(AGENT)
& agentposition(AGENT, AX, _, AY) & distance_under(AX, Y, MD) & distance_under(AY, Y,
MD).

// sleep states
c1 :- energy(E) & E >= 90.  c2 :- energy(E) & E < 90 & E >= 50.  c3 :- energy(E) & E <
50 & E >= 20.  c4 :- energy(E) & E < 20.  c5 :- sleeping.  c6 :- wakingUp.

// sympathy
d1 :- sympathy(S) & S > 70.  d2 :- sympathy(S) & S <= 70 & S >40.  d3 :- sympathy(S) &
S <= 40 & S >10.  d4 :- sympathy(S) & S <=10.

// === GOALS ===

!start.

// === PLANS ===

+!report :  my_agent(AGENT) & sympathy(S) & energy(E) & movement(M) &
activity_level(AL) & bodymotion(AGENT, BM) <- .print("REPORT: Sympathy:  ", S, "
energy:  ", E, " movement:  ", M, " activity level:  ", AL, " bodymotion:  ", BM);
.wait(5000); !!report.

+!report :  my_agent(AGENT) & not(sympathy(S) & energy(E) & movement(M) &
activity_level(AL) & bodymotion(AGENT, BM)) <- .print("REPORT failed to gather data.
trying again."); .wait(5000); !!report.

+!adjust_values :  my_agent(AGENT) & movement(M) & activity_level(AL) &
bodymotion(AGENT, BM) & wait_time(WT) & bodymotion_multi(BMM) & movement_multi(MM) <-
-+activity_level(BM * BMM + M * MM); !adjust_energy; .wait(WT); !!adjust_values.

+!adjust_values :  my_agent(AGENT) & not (energy(E) & movement(M) & activity_level(AL)
& bodymotion(AGENT, BM)) & wait_time(WT) <- .print("failed to adjust values.  trying
again"); .wait(WT); !!adjust_values.

@very_active_live[atomic] +!adjust_energy :  activity_level(AL) & energy(E) &
activity_level_th1(ALT1) & activity_level_th2(ALT2) & small_energy_cost(SMEC) &
strong_energy_cost(STEC) & AL >= ALT2 & my_agent(AGENT) & agentstate(AGENT, live) <-
-+energy(E-STEC).
```

```
@less_active_live[atomic] +!adjust_energy :  activity_level(AL) & energy(E) &
activity_level_th1(ALT1) & activity_level_th2(ALT2) & small_energy_cost(SMEC) &
strong_energy_cost(STEC) & AL < ALT2 & AL >= ALT1 & my_agent(AGENT) & agentstate(AGENT,
live) <- -+energy(E-SMEC).

@not_active_live +!adjust_energy :  activity_level(AL) & energy(E) &
activity_level_th1(ALT1) & activity_level_th2(ALT2) & small_energy_cost(SMEC) &
strong_energy_cost(STEC) & AL < ALT1 & my_agent(AGENT) & agentstate(AGENT, live) .

@very_active_notlive[atomic] +!adjust_energy :  my_agent(AGENT) & energy(E) & not
agentstate(AGENT, live) & bodymotion(AGENT, BM) & bodymotion_multi(BMM) &
activity_level_th1(ALT1) & activity_level_th2(ALT2) & strong_energy_cost(STEC) & BM *
BMM >= ALT2 <- -+energy(E-STEC).

@less_active_notlive[atomic] +!adjust_energy :  my_agent(AGENT) & energy(E) & not
agentstate(AGENT, live) & bodymotion(AGENT, BM) & bodymotion_multi(BMM) &
activity_level_th1(ALT1) & activity_level_th2(ALT2) & small_energy_cost(SMEC) & BM *
BMM < ALT2 & BM * BMM >= ALT1 <- -+energy(E-SMEC).

@not_active_notlive[atomic] +!adjust_energy :  my_agent(AGENT) & not agentstate(AGENT,
live) & bodymotion(AGENT, BM) & bodymotion_multi(BMM) & activity_level_th1(ALT1) & BM *
BMM < ALT1 .

@energy_not_negative[atomic] +energy(E) : E < 0 <- -+energy(0).

+energy(E) : E < 10 & E > 0 & objectpositon(bedlie, X, Y, _, _) <- .print("I am very
tired").

+energy(E) : E < 50 & E > 20 & not livedelay <- +livedelay; live_delay(3).

+energy(E) : (E > 50 | E < 20) & livedelay <- -livedelay; live_delay(0).

+!start :  true <- .wait(250); ?my_actor(ACTOR); +welcoming; if(my_agent(agent0)) {
!!playWelcomeAudio2; !!spawnOffset(1001, -1.5, 0); !!spawnOffset(1002, 1.5, 0);
switchToSwapping; +swapping; } .wait(my_actor(ACTOR) & actorscaling(ACTOR, VALUE) &
VALUE >= 1); if(swapping) { switchToSwapping; -swapping; } !!standbyOffsets;
-welcoming; !goLive; ?actorposition(ACTOR,X,Y,Z); +lastposition(X,Z); !!checkMovement;
!!adjust_values; !!report; +cfc(0); !!checkForCredits; ?t_before_emancipation(TBE);
.wait(TBE); !!killOffsets; +ec(0); !start_emancipation.

+!checkForCredits :  activity_level(AL) & AL > 0.3 <- -+cfc(0); .wait(1000);
!checkForCredits.

+!checkForCredits :  activity_level(AL) & AL <= 0.3 & cfc(CFC) & t_before_credits(TBC)
& CFC < TBC <- -+cfc(CFC+1); .wait(1000); !checkForCredits.

@go_credits[atomic] +!checkForCredits :  activity_level(AL) & AL <= 0.3 & cfc(CFC) &
t_before_credits(TBC) & CFC >= TBC <- -phase(_); .drop_desire(doSomething);
?objectposition(bedsit, X, Y, DX, DY); moveTo(X, Y, DX, DY); !waitForIdle(1000,0);
playanimation(sit_down_bed, 1); queueanimation(sit_on_bed, 1,1,1);
displayTexture(Credits, 60, 0,-1,0,1); .wait(60000); playanimation(sit_up_bed, 1);
!waitForIdle(1000,500); switchToLive; !!playWelcomeAudio2; !!spawnOffset(1001, -1.5,
0); !!spawnOffset(1002, 1.5, 0); if(not swapping) { switchToSwapping; +swapping; }
.wait(10000); .wait(activity_level(AL) & AL > 2); switchToSwapping; -swapping;
+phase(sleep).

+!start_emancipation :  my_agent(AGENT) & ((bodymotion(AGENT, BM) & bm_min_start(BMS) &
BM > BMS) | (movement(M) & m_min_start(MMS) & M > MMS)) & ec(EC) & EC < 10 <-
.print("actor too active.  delaying emancipation."); -+ec(EC+1); .wait(3000);
!start_emancipation.

+!start_emancipation :  my_agent(AGENT) & ((bodymotion(AGENT, BM) & bm_min_start(BMS) &
BM <= BMS & movement(M) & m_min_start(MMS) & M <= MMS) | (ec(EC) & EC >= 10)) <-
+phase(sleep).

@movement_update[atomic] +!checkMovement :  lastposition(LX,LZ) & my_actor(ACTOR) &
actorposition(ACTOR, X, Y, Z) <- ?movement(M); if(LX > X) { -+movement(M+(LX-X)); }
```

```
else { -+movement(M+(X-LX)); } ?movement(M2); ?delta_movement(D); if(LZ > Z) {
-+movement(M2+(LZ-Z) - D); } else { -+movement(M2+(Z-LZ) - D); } -+lastposition(X,Z);
?movement(M3); ?min_movement_walking(MM); if(M3-M > MM) { +moving; if(M3 - M > 3) {
-+movement(M+3); } } else { -moving; } !!reCheckMovement.

+!reCheckMovement :  true <- ?wait_time(W); .wait(W); !!checkMovement.

@movement_not_negative[dynamic] +movement(M) : M < 0 <- -+movement(0).

@movement_not_too_high[dynamic] +movement(M) : M > 50 <- -+movement(40).

@energy_not_too_high[dynamic] +energy(E) : E > 150 <- -+energy(150).

+phase(P) : true <- .print("entered phase :  ", P); !!doSomething.

-!doSomething :  true <- .print("SOMETHING TERRIBLE HAPEND"); .wait(500); !doSomething.

@standing +!doSomething :  a1 & c3 & my_agent(AGENT) & not agentstate(AGENT, idle) &
activity_level(AL) & AL < 1 <- switchToIdle; ?t_action_interval(TAI); .wait(TAI);
!doSomething.

@react_fast_go_live +activity_level(AL) & AL > 2 & auto & not sleeping <- -auto;
switchToLive.

@walk_left +!doSomething :  a1 & b2 & c1 & not sleeping & activity_level(AL) & AL < 0.5
<- +auto; .random(RAND); moveTo(-2+RAND, 0); !waitForIdle(2000,1000);
?t_action_interval(TAI); .wait(TAI); !doSomething.

@walk_right +!doSomething :  a1 & b1 & c1 & not sleeping & activity_level(AL) & AL <
0.5 <- +auto; .random(RAND); moveTo(2-RAND, 0); !waitForIdle(2000,1000);
?t_action_interval(TAI); .wait(TAI); !doSomething.

@seat +!doSomething :  (a1 | a2) & b7 & (c2|c3) & objectposition(bedsit, X, Y, DX, DY)
& not sleeping <- +sitting; .print("sitting"); moveTo(X, Y, DX, DY);
!waitForIdle(1000,0); playanimation(sit_down_bed, 1); queueanimation(sit_on_bed,
1,1,1); ?t_action_interval(TAI); .random(RAND); .wait(TAI * (RAND+1) * 10);
playanimation(sit_up_bed, 1); !waitForIdle(1000,500); -sitting; !doSomething.

@bed +!doSomething :  (a1 | a2) & b6 & (c2|c3|c4) & objectposition(bedlie, X, Y, DX,
DY) & not sleeping <- +sleeping; moveTo(X, Y, DX, DY); !waitForIdle(1000,0);
playanimation(lie_down_bed, 1); queueanimation(lie_on_bed, 1,1,1);
?t_action_interval(TAI); .random(RAND); .wait(TAI * (RAND+1) * 10);
playanimation(lie_up_bed, 1); !waitForIdle(1000,500); -sleeping; !doSomething.

@stroll_around +!doSomething :  a1 & (c1|c2) & not sleeping & activity_level(AL) & AL <
2 <- +auto; !moveAround; !doSomething.

@get_up +!doSomething :  (not a4 & not a3) & c1 & sleeping <- -sleeping;
playanimation(deep_get_up, 1); !waitForIdle(1000, 0); ?t_action_interval(TAI);
.wait(TAI); !doSomething.

@default_live[chance(3)] +!doSomething :  my_agent(AGENT) & agentstate(AGENT, idle) &
activity_level(AL) & AL > 2 & not sleeping & not a4 <- -auto; switchToLive;
?t_action_interval(TAI); ?t_min_live(TML); .wait(TAI+TML); !doSomething.

@default_live_lying +!doSomething :  my_agent(AGENT) & agentstate(AGENT, idle) & a4 &
objectposition(bedlie, X, Y, DX, DY) & b6 <- +sleeping; moveTo(X, Y, DX, DY);
!waitForIdle(1000,0); playanimation(lie_down_bed, 1); queueanimation(lie_on_bed,
1,1,1); !blendin; ?t_action_interval(TAI); .wait(TAI); !doSomething.

@default_live_lying_somewhere_else +!doSomething :  my_agent(AGENT) & agentstate(AGENT,
idle) & a4 & my_actor(ACTOR) & actorposition(ACTOR, X, Y, Z) & not b6 <- +sleeping;
moveTo(X, Z); !waitForIdle(1000,0); playanimation(lie_down, 1);
queueanimation(sleep_loop, 1,1,1); !blendin; ?t_action_interval(TAI); .wait(TAI);
!doSomething.

@default_live_sleeping +!doSomething :  my_agent(AGENT) & activity_level(AL) & AL > 3 &
sleeping & b6 <- -sleeping; -auto; +wakingUp; playanimation(lie_up_bed, 1);
```

```
!waitForIdle(1000,250); switchToLive; ?t_action_interval(TAI); .wait(TAI); -wakingUp;
!doSomething.

@do_nothing[chance(2)] +!doSomething :  true <- ?t_action_interval(TAI); .wait(TAI);
!doSomething.

+actorstate(ACTOR, waistdown) :  my_actor(ACTOR) & not sleeping <- +sleeping.

+actorstate(ACTOR, waistdown) :  my_actor(ACTOR) & sleeping <- !blendin.

-actorstate(ACTOR, waistdown) :  my_actor(ACTOR) & sleeping <- -sleeping; !blendout.

+!blendin :  not blendedin <- +blendedin; blend_in(upper_body, 3); blend_in(lower_body,
3).

+!blendin :  blendedin .

+!blendout :  not blendedin .

+!blendout :  blendedin <- -blendedin; blend_out(upper_body, 3); blend_out(lower_body,
3).

@walk_slow +!doSomething :  b4 & (c3 | c4) & not slow <- +slow; movespeed(0.5);
?t_action_interval(TAI); .wait(TAI); !doSomething.

@walk_normal +!doSomething :  (c1 | c2) & slow <- -slow; movespeed(1);
?t_action_interval(TAI); .wait(TAI); !doSomething.

@deep_sleep +!doSomething :  a4 & (c1 | c2) & activity_level(AL) & AL < 1 <- -auto;
.print("starting deep sleep!"); !blendout; +deepsleep; switchToCamera(CameraHand, 3);
switchToLive; ?t_deep_sleep(TDS); .wait(TDS); .wait(activity_level(AL2) & AL2 < 1);
switchToCamera(CameraFront, 3); switchToSwapping; .wait(5000); !doSomething.

@narcotic_sleep +!doSomething :  a4 & b4 & (c3 | c4) & not b6 <- !blendout;
environment(narcotic); playanimation(narcotic_sleep_loop, 1, 1, 1);
?t_action_interval(TAI); .wait(TAI); !doSomething.

@narcotic_sleep_bed +!doSomething :  a4 & b4 & (c3 | c4) & b6 <- !blendout;
environment(narcotic); playanimation(narcotic_sleep_loop, 2, 1, 1);
?t_action_interval(TAI); .wait(TAI); !doSomething.

@zoom_out +!doSomething :  a1 & b4 & not zoom(-3) <- -+zoom(-3); setTimedZoom(-3, 4);
?t_action_interval(TAI); .wait(TAI); !doSomething.

@zoom_in +!doSomething :  a1 & b3 & not zoom(0) <- -+zoom(0); setTimedZoom(0, 4);
?t_action_interval(TAI); .wait(TAI); !doSomething.

@zoom_in_less +!doSomething :  a1 & b5 & not zoom(-1.5) <- -+zoom(-1.5);
setTimedZoom(-1.5, 4); ?t_action_interval(TAI); .wait(TAI); !doSomething.

@unsettled_sleep +!doSomething :  a4 & (c1 | c2) & activity_level(AL) & AL >= 8 & not
b6 <- playanimation(unsettled_sleep_loop, 1, 1, 1); ?t_action_interval(TAI);
.wait(TAI); !doSomething.

@unsettled_sleep_bed +!doSomething :  a4 & (c1 | c2) & activity_level(AL) & AL >= 8 &
b6 <- playanimation(unsettled_sleep_loop, 2, 1, 1); ?t_action_interval(TAI);
.wait(TAI); !doSomething.

@fall_asleep +!doSomething :  energy(E) & E < 10 & objectposition(bedlie, X, Y, DX, DY)
<- +sleeping; moveTo(X, Y, DX, DY); !waitForIdle(1000,0); playanimation(lie_down_bed,
1); queueanimation(lie_on_bed, 1,1,1); ?t_action_interval(TAI); .wait(TAI);
!doSomething.

+!playWelcomeAudio2 :  my_actor(ACTOR) & actorscaling(ACTOR, AS) & AS > 0 <-
.print("skipping welcome audio").

+!playWelcomeAudio2 :  my_actor(ACTOR) & ((actorscaling(ACTOR, AS) & AS <= 0) | not
actorscaling(ACTOR, _)) <- playsound(explain1); .wait(60000 * 7); !playWelcomeAudio2.
```

```
+!playWelcomeAudio :  my_actor(ACTOR) & actorscaling(ACTOR, AS) & AS >=1 <-
.print("skipping welcome audio because scaling already finished").

+!playWelcomeAudio :  my_actor(ACTOR) & ((actorscaling(ACTOR, AS) & AS < 1) | not
actorscaling(ACTOR, _)) <- playsound(welcome1); ?t_welcome1(T); .wait(T);
playsound(welcome2); +lastsound(welcome2); ?t_welcome2(T2); .wait(T2); !waitRandomTime;
!welcomeAudioLoop.

+!welcomeAudioLoop :  welcoming <- .random(RAND); !playWelcome(RAND).

+!welcomeAudioLoop :  not welcoming.

+!playWelcome(R) : R > 0.6 & not lastsound(welcome4) <- playsound(welcome4);
-+lastsound(welcome4); ?t_welcome4(T); .wait(T); !waitRandomTime; !welcomeAudioLoop.

+!playWelcome(R) : R > 0.3 & not lastsound(welcome3) <- playsound(welcome3);
-+lastsound(welcome3); ?t_welcome3(T); .wait(T); !waitRandomTime; !welcomeAudioLoop.

+!playWelcome(R) : not lastsound(welcome2) <- playsound(welcome1);
-+lastsound(welcome1); ?t_welcome1(T1); .wait(T1); playsound(welcome2);
-+lastsound(welcome2); ?t_welcome2(T); .wait(T); !waitRandomTime; !welcomeAudioLoop.

+!playWelcome(R) : true <- .wait(100); !welcomeAudioLoop.

+startintrodonning :  welcoming <- -welcoming; playsound(null, 1); if(swapping) {
switchToSwapping; -swapping; } !!standbyOffsets; ?t_thankyou1(T); .wait(T); +donning;
!new_donning.

+!new_donning :  true <- moveTo(5,0); ?speed_dontex(SDT); ?t_dontex001(DT001);
?t_dontex002(DT002); ?t_dontex1(DT1); ?t_dontex2(DT2); ?t_dontex3(DT3);
?t_dontex4(DT4); ?t_dontex5(DT5); ?t_dontex6(DT6); ?t_dontex7(DT7); ?t_dontex8(DT8);
?t_dontex9(DT9); ?t_dontex10(DT10); ?t_dontex11(DT11); ?t_dontex12(DT12);
displayTexture(donning001, DT001*SDT/1000); .wait(DT001*SDT);
displayTexture(donning002, DT002*SDT/1000); .wait(DT002*SDT); displayTexture(donning01,
DT1*SDT/1000); .wait(DT1*SDT); displayTexture(donning02, DT2*SDT/1000); .wait(DT2*SDT);
displayTexture(donning03, DT3*SDT/1000); .wait(DT3*SDT); displayTexture(donning04,
DT4*SDT/1000); .wait(DT4*SDT); displayTexture(donning05, DT5*SDT/1000); .wait(DT5*SDT);
displayTexture(donning06, DT6*SDT/1000); .wait(DT6*SDT); displayTexture(donning07,
DT7*SDT/1000); ?camerapos(CX,CZ); moveTo(-1,0, CX, CZ); !waitForIdle(2000,500);
playanimation(donning_shadow, 1); .wait(DT7*SDT); !waitForIdle(2000,1000); moveTo(5,0);
displayTexture(donning08, DT8*SDT/1000); .wait(DT8*SDT); displayTexture(donning09,
DT9*SDT/1000); .wait(DT9*SDT); displayTexture(donning10, DT10*SDT/1000);
.wait(DT10*SDT); displayTexture(donning11, DT11*SDT/1000); .wait(DT11*SDT);
displayTexture(donning12, DT12*SDT/1000); .wait(my_actor(ACTOR) & actorscaling(ACTOR,
VALUE) & VALUE > 0); displayTexture(donning12, 0.1);
// start scaling here
.

+actorscaling(ACTOR, AS) : my_actor(ACTOR) & donning & AS >= 1 <- -donning;
playsound(thankyou3, 1).

+actorscaling(ACTOR, AS) : my_actor(ACTOR) & donning & AS >= 0.5 & not verygood <-
+verygood; playsound(verygood1, 1).

+!goLive :  my_actor(ACTOR) & actorscaling(ACTOR, AS) & AS < 1 <- .print("goLive
reached, but scaling not finished.  restarting."); !start.

+!goLive :  my_actor(ACTOR) & actorscaling(ACTOR, AS) & AS >= 1 <- switchToLive.

@stop_in_motion +!moveAround :  movement(M) & M > 0.5 .

@leerlauf[chance(30)] +!moveAround :  movement(M) & M <= 0.5 <- .wait(200);
!moveAround.

@move[chance(5)] +!moveAround :  movement(M) & M <= 0.5 <- .random(R); ?bound_left(BL);
?bound_right(BR); moveTo(BL + (BR-BL)*R, 0); !waitForIdle(1000,250); !moveAround.
```

```
@lookIntoCam[chance(3)] +!moveAround :  movement(M) & M <= 0.5 <- !lookIntoCamera;
.random(R2); !waitForIdle(1000, 500 + R2 * 1000); !moveAround.

@stopRandomly[chance(1)] +!moveAround :  movement(M) & M <= 0.5 .

+!waitRandomTime :  true <- ?t_random_min(TRMIN); ?t_random_max(TRMAX); .random(RAND);
.print("waiting:  ", TRMIN + (TRMAX-TRMIN)*RAND, " ms."); .wait(TRMIN +
(TRMAX-TRMIN)*RAND).

+!spawnOffset(NR, X, Y) : true <- ?my_agent(AGENT); spawnOffset(NR, X, Y, 1);
.concat("sleeperOffset", NR, AGNAME); +active_agent(AGNAME); .create_agent(AGNAME,
"170307_sleeperOffset.asl"); .send(AGNAME, achieve, run); .print("created agent:  ",
AGNAME).

+!killOffsets :  active_agent(AGNAME) <- -active_agent(AGNAME); !!killOffsets;
.wait(20000); .send(AGNAME, achieve, die).

+!killOffsets :  not active_agent(_).

+!standbyOffsets :  active_agent(AGNAME) & not standby_agent(AGNAME) <- .send(AGNAME,
achieve, walkOut); +standby_agent(AGNAME); !!standbyOffsets.

+!standbyOffsets :  not (active_agent(AGNAME) & not standby_agent(AGNAME)).

+!walkInOffsets :  active_agent(AGNAME) & standby_agent(AGNAME) <- .send(AGNAME,
achieve, walkIn); -standby_agent(AGNAME); !!walkInOffsets.

+!walkInOffsets :  not (active_agent(AGNAME) & standby_agent(AGNAME)).

+!walkAroundOffsets :  active_agent(AGNAME) & not walking_agent(AGNAME) <-
.send(AGNAME, achieve, walkAround); +walking_agent(AGNAME); !!walkAroundOffsets.

+!walkAroundOffsets :  not (active_agent(AGNAME) & not walking_agent(AGNAME)).

+sleeping :  true <- !rest.

+!rest :  not sleeping .

+!rest :  sleeping & energy(E) <- -+energy(E+1); .wait(500); !!rest.

// === NOT AGENT SPECIFIC PLANS ===

+!lookIntoCamera :  camerapos(CX,CZ) & my_actor(ACTOR) <- ?agentposition(AGENT, X, Y,
Z); moveTo(X,Z, CX,CZ); !waitForIdle(1000,500).

+!lookIntoCamera :  not camerapos(_, _) & my_actor(ACTOR) <- ?agentposition(AGENT, X,
Y, Z); moveTo(X,Z, 0,-3); !waitForIdle(1000,500).

// default waitForIdle = wait 1/2 second, then wait until agent is idle, then wait
another 1/4 second
+!waitForIdle :  true <- .wait(500); if(my_agent(AGENT) & not agentstate(AGENT, idle))
{ .wait(my_agent(AGENT) & agentstate(AGENT, idle)); } .wait(250).

+!waitForIdle(N,M) : true <- .wait(N); if(my_agent(AGENT) & not agentstate(AGENT,
idle)) { .wait(my_agent(AGENT) & agentstate(AGENT, idle)); } .wait(M).
```

List of Figures

Glossary

agentfigure Also virtual agent-figure. Refers to the agent controlled figure on the screen. 2, 3, 6, 13–16, 24–34, 36–41, 44, 45, 59, 61

island-system A concept similar to finite-state machines, used to visualize and model Jason agents. Described more closely in Section 4.2. 35, 37, 40, 45

Jason agent The agent program controlling the agentfigure. 13, 14, 24–27, 34, 37, 61

Live Also live state or live phase. Refers to the state, in which the agentfigure mirrors the movements of the Visitor it was assigned to. 24, 25, 28–30, 32–34, 36, 37, 39, 45

Visitor Active user of the INTRA SPACE installation. 1–3, 6–8, 11, 13–17, 22, 24, 25, 27, 28, 30–41, 43–46, 59, 61

Acronyms

AI Artificial Intelligence. 2, 3, 8, 9, 11, 18, 45

AS AgentSpeak. 1, 9, 10, 17, 20, 22, 24, 34, 44, 45

BDI Belief-Desire-Intention. 9, 10, 17, 19, 20, 44, 45

BSON Binary JSON. 13, 24

CArtAgO Common ARTifact infrastructure for AGents Open environments. 10, 18

FUBI Full Body Interaction Framework. 13, 23, 28, 30, 46

FWF Fonds zur Förderung der wissenschaftlichen Forschung (Austrian Science Fund). 2

IA Intelligent Agent. 10, 18–21, 28, 34, 44

JSON JavaScript Object Notation. 13, 63

MAS Multi-agent system. 10, 11, 17, 18, 20, 22–24, 27, 28, 34

UDP User Datagram Protocol. 13, 22, 24, 43

Bibliography

[BBH+13] Olivier Boissier, Rafael H Bordini, Jomi F Hübner, Alessandro Ricci, and Andrea Santi. Multi-agent oriented programming with jacamo. *Science of Computer Programming*, 78(6):747–761, 2013.

[BFG+95] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. Metatem: An introduction. *Formal Aspects of Computing*, 7(5):533–549, Sep 1995.

[BH+04] Rafael H Bordini, Jomi F Hübner, et al. Jason–a java-based agentspeak interpreter used with saci for multi-agent distribution over the net. *On-line at http://jason. sourceforge. net*, 2004.

[BHW07] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, Inc., USA, 2007.

[BJNDL15] Elisabetta Bevacqua, Céline Jost, Alexis Nédélec, and Pierre De Loor. Gestural coupling between humans and virtual characters in an artistic context of imitation. In *International Conference on Intelligent Virtual Agents*, pages 194–197. Springer, 2015.

[BPR99] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Jade–a fipa-compliant agent framework. In *Proceedings of PAAM*, volume 99, page 33. London, 1999.

[BRASC10] Anton Bogdanovych, Juan A Rodríguez-Aguilar, Simeon Simoff, and Alex Cohen. Authentic interactive reenactment of cultural heritage with 3d virtual worlds and artificial intelligence. *Applied Artificial Intelligence*, 24(6):617–647, 2010.

[Bre10] Michael Brenner. Creating dynamic story plots with continual multiagent planning. In *AAAI*, 2010.

[BSM+14a] Elisabetta Bevacqua, Igor Stanković, Ayoub Maatallaoui, Alexis Nédélec, and Pierre De Loor. Effects of coupling in human-virtual agent body interaction. In Timothy Bickmore, Stacy Marsella, and Candace Sidner, editors, *Intelligent Virtual Agents*, pages 54–63, Cham, 2014. Springer International Publishing.

[BSM+14b] Elisabetta Bevacqua, Igor Stanković, Ayoub Maatallaoui, Alexis Nédélec, and Pierre De Loor. Effects of coupling in human-virtual agent body interaction. In *International Conference on Intelligent Virtual Agents*, pages 54–63. Springer, 2014.

[cap] Captury live - human motion in real-time. `http://thecaptury.com/captury-live/`. Accessed: 2018-11-28.

[CMC+03] Marc Cavazza, Olivier Martin, Fred Charles, Steven J. Mead, and Xavier Marichal. Interacting with virtual agents in mixed reality interactive storytelling. In Thomas Rist, Ruth S. Aylett, Daniel Ballin, and Jeff Rickel, editors, *Intelligent Virtual Agents*, pages 231–235, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[FW99] Jacques Ferber and Gerhard Weiss. *Multi-agent systems: an introduction to distributed artificial intelligence*, volume 1. Addison-Wesley Reading, 1999.

[GC98] Stephen Grand and Dave Cliff. Creatures: Entertainment software agents with artificial life. *Autonomous Agents and Multi-Agent Systems*, 1(1):39–57, 1998.

[GRA+02] Jonathan Gratch, Jeff Rickel, Elisabeth André, Justine Cassell, Eric Petajan, and Norman Badler. Creating interactive virtual humans: Some assembly required. *IEEE Intelligent systems*, (4):54–63, 2002.

[JK03] Bernhard Jung and Stefan Kopp. Flurmax: An interactive virtual agent for entertaining visitors in a hallway. In Thomas Rist, Ruth S. Aylett, Daniel Ballin, and Jeff Rickel, editors, *Intelligent Virtual Agents*, pages 23–26, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[JO] Christina Jauernik and Simon Oberhammer. Intra space - about. `https://intraspace.akbild.ac.at/imprint/`. Accessed: 2018-11-28.

[JZ07] Li Jia and Miao Zhenjiang. Entertainment oriented intelligent virtual environment with agent and neural networks. In *Haptic, Audio and Visual Environments and Games, 2007. HAVE 2007. IEEE International Workshop on*, pages 90–95. IEEE, 2007.

[KB15] Kalliopi Kravari and Nick Bassiliades. A survey of agent platforms. *Journal of Artificial Societies and Social Simulation*, 18(1):11, 2015.

[Kis] Felix Kistler. Fubi - full body interaction framework. `https://www.informatik.uni-augsburg.de/en/chairs/hcm/projects/tools/fubi/`. Accessed: 2018-11-28.

[LB17] Gerard Llorach and Josep Blat. Say hi to eliza. In *International Conference on Intelligent Virtual Agents*, pages 255–258. Springer, 2017.

[Mae95] Pattie Maes. Artificial life meets entertainment: lifelike autonomous agents. *Communications of the ACM*, 38(11):108–114, 1995.

[MBB12] Rachel McDonnell, Martin Breidt, and Heinrich H Bülthoff. Render me real?: investigating the effect of render style on the perception of animated virtual humans. *ACM Transactions on Graphics (TOG)*, 31(4):91, 2012.

[MHK06] Thomas B Moeslund, Adrian Hilton, and Volker Krüger. A survey of advances in vision-based human motion capture and analysis. *Computer vision and image understanding*, 104(2-3):90–126, 2006.

[OFS99] Eugenio Oliveira, Klaus Fischer, and Olga Stepankova. Multi-agent systems: which research for which applications. *Robotics and Autonomous Systems*, 27(1-2):91–106, 1999.

[PL11] Roberto Pugliese and Klaus Lehtonen. A framework for motion based bodily enaction with virtual characters. In *International Workshop on Intelligent Virtual Agents*, pages 162–168. Springer, 2011.

[PPP14] Andre Pereira, Rui Prada, and Ana Paiva. Improving social presence in human-agent interaction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1449–1458. ACM, 2014.

[Rao96] Anand S Rao. Agentspeak (l): Bdi agents speak out in a logical computable language. In *European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, pages 42–55. Springer, 1996.

[RB12] Mark Owen Riedl and Vadim Bulitko. Interactive narrative: An intelligent systems approach. *Ai Magazine*, 34(1):67, 2012.

[RCP11] Surangika Ranathunga, Stephen Cranefield, and Martin Purvis. Interfacing a cognitive agent platform with second life. In *International Workshop on Agents for Educational Games and Simulations*, pages 1–21. Springer, 2011.

[RN16] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[RPV11] Alessandro Ricci, Michele Piunti, and Mirko Viroli. Environment programming in multi-agent systems: an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 23(2):158–192, 2011.

[RS06] Mark O. Riedl and Andrew Stern. Believable agents and intelligent story adaptation for interactive storytelling. In Stefan Göbel, Rainer Malkewitz, and Ido Iurgel, editors, *Technologies for Interactive Digital Storytelling and Entertainment*, pages 1–12, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[RVO06] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. Cartago: A framework for prototyping artifact-based environments in mas. In *International Workshop on Environments for Multi-Agent Systems*, pages 67–86. Springer, 2006.

[SCB+12] Anna Stenzel, Eris Chinellato, Maria A Tirado Bou, Ángel P del Pobil, Markus Lappe, and Roman Liepelt. When humanoid robots become human-like interaction partners: corepresentation of robotic actions. *Journal of Experimental Psychology: Human Perception and Performance*, 38(5):1073, 2012.

[SCFH98] Kenji Suzuki, Antonio Camurri, Pasqualino Ferrentino, and Shuji Hashimoto. Intelligent agent system for human-robot interaction through artificial emotion. In *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on*, volume 2, pages 1055–1060. IEEE, 1998.

[SM12] Miguel A Salichs and Maria Malfaz. A new approach to modeling emotions and their use on a decision-making system for artificial agents. *IEEE Transactions on affective computing*, 3(1):56–68, 2012.

[TW04] Seth Tisue and Uri Wilensky. Netlogo: Design and implementation of a multi-agent modeling environment. In *Proceedings of agent*, volume 2004, pages 7–9, 2004.

[Win05] Michael Winikoff. *Jack^{TM} Intelligent Agents: An Industrial Strength Platform*, pages 175–193. Springer US, Boston, MA, 2005.

[WKSE11] Juan Pablo Wachs, Mathias Kölsch, Helman Stern, and Yael Edan. Vision-based hand-gesture applications. *Communications of the ACM*, 54(2):60–71, 2011.