

ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA  
CAMPUS DI CESENA

---

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA  
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

# TOWARDS AGENTS' EMBODIMENT IN WEB-BASED MULTI-AGENT SYSTEMS

*Tesi in*  
PERVASIVE COMPUTING

*Relatore*

Prof. ALESSANDRO RICCI

*Presentata da*

MATTEO CASTELLUCCI

*Corelatore*

Prof. ANDREI CIORTEA

Dott. SAMUELE BURATTINI

Anno Accademico 2022 – 2023



*To all those who endured this with me*



# Table of contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem statement . . . . .	2
1.2	Proposed solution . . . . .	3
1.3	Use case . . . . .	4
<b>2</b>	<b>Background and Related Work</b>	<b>7</b>
2.1	Multi-agent systems . . . . .	7
2.1.1	Agents . . . . .	8
2.1.2	Environment . . . . .	9
2.1.3	Artifacts . . . . .	11
2.1.4	Bodies . . . . .	14
2.2	Web-based multi-agent systems . . . . .	17
2.2.1	The REST architectural style and the Web . . . . .	18
2.2.2	Multi-agent systems without the Web . . . . .	21
2.2.3	Multi-agent systems with the Web . . . . .	23
2.2.4	Embodiment on the Web . . . . .	27
<b>3</b>	<b>Embodying agents in a Web-based MAS</b>	<b>31</b>
3.1	Motivating factors for embodiment . . . . .	31
3.1.1	Agent discoverability . . . . .	31
3.1.2	Agent communication through behavior . . . . .	32
3.1.3	Agent accountability . . . . .	33
3.1.4	Agent situation-dependent interaction . . . . .	34
3.2	Alternative and integrative approaches . . . . .	35
3.3	Defining a body . . . . .	36
3.3.1	A body is concrete . . . . .	37
3.3.2	A body is identifiable . . . . .	38
3.3.3	A body is clear . . . . .	40
3.3.4	A body is timely perceiving . . . . .	41
3.3.5	A body is focusable . . . . .	42
3.4	Presenting a body . . . . .	44

<b>4</b>	<b>Implementation and Evaluation</b>	<b>49</b>
4.1	Bringing agents' bodies to Yggdrasil . . . . .	50
4.1.1	Creating and manipulating agents' bodies . . . . .	50
4.1.2	Querying and observing agents' bodies . . . . .	50
4.2	Bringing Yggdrasil in a testing framework . . . . .	51
4.2.1	Configuring hypermedia environments . . . . .	52
4.2.2	Improving Yggdrasil architecture . . . . .	54
4.2.3	Improving Yggdrasil quality . . . . .	56
4.3	Use case realization . . . . .	58
4.3.1	Requirement analysis . . . . .	58
4.3.2	Design . . . . .	61
4.3.3	Showcase . . . . .	68
<b>5</b>	<b>Conclusions</b>	<b>71</b>
5.1	Future works . . . . .	72
	<b>Acknowledgements</b>	<b>73</b>
	<b>Bibliography</b>	<b>75</b>

# Abstract

The situatedness of agents in Web-based MASs is a well-studied problem: it calls for applying hypermedia and REST architectural principles to the engineering of the environment, leading to inheriting the non-functional properties of being open, long-lived, and Internet-scale. At the same time, a more specific property than situatedness, embodiment, is not well understood in this context regarding its motivations and implications.

During this thesis, we analyzed the notion of embodiment of agents in Web-based MASs, following the hypermedia principles of the Web and drawing inspiration from robotics. We rooted our work in the one previously done on embodiment in virtual environments by A&A, one of the most prominent meta-models proposing the environment as a first-class abstraction, and its reference implementation, the CArtAgO platform. We found that the motivating factors for embodiment are agent discoverability, communication through behavior, agent accountability, and context-dependent interaction. We also found that a body is an abstraction that must concretize and identify its owner in the environment, mediate the owner's action and perception, and allow its own observability.

We created a test framework for Web-based MASs supporting the embodiment of their agents, enabling us to test a prototype system built on a specific use case to demonstrate the technical utility of our work. The realized framework will also promote further research on the Web-based MASs topic for the WebAgents community group. We believe that embodiment, granting more flexibility to agents, could bring us closer to solving the “arrive and operate” problem, the problem for which an agent should be capable of arriving in a new environment and starting to operate with minimal *a priori* information. Moreover, our concept of an agent's body could be integrated into the A&A meta-model, leading to an extension of the latter.





# Chapter 1

## Introduction

This chapter as a whole presents the question we addressed during this thesis: explore the notion of embodiment in Web-based multi-agent systems, also known as Web-based MASs. We explain in detail what adopting such a context implies, including undesirable alternative approaches to embodiment, and give the motivations leading us to prefer the solution we provided. Then, we present how we envision the embodiment of an agent given the discussed context and motivations, bridging it to the two objectives we pursued to concretize this vision. The first objective was to define the notion of embodiment in Web-based MASs, while the second was to realize a tool that can aid the testing of Web-based MASs, functional in future test beds. The validation happened then by realizing a prototype system based on our testing tool implementing a “yogurt manufacturing” use case, described at the end of this chapter.

The rest of this thesis is structured as follows. The second chapter discusses the context in which we place the thesis work, discussing in detail what a MAS is, what it means to have a system interweaved in the Web, and the roots of the notion of embodiment. The third chapter presents the work done in this thesis, explaining our meaning for an agent’s body in the context of a Web-based MAS through the characteristics it must have and the advantages that lead to its use. The fourth chapter explains the work done for implementing such an abstraction in a reference software platform and creating a testing tool for testing Web-based MASs, which we used to test also our prototype system. The fifth and last chapter concludes the thesis, explaining what we envision as part of a future effort to expand our work on embodiment.

## 1.1 Problem statement

MASs are systems composed of multiple autonomous agents, entities encapsulating their control flow along with the criterion for governing it [46], situated in a shared environment they perceive and act upon [64]. Web-based MASs are a subcategory of MASs where the Web is the infrastructure used for building the environment, and there has been extensive research to exploit the architectural style of the Web, REST, when designing MAS environments, leading to the birth of hypermedia MASs [18]. In hypermedia MASs, where we embed agents in the fabric of the Web, the environment inherits all properties of REST, such as Internet-scale scalability, loose-coupling, reliability, extensibility, and visibility of interactions, supporting the development of long-lived, open, and Web-scale systems [25].

The question we posed in this thesis is to explore the notion of embodiment in the context of Web-based MASs. Embodiment means, first and foremost, to provide each agent with a representation of itself in the environment, with a body, making the agent's actions part of the environment dynamic and to make the agent receive direct feedback from their environment [12]. But embodiment is not only a sensible property in robotics but also for virtual environments like the ones we are concerned with, as the Agents & Artifacts (A&A) meta-model demonstrates through the adoption of the concept of body in its reference implementation, the CArAgO platform [53].

From our research, we understood that four motivating factors exist to promote the embodiment of agents in Web-based MASs: discoverability of agents, Behavioral Implicit Communication (BIC) [59], accountability, and situational interaction. The first factor means that embodiment allows the discovery of agents: the possibility to acknowledge their presence in the environment. The second factor refers to enabling a form of implicit communication where the addressee agents capture some meaning from the behavior the source agent is enacting, while the source agent enacts the behavior to reach its goal and to let the other agents observe and understand it [59]. The third factor refers to the possibility for an agent to monitor the behavior of another agent and make it accountable for its misbehavior when it happens, which is a feature of relevance in open systems like Web-based ones. The fourth and last factor refers to enabling interaction between agents that is context, meaning environment, dependent, stemming from the possibility for agents to advertise their abilities and preferred way of communicating through their bodies.

We could attempt other approaches for implementing the same presented functional properties, but we argue they are not as apt as solutions as the embodiment is for enabling our motivating factors. We could use coordination artifacts, entities to be used by agents for helping them coordinate [65], we

could implement known patterns like the Agent Management System or the Directory Facilitator defined in the Foundation for Intelligent Physical Agents specification [1] or simpler publish-subscribe message brokers. All of them have, to some degree, some disadvantages that make them unfit for reaching our goals.

## 1.2 Proposed solution

Given what we previously said, the solution proposed in this thesis is to push Web-based MASs towards the embodiment of their agents, rooting the solution in the A&A meta-model. This meta-model designs the MAS environment as composed of multiple non-autonomous reactive entities called artifacts, first-class programming abstractions representing tools to be used by agents, collected in workspaces [46]. This decision follows from acknowledging that A&A is one of the leading proponents of the vision of the environment as a first-class abstraction, an abstraction not subjected to any else [65]. This notion is fundamental to support if we want to develop MASs that conceptually integrate the Web [19].

To properly reach this solution, we devised two objectives that we wanted to achieve. The first is to propose a definition for the agent's body concept, which should support the implementation of the four motivating factors described in Chapter 1.1. This definition should be fit for Web-based MASs and especially for hypermedia environments, so we should root it in the principles of hypermedia MASs since they derive from the REST architectural style. This design rationale allows for inheriting the REST non-functional properties, but we also go beyond these principles to formulate a proper solution to our problem. The second objective is to develop a flexible testing framework for Web-based MASs to test their capabilities and functionalities, allowing for easy modeling of scenarios to check the correctness of the system involved in the testing procedure. The testing tool will aid the proceedings of the WebAgents community group [61] devoted to the Web-based MASs research topic.

As a last step, we implement a prototype system both for validating the technical utility of the definition of embodiment we present and for demonstrating our tool capabilities to realize test scenarios for Web-based MASs. The prototype system implementation satisfies a yogurt manufacturing use case we designed, derived from an original one first proposed during the third summer school on AI technologies for trust, interoperability, autonomy, and resilience in Industry 4.0 [58] and further developed in the HyperAgents project [38]. The original and our use case model some part of an assembly line for producing cups of yogurt, and we describe our own fully in the following chapter,

showing how the motivating factors for embodiment reflect in the use case, making it a valid use case for testing our definition.

### 1.3 Use case

A yogurt manufacturing company is deploying a new automation system containing an agent named Carl on top of an already installed MAS, made of two agents named Alice and Bob. These two sub-systems, part of the complete final system, fully embody their agents since the final system shows the motivating factors for embodiment discussed in Chapter 1.1.

Alice and Bob can maneuver the robotic arm the system assigns them but do not expose this ability by default. Instead, they acquire it when they join a workspace where a robotic arm is present, knowing from their programming that they can maneuver it, allowing for situational interaction between them and the Carl agent. The two agents also expose some faulty behavior, meaning that sometimes the agents misbehave and do not perform the action associated with the task they receive. This faultiness means that the newly deployed Carl agent must monitor their behavior to ensure the correct execution of tasks in the system, requiring both BIC between agents and accountability of agents. A final requirement is that, by design, no hard-coding of properties can happen on the Carl agent side. It should dynamically discover its environment and the environment's properties after its deployment, requiring the system to enable the discoverability of the agents' bodies.

After the Carl agent is booted up, it is tasked by its human operator with moving two fully molded cups from the shop floor to the warehouse, waiting for further processing. Carl then starts searching in its environment for two agents capable of using robotic arms for moving cups, finding Alice and Bob. It understands from their representation in the environment which abilities they have and, to task them with the cups moving, decides to start looking at the agents to check on their work. Finally, he tasks the two agents to move their robotic arms to store away the cups, but Alice does not fail to achieve the objective given, while Bob fails in its job. Then, Carl punishes Bob by sending a special command to it and re-tasks Alice to move the cup from the shop floor to the warehouse.

The Carl agent has access to the representation of its environment, allowing the agent to query it and discover information about it. It will allow him to discover that two agents are part of the cup production plant workspace and to identify them by finding their agents' bodies. It will also understand from reading the content of their bodies that the two agents show the capability, i.e., offer the affordance, to send them a message for moving the robotic arm

from place to place. This requirement displays situational interaction since the ability to move their robotic arm is contextual to the workspace where Alice and Bob find themselves, in which the robotic arms are also present.

Carl can focus on Alice's and Bob's bodies to observe their actions, monitor the correctness of their behaviors, and later punish or reward them, making the agents accountable for their actions. An agent's body always generates events related to the start and completion of the agent's actions without them having to program it themselves, which results in the Carl agent not needing to check for the tasks' completion itself since it gets automatically notified. This requirement displays BIC since the monitored agents communicate through their actions, not explicitly meant for communication, some meaning that the supervisor agent can catch beyond performing the task itself. Carl can change its behavior after evaluating the agents' actions and judging them as worthy of praise or punishment.

Since the new system gets deployed on top of the older one, we can not expect these systems to be coherent in their specification while resulting in a concentrated solution. Then, the complete system needs to be distributed and composed of three sub-systems: one representing the environment platform and the others representing the MASs. In this way, we can show the previous four features of the system working seamlessly in a distributed context like the Web.



# Chapter 2

## Background and Related Work

This thesis positions itself in the context of engineering multi-agent systems in the larger field of Artificial Intelligence. This chapter, as a whole, discusses the fundamental properties of MASs, and in particular, we examine the notions of situatedness and embodiment of autonomous agents, the one we are most concerned with. We consider where these properties came from, how they intersect with our research, and where they lead us. The discussion then addresses the Agents & Artifacts (A&A) meta-model for MASs [46], one of the most prominent examples, and how it envisions engineering of the environment dimension.

After explaining the fundamentals behind the context, we present the sub-category of MASs we are most concerned with in this thesis, Web-based MASs. We discuss how the original vision for the Web wanted to enable them from the beginning and which are the properties of MASs that stem from being interwoven in the fabric of the Web. Finally, we present one of the earliest examples of hypermedia environments for autonomous agents and demonstrate, thanks to it, how essential embodiment is on the Web.

### 2.1 Multi-agent systems

A MAS is a system where multiple autonomous agents are situated in an environment they share for interacting and coordinating with one another [46]. We can extrapolate from this definition some fundamental concepts, which we will analyze in more detail later: autonomous agents, environment, situatedness, interaction, and coordination. For now, we will assume simply that a MAS is a set of multiple *loci* of control interacting with each other by exchanging information [46]. Even from this simple description, we can notice how such a system is inherently concurrent and distributed. We cannot derive from the definition any assumptions concerning the timing of the relative

executions of its control flows nor the computational context in which they happen.

Given the previous definition, a MAS is also a decentralized model for building a system: agents are free to collaborate if they share the same goals, cooperate if the agent goals are compatible but not shared, or directly compete if their goals are incompatible. Being part of a society is advantageous for agents: it means not being forced to do everything by themselves but sharing knowledge, interacting with others, and possibly learning to adapt to evolving conditions [42].

Given these properties, we argue that such systems are a versatile architecture any software engineer needs to have in their tool-belt. They can be a means for modeling many real-world situations where we have distributed and concurrent agents performing their tasks to meet their goals. MASs can also be an alternative for developing complex engineering systems where the complexity stems from the presence of multiple autonomous entities but also the complexity of their interaction. In these systems, agents typically need social structures and norms to regulate their overall social behavior, and their shared environment is an essential and efficient source of coordination means for the agents [9]. It could also imply an overall quality improvement of the provided solution, reducing the complexity of translating a problem into a solution using improper abstractions that do not capture all the properties the designer is interested in. Wrong abstractions would lead to more boilerplate, ad-hoc implementations that ultimately degrade the system maintenance and evolution, providing a sub-optimal solution, as already mentioned [48].

### 2.1.1 Agents

Since we said that the notion of agent is central to MASs, as it stands out from the name, we now give it a more detailed definition. A definition of an agent is the one from Russell and Norvig's well-cited book "Artificial Intelligence: A Modern Approach":

"Anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators" [54].

A depiction of this definition, realized by the authors, is shown in Figure 2.1. This property of perceiving and acting on an environment is called *situat- edness* and is quoted in other definitions in agreement with the one presented, such as the one from Maes [42] and the one from Franklin and Graesser [26]. These last two definitions introduce another fundamental concept for an agent definition: *autonomy*. An agent is autonomous if it pursues an agenda of its own, deciding by itself what actions to perform and attend to its goals at its



best. We conclude by stating that the kind of agents we are interested in are, most importantly, autonomous.

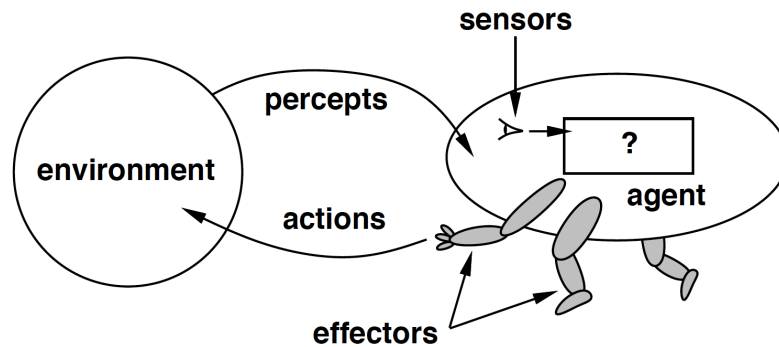


Figure 2.1: Graphical representation of an agent following the Russell and Norvig definition, taken from [54].

Going back to situatedness, it is central to the notion of agency in many ways, from implementation to design. For example, the agent can modify its environment, making it behave as an “external memory” reminding the agent what he has previously done, and in general, the agent can exploit the environmental characteristics to achieve its plans. Situatedness leads to a simplified agent architecture but not to a less complex agent. The interactions between the agent and its environment can lead to emergent functionality of the agent, may it be dubbed intelligence, reflecting the ability of a simple agent to tackle a complex environment more than reflecting an intrinsic complexity. Emergent complexity in an agent is more robust, flexible, and fault-tolerant since no specific component is more complex than others or more intelligent than others, showing graceful degradation upon failing modules [42].

Furthermore, being situated also means it in time, which implies responding in a timely fashion to the stimuli coming from the external world, which is certainly a desirable feature, even if difficult to achieve. It is explicitly stated in the Wooldridge and Jennings definition of agent [66], indicating this property as reactivity and directly deriving from the notion of situatedness. As a side note, the Wooldridge and Jennings definition also underlines the importance for an agent to communicate with other agents, making interaction and MASs consequentially of uttermost importance in the study of agents in general.

### 2.1.2 Environment

Given all the definitions we gave earlier, we saw how situatedness is a pivotal concept for defining an agent, which relates to reacting promptly to

environmental stimuli, which is reactivity. An agent without an environment is not an agent, or, at least, an autonomous agent, the kind of agent we are interested in. But we should not be misled: the environment is not simply a means for defining agents. The environment is a concept with a dignity of its own, a dimension in modeling a MAS capable of assisting the engineer in creating more powerful abstractions, like the notion of artifact we will discuss in the following. It's no accident that a paradigm part of multi-agent-oriented programming concerned with only the modeling of the environment exists: the Environment Oriented Programming field.

The environment has been called a first-class abstraction [65], a design abstraction that does not derive from any other and exists as an independent concept. Weyns et al. propose a fourfold argument for which we should take this stance.

First, the environment provides agents access to their deployment context, composed of the tools they can exploit to achieve their goals. We can argue that without this, an agent could accomplish hardly anything meaningful, confined inside its mental computations.

The second reason relates to having the ability to abstract from unnecessary details since the environment can act as an interface between the previously identified deployment context and the agents accessing it. As already stated, having the tools to capture the right abstraction in the system allows us to build more complex but less complicated systems, meaning systems that can do more with less maintenance and evolution-related problems.

The third reason relates to the environment having the ability to enable the government of interaction and coordination, providing an interface over the resources and also an interface over how the agents communicate with one another. To allow the environment to mediate interaction is to make the environment an active entity per se since it needs to regulate the actions of the agents on itself, on par with them.

The fourth and last argument leading to a first-class environment is that such an idea enables reflectiveness, allowing agents to modify their environment and, therefore, their system as a whole as they please [65]. In the following chapter, we introduce A&A properly, but we can already note how the powerfulness of this meta-model also descends from the fact that it was the one that introduced this reflective level and did it by enabling the dynamic change of the entities that compose the environment [46].

We previously stated that interaction between agents is fundamental in a MAS, as is coordination, the ability for agents to interact in an ordered fashion to achieve some common goal. There exists a model demonstrating this, which shows all dimensions that a MAS engineer should tackle while modeling their system, known as the vowels coordination model [22]. The dimensions are,

not surprisingly: Agent, Environment, Interaction, and Organization. The organization dimension relates to the possibility of governing and regulating social relations between agents, defining norms, groups, roles, et cetera. It goes back to the importance of agents being part of a society and constituting a MAS, which should not happen randomly but in a carefully planned way by the system designer. Given all that was said previously, the environment can be used to reify the “Interaction” and “Organization” dimensions, making it again worthy of its first-class abstraction [65]. The A&A meta-model for MASs presented in the following is another proof of the undeniability of the usefulness of having an explicitly modeled environment.

### 2.1.3 Artifacts

The A&A meta-model exists to tackle the problem of modeling suitably MAS environments [46]. As the name suggests, it wishes to put at the same level agents and the environment, but more specifically, to model the latter with high-level abstractions that can capture its complexity but at a suitable level of generality.

The core idea is to see the environment as made of different tools, named artifacts, which are first-class abstractions agents can use while achieving their goals. These artifacts are passive and reactive entities, meaning they are “activated” by the agents using them, reacting to their requests, similar to objects in the object-oriented paradigm. In contrast with agents, artifacts are not autonomous and proactive: they do not show any “will” to achieve some assigned task or goal. They encapsulate the services and functionalities the environment shows to the agents: they have a distinct function put there by the artifact designer. It helps modularize and decompose the functionalities the environment designer wants to offer to the agents in the system.

The designer defines the function of an artifact in terms of the changes that the agent using it causes to the artifact, changes which translate to the environment as a whole. So, in a change of perspective, an artifact model is defined in terms of which actions, or to give them a better name, operations, it makes available to the agent. Artifacts are not only for use by agents but can also be linked to other artifacts to enable behavior stemming from their composition, allowing for a better modularization and separation of concerns, distinguishing between a usage interface and a linking interface. The usage interface contains the operations agents will use, and the linking interface contains the ones other artifacts will use.

Another fundamental concept in this meta-model is the workspace, which is an abstraction allowing for structuring the space of interactions in a topological way, being it a place where activities involving agents and artifacts can happen.

This meta-model received an implementation thanks to the realization of the CArtAgO platform [52]. In this platform, the programmer can design different artifact types by defining their observable properties, signals, usage interface, and linking interface, as shown in Figure 2.2. Since we already discussed the usage and the linking interfaces, we will concentrate only on observable properties and signals. Observable properties define the state variables an artifact exposes, which can change with time since they are variables. These changes, along with the properties creation and deletion, are part of the percepts received by an agent that focuses on that artifact. Signals are similar to observable properties, meaning that agents can perceive them, but they exist for modeling non-persistent atomic events triggered by operations, meaning that no operations exist to alter their value. Given the definition of observable properties and signals, this meta-model implies an event-based perception model, where the agent receives only the changes in its environment, not a snapshot of its entire state. So, the agent needs to define how to react to each event accordingly, but it will never risk losing any possible update of the environment.

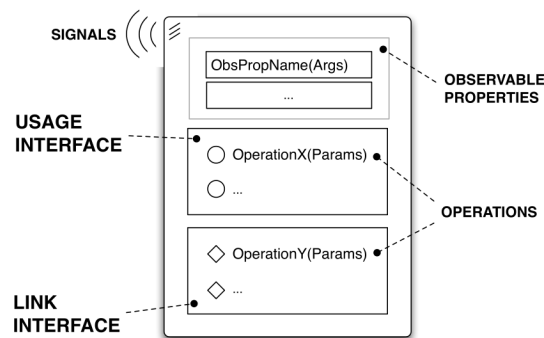


Figure 2.2: An artifact model as defined in the CArtAgO platform, taken from [52].

Another pivotal point of CArtAgO is that it allows the artifacts to be created and destroyed dynamically by agents, making their action repertoire, which contains the operations artifacts offered to the agents, also dynamic. The operations defined in usage and linking interfaces are process-like, so possibly they can be long-term computations that could interleave each other if, for example, they are composed of multiple atomic steps. This decision leads to richer semantics, allowing easier modeling of synchronization actions between agents: for example, with this action execution model, synchronization can happen in the event of an action starting or completing by design. An action succeeds when it completes with success, including the production of related signals and observable properties updates, not merely when it gets accepted

by the artifact that offers it. It allows for simplified programming of agents, not needing to check their percepts to inspect the action's success.

Considering the concurrency model adopted in CArTAgO, signals and observable properties updated by an operation are handled concurrently compared to the control flow executing the operation that updated them. It implies that an agent, having multiple flows of control available, could execute multiple plans concurrently, making its execution more performant although prone to race conditions if not handled correctly.

Each artifact belongs to only one workspace, and an agent needs to join it before it can act on the artifact and, in particular, focus on it, but there is no limitation in terms of workspaces an agent can be in. Workspaces can also be left and created, and, as for joining, these actions can be done at runtime, adding more possibilities for updating the action repertoire of an agent. Workspaces can also be dispersed across different platform nodes to build a distributed MAS.

The whole model of the CArTAgO platform is summarized in Figure 2.3.

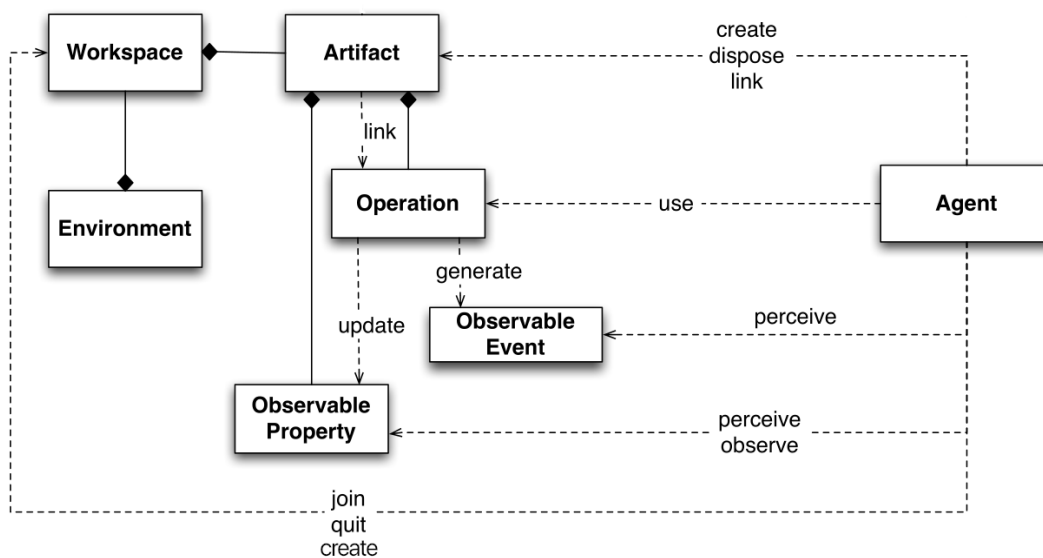


Figure 2.3: The CArTAgO model in UML-like notation taken from [52] with minor edits.

The CArTAgO platform received an extension becoming part of the JaCaMo platform [9], which integrated it with the Jason language and platform for agent programming and the MOISE framework for programming organizations. CArTAgO was born with the capability to support multiple agent programming languages. Even if JaCaMo constrains it to be used only with the Jason language, this CArTAgO extension is still powerful since it can tackle

the agent, environment, interaction, and organization needs of a MAS in a single platform. Agent programming is in Jason’s “domain”, CArtAgO handles environment programming, MOISE preoccupies itself with organization structuring, and all components jointly coordinate interaction, depending on if communication should be direct, mediated through the environment, or regulated by norms. Nonetheless, as the implementation following the ORA4MAS proposal demonstrates [36], the MOISE language and platform can be implemented on top of CArtAgO, reifying its model and its concepts as artifacts. It demonstrates how the environment indeed can be seen as the dual part in MASs to agents, capable of covering all aspects of MASs that are not agents themselves.

The whole model of the JaCaMo platform is summarized in Figure 2.4.

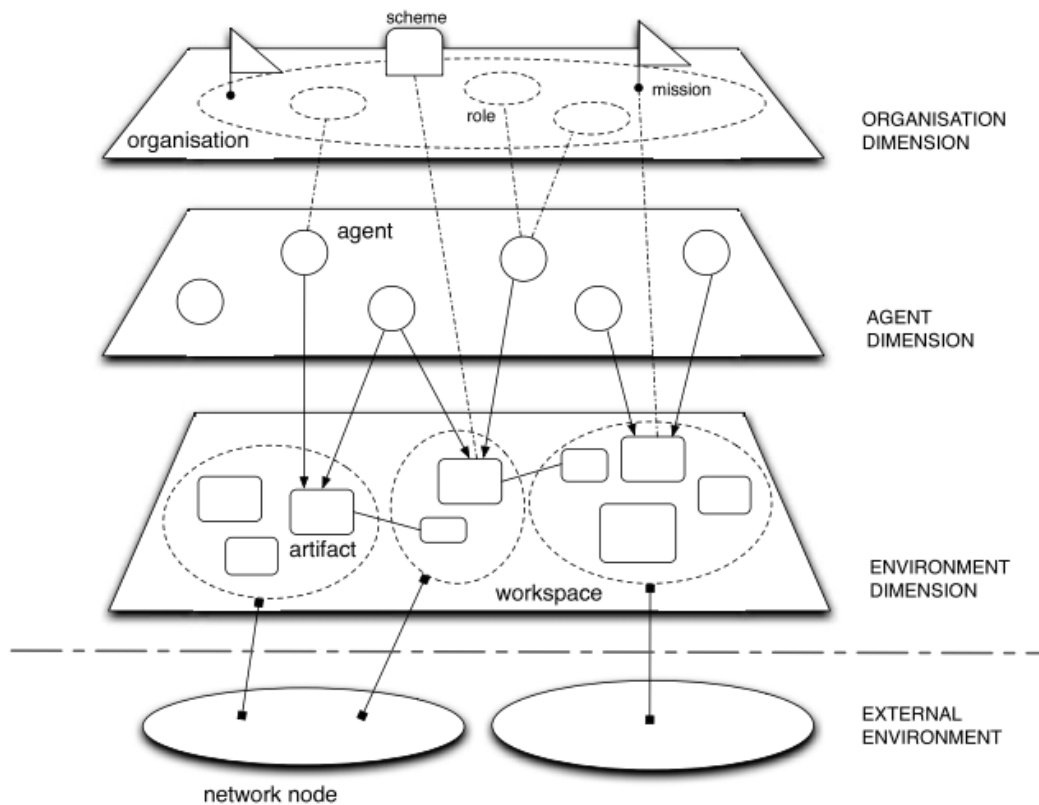


Figure 2.4: A graphical representation of the JaCaMo model, taken from [9].

### 2.1.4 Bodies

Even with its importance, situatedness has shortcomings, too. The proof is the work done in the Robot Lab at the Massachusetts Institute of Tech-

nology in the 90s. The lab was interested in developing an architecture for agents being physical systems situated in the world, meaning robotic agents, that could respond timely and exhibit robust behavior despite dynamic and unpredictable environments [12]. To demonstrate that the subsumption architecture they devised was up to the prefixed standard, they decided to build Herbert, the can collecting robot [20], visible in Figure 2.5. The main defining feature of Herbert was being an embodied agent, meaning having a body as part of the environment in which the agent was situated. The architecture proved successful, standing the test of time, since Herbert is considered one of the forefathers of the Roomba vacuum cleaning robot, one of the biggest commercial successes of the iRobot company.



Figure 2.5: A picture of Herbert, the can collecting robot, now on display at the MIT Museum in the Robot Exhibit, photo courtesy of Chris Atkeson.

As further literature has shown, this property, called embodiment, is more specific than the one of situatedness since it encompasses it while extending it further. For an embodied agent, its body is the “vessel” for perception and action in the environment, and it is through the body that they both need to pass, demonstrating that an embodied agent is also situated. Without a body, an agent could not perceive its environment since it would not have the sensors to do it, and it would not act on its environment since it would not have the actuators to do it. At the same time, having a body does not simply imply that an agent is concerned with its environment, but it also enables the agent’s actions to become part of the environment dynamics and the agent to receive immediate feedback from its environment [12]. It is because now the agent is an element of the environment, it is part of it, that it can be perceived and acted on as any other environmental element. It unlocks the possibility for agents to perceive each other and act on one another, allowing for even more expressiveness in the interaction between the agents.

Embodiment is not only for agents situated in physical environments, like the ones with which robotics is concerned but also for agents situated in virtual environments, as put forward by the A&A meta-model through its reference implementation [53], the CArtAgO platform. This platform implements the notion of an agent’s body as an artifact that gets assigned to an agent whenever it joins a new workspace, and it is part of the workspace joined. In this way, the platform is sure that the actions and the percepts flow through the body of the agent, as per the definition of the body, while restricting the actions an agent can do and the percepts it receives to only the ones related to the workspaces it joined, being the body intrinsically connected to the workspace itself. Moreover, the body serves as a proxy for the platform to the agent, keeping them separated through an interface. It helps satisfy the requirement for which CArtAgO should be agent programming language independent since, for interacting with the CArtAgO platform, an agent interacts with the interface and not with some other platform internals. At last, having a body for agents allows for a simpler distribution of the CArtAgO nodes since the implementation of the operations given by the body interface can involve network calls or not depending on the location of the workspace hosting the body, remaining transparent to distribution. So, a body can behave like a client stub in a Remote Procedure Call middleware, which replaces a local call with a remote one to the correct server without the caller noticing the difference [60].

As a final remark, to underline how much embodiment is indeed a powerful concept, we want to show that it also has its explorations in other areas than Artificial Intelligence and Software Engineering, for example, in the field of Philosophy. Arguably, it could have been that the philosophical formulation came first, and that definition led scientists to apply this concept in their



research field. In particular, we want to quote the definition of embodied cognition by Merleau-Ponty, which states:

“The body is the vehicle of being in the world, and having a body is, for a living creature, to be intervolved in a definite environment, to identify oneself with certain projects, and to be continually committed to them” [43].

Having a body, for a living creature, means being part of the world they inhabit while guaranteeing their existence in the environment and identifying it from all the other existing beings as a separate entity. So, living creatures’ existence depends on their way of experiencing the world since existence and experience are inevitably tied, and the latter happens through their bodies. Therefore, the body is not subjugated to the mind but completes it, acting as its complementary.

## 2.2 Web-based multi-agent systems

A definition for the Web is the space populated with all global network-accessible information in which people interact by retrieving, generating, and removing such information [7]. But the Web is not only somewhere to share information: it is also an Internet-scale, flexible distributed system of knowledge that can be independent of its specific hardware implementation, overcoming incompatibilities between systems accessing it, making it truly open [7].

Since its original vision, supported by its creator, Tim Berners-Lee, the Web was to be used by humans and machines, and later on, after the Web’s first developments, Berners-Lee dubbed this notion as the Semantic Web [8]. In the Semantic Web, the Web of documents existing at the time was not to be replaced but extended with additional capabilities, the most important being the possibility to encode semantic information into documents in machine-readable format to create a Web of data. Since the core Web features were to be left unchanged, the Semantic Web inherited the properties of distribution, decentralization, and universality of information representation and navigation control. At the same time, the Semantic Web would enable the possibility for machines, such as Web clients, not only to parse the syntax of a page, for example, by recognizing links and headers but also to parse the semantics, recognizing what the content of a page was really about [8].

So, we would see the real power of the Semantic Web only when agents would come into play, pulling information from different sources and exchanging it with other agents. The agents’ presence would allow the creation of systems that exhibit intelligent behavior by automatically processing meaning

and then delegating to other services discovered at run-time the task to find the missing information for completing a request, something impossible to do without accessing the request semantics. Then, the last step would be to extend the semantic Web to the concrete world, making physical objects part of the Web [8], enabling agents to “browse through reality” [6]. This vision wants to achieve something also present in other literature and known as a mirror world, a digital duplicate of the physical world [29].

Tim Berners-Lee was right: nowadays, not only humans browse the Web, which has become semantic, but even software agents like crawlers, Web scrapers, and recommendation systems. However, only human agents exhibit autonomous, cooperative, and long-lived behavior, not software agents. So, the era of widespread Web-based autonomous agents and Web-based MASs is yet to come, even if the need for such systems has become more and more pressing. More and more service APIs get published on the Web, and services that integrate these APIs with them, but this integration happens manually, so it is prone to errors, leads to increased costs, and is a time bottleneck on new systems development. It is even more true now that initiatives like the Web of Things [34], bringing the Internet-of-Things devices on the Web, are getting more widespread acceptance, but no Industry 4.0 will happen if we do not have more autonomous clients accessing the Web [19].

Dynamic, open, and long-lived systems require the Web architecture as their foundation, as they need the REST architectural style, which sits at the core of that architecture. In turn, a concept central to the REST architectural style is the notion of hypermedia, now increasingly used for designing highly scalable, dynamic, open, and interoperable systems. So, it is clear that this is the most promising way to design MAS environments, to interweave them in the fabric of the Web, to grant them the same properties the Web has [19]. Then, in the following, we demonstrate the properties the environment inherits by being RESTful, but first, we discuss what REST means.

### **2.2.1 The REST architectural style and the Web**

REST is an acronym for “REpresentational State Transfer,” a name encompassing the fundamental behavioral property of the Web. In this architectural style, hypermedia documents, the core elements of the Web, can be seen as finite state machines, where states are single pages, and the transitions happen by dereferencing the links or submitting forms found on those pages. Then, when the user reaches a new state, its representation is transferred from where the whole finite state machine is hosted, the server, to the client, where the user can use it to progress to the next state, and so on. This principle is also known as HATEOAS, acronym for “Hypermedia As The Engine Of Application

State,” which requires that hypermedia control is embedded into, or layered upon, the state representation, fusing information and control together.

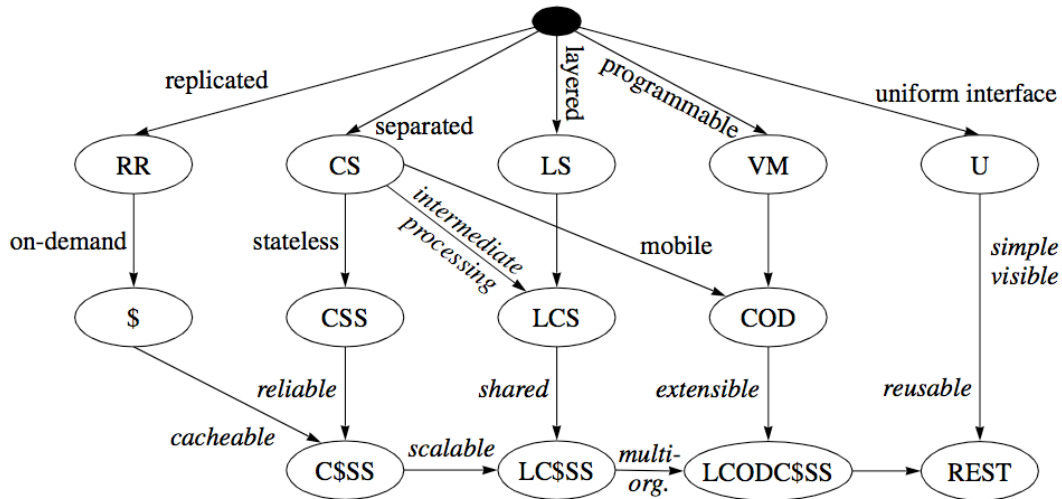


Figure 2.6: A graph representing the relations between the different style constraints applied on REST and which non-functional properties they grant when applied, taken from [25]

To support such behavior, REST inherited some architectural styles visible in Figure 2.6, which grant some non-functional properties that satisfy the use cases for which this architecture was born. The first style is client-server (CS), which allows for separating concerns between the User Interface of a Web application, residing on the client, and the data layer, residing on the server. Moreover, this constraint allows for the independent evolution of the two components, which better supports creating Internet-scale Web applications. But being client-server is not enough: the communication between the two should be stateless, bringing to the client-stateless-server style (CSS), which implies that each request from client to server and response in the other direction should contain all information about itself. No additional application state stored on the server is needed between the two to communicate since all of it should reside on the client. Not needing to rely on additional external information when analyzing requests makes them more visible, meaning easier to process as a single entity. At the same time, the CSS style leads to increased reliability, since if a single request fails, there is no need to replay all that had come before to recover all information about the request. It also leads to increased scalability, since without keeping the connection state, the server needs to allocate fewer resources. The downsides of this style are decreased network performance, since all data needs to be re-sent each time with each request, in-

cluding repeated data, and less server control over application behavior, which is wholly in the hands of the client. Network performances can improve if we add the cache (\$) architectural constraint, deriving from the replicated repository (RR) constraint since a cache is a repository replicating data, leading to the client-cache-stateless-server (C\$SS) style. This style allows the existence of intermediaries between client and server, such as proxies, to eliminate some requests to the server by reusing cached data. Introducing a cache and marking resources as cacheable or not will improve efficiency, scalability, and user-perceived performances because the average latency for requests and the number of requests directly handled by the server will decrease. But this happens at the expense of reducing reliability when data becomes stale, since the user will not be able to know when it is retrieving data from a cache or directly from the server, being transparent to them.

Another pillar of REST is the emphasis on the constraint of uniform interface (U) between components, applying the same general interface between them and not resorting to different, ad-hoc ones each time. It simplifies the architecture, interactions become more visible since they follow the same patterns over and over, and it allows for the independent evolution of components being their implementation hidden by the interface. At the same time, it degrades performance because we cannot exploit the particular properties of each component, which again lie hidden behind the interface. One of the key elements enabling a uniform interface is the notion of resource, representing any information that can be named, a time-varying mapping between a concept and a set made of its representations or identifiers. This notion is apt for building a uniform interface since it provides generality over any representable domain, abstraction from the single concept implementations, and late-binding from a concept to its implementation, allowing for choosing the best alternative or none if none exists.

Then, REST applies the layered system (LS) style, arranging components in layers where each can only communicate with the one above and the one below it and nothing else. It simplifies the architecture, promotes separation of concerns by not agglomerating everything in the same layer, and improves scalability through load balancing. At the same time, it adds overhead and latency to requests by making clients jump through more hoops, downsides already addressed by caching. At last, the final but optional architecture constraint is code on-demand (COD), deriving from the virtual machine constraint (VM), which allows for extending the client capabilities by making it download and execute code locally, giving it the abilities of a virtual machine. With this constraint, the system is more extensible because clients can gain new abilities dynamically, and the whole architecture gets simplified, but the visibility of interactions is then reduced, thus making it an optional constraint.

In conclusion, each system that adopts the REST architecture inherits its upsides of simplicity, visibility, reliability, scalability, loose coupling, and efficiency [25].

### 2.2.2 Multi-agent systems without the Web

In designing a Web-based MAS, we could decide not to follow the REST architecture, as non-RESTful Web services were the standard in the past. It means that interaction between the system components would happen in a Remote Procedure Call (RPC) style, where an HTTP request would represent a procedure call, remaining blocked without a response until the result is available. The HTTP request would contain the serialization of the call and its arguments, while the response would enclose the serialization of the response, using the Web only as a transport layer. Not only was this the standard, meaning it was commonplace, but a family of protocol standards also has been developed to build Web services communicating via RPC, which was the WS-\* standards family, covering from message formatting to interface definition. Moreover, the SOAP standard for message formatting in RPC-based Web services, in particular, is independent of the underlying protocol used for transport, meaning that it can exchange messages not only through HTTP but also with SMTP, the server-side mail exchange protocol. If agents in our system decided to email procedure calls between themselves, surely we would be outside the Web domain, which means that, when following the RPC approach, the Web is a replaceable means of communication not affecting the system architecture [19].

Nowadays, we widely recognize how the RPC approach does not cover all possible use cases in distributed systems, as its object-oriented counterpart does not, the Remote Method Invocation (RMI) or Distributed Object style [60]. The main difference between RPC and RMI is that the latter approach generates a stub for a whole object, and its method calls, not generic procedure calls, are the ones that perform network communication with a remote server [60]. The problem was that RPC and RMI wanted to hide distribution from the developer, using a uniform approach to represent local, fine-grained, and remote, coarse-grained objects or procedures. However, the inherent difference between local and remote objects is not reconcilable, making unifying the local and remote computing models impossible without accepting inconceivable compromises. For one, the difference in latency between an invocation involving a network system call and one not involving it is many orders of magnitude different, a problem that we cannot sweep under the rug as a simple “implementation detail” [63].

But even if we could have the fastest network available, making the latency

difference negligible, three other conceptual problems would arise from memory access, partial failure, and concurrency. Accessing an object placed in an address space different from the one of the caller requires that the infrastructure supporting remote method invocation handles fully all memory accesses, simply because pointers will not be valid in the caller address space. It is an unacceptable compromise in languages that allow pointers declaration like C++, but also in languages where pointers are not available as a construct like Java, passing an argument by reference must be handled differently if the call is a local one or a remote one [39]. Partial failure relates to the fact that in distributed systems, their components can fail independently without leading to the entire system failing, making them more resilient than an equivalent concentrated solution, which can only undergo total failures. These failures can be due not to components failing on their own but because network partitioning happens, a problem not present when dealing with only local procedures and objects. This problem is not as straightforward as an exception and needs more complex handling involving architectural decisions. It is especially true if we consider that distributed systems are concurrent by nature, while concentrated systems can choose not to be. So, unifying those two computational models would imply either ignoring concurrency and forfeiting consistency in the application state when a network partition happens or adding unnecessary complexity when programming concentrated applications, both undesirable solutions [63].

The CAP Theorem [11] in distributed computing well illustrates the choices and trade-offs a software engineer must endure while building such systems, needing to choose between different degrees of consistency and availability when a network partitioning occurs. The theorem again confirms that distributed systems have inherent problems that are unique to them, and need particular care in their design.

Even if we decide not to rely on RPC-based communications, this would not imply automatically choosing a REST architecture style. Choosing REST relies on the non-functional properties left behind when we decide not to adopt this architecture and the requirements that become more difficult to fulfill in this way. A non-RESTful application could have a higher entry barrier, leading to a slowdown in adoption since it could decide not to use uniform interfaces based on hypermedia, which we know are simple and general. The adopted interfaces could limit their structuring, could not support queries properly, or could not allow the creation and fruition of content when linked sources are either not yet available or not available anymore. Moreover, the strong emphasis on text protocols on the Web enables visibility of interaction, another non-functional property that could get lost in non-RESTful architectures.

REST allows for extensible, long-lived applications prepared for change

thanks to scalability and loose coupling. Also, it is an architecture for distributed applications by design, encompassing solutions to problems arising in this context, such as user-perceived latency. Finally, REST allows for the seamless realization of Internet-scale applications, for which their components reside in geographically dispersed intranets separated by different organizational boundaries. Such applications are by definition open, meaning that their components could be at cross-purposes, i.e., components could receive unanticipated load or malformed requests from other system elements due to genuine or malicious intents. In open systems, security becomes a more prominent concern, but change and evolution of the overall architecture must also be expected, as gradual and fragmented. Founding an architecture on principles like separation of concerns and integrating coherent mechanisms for authentication and authorization will help model such systems, an objective more difficult to achieve without REST [25].

### 2.2.3 Multi-agent systems with the Web

Considering all the upsides of REST and the downsides of RPC for our context of interest, which are Web-based MASs, when introducing an explicit, first-class notion of environment situating the agents, it seems natural to apply the REST architectural style to its design, inheriting all of its previously said properties. So, a sensible path towards the engineering of Web-based MASs is to use hypermedia as a uniform interaction engine, situating agents in a distributed hypermedia environment that they can navigate and use in pursuit of their goals, leading to “transforming” Web-based MASs in hypermedia MASs [18]. This decision is why we thought about motivating factors and characteristics of embodiment in the context of Web-based MASs, but keeping hypermedia MASs principles in mind, which does not exclude that we cannot extend our work to more general Web-based MASs in the future. To properly interweave the environment in the “hypermedia fabric” of the Web, we now illustrate three design principles environment engineers should follow, which will lead to the exploitation of the full potentiality of the Web in their solutions.

The first principle states that all entities in a hypermedia MAS and all their relationships should have a uniform and resource-oriented representation in the “hypermedia” environment. It implies that the system elements, such as agents, artifacts, and workspaces after what we said in Chapter 2.1.3, must be modeled as interrelated resources, following the uniform interface described in Chapter 2.2.1. Since interactions between agents and resources should be codified adhering to the REST architectural constraints, the relations between all resources must be modeled by forms and hyperlinks, observing the HATEOAS

principle also described in Chapter 2.2.1.

To support this first principle, one must enforce uniform identification through Internationalized Resource Identifiers of all entities, which allows for global identifiers, valid despite the physical locations of resources and their actual implementation. If this were not true, agents could rely on the uniform identification scheme defined by the Foundation for Intelligent Physical Agents consortium, valid only for agents and not for resources in general, meaning it could not be a working approach on the Web. Otherwise, environment engineers should rely on platforms' decisions about handling identifiers valid locally to nodes and their low-level network information, harming the interoperability and openness of the Web in general.

Also, supporting this first principle means having uniform representations for resources hiding implementation-specific details, which could turn into having semantic descriptions of resources in RDF format following some standard ontology. For example, the Web of Things initiative defines a model for describing an Internet-of-Things thing, called Thing Description [40], along with an ontology for encoding this description [15]. Deciding not to have a uniform representation through semantic Web technologies could mean, for example, defaulting to FIPA's Agent Communication Language message format and its standard description for agents and services, which cannot be applied to resources and cannot be considered a viable alternative.

Finally, encoding relations between resources in the environment leads agents to be able to crawl, interpret, reason upon, and interact with their system, so with both other agents and the environment, by the hypermedia itself, since everything is a resource. Failing to do so would potentially lead to agents necessitating to resort to Directory Facilitators, and Agent Management Systems patterns defined by FIPA to discover agents and services previously registered on them. These approaches are biased towards the locality of agents, which can access only their local Directory Facilitator and Agent Management System that can propagate a decentralized query only up to a certain distance to other analogous components, a limitation to decentralization not present otherwise.

In conclusion, we argue that the uniform resource-based representations principle allows for a better decentralization of the hypermedia environment we are building, with more support for openness, interoperability, and system evolution.

The second principle states that, given a single entry point into the environment, an agent should be able to discover the knowledge required to participate in the system by simply navigating the hypermedia. The idea behind the principle translates to agents that can use the hypermedia, and only the hypermedia, to access the system, meaning without resorting to out-of-band



systems and information, promoting the decoupling of components by not relying on hard-coded settings. Another focal point of this principle is that to have minimal *a priori* knowledge of the system, not only does the knowledge need to be available in the hypermedia, but it must also be findable by navigation, for example, by crawling as previously mentioned. In particular, navigation of the environment by the agents allows for discovery at run-time of interaction modalities with artifacts without hard-coding them, partially solving the arrive and operate problem mentioned in Chapter 1.1 [19].

The last of the three principles is observability, stating that any resource compelling to agents should be observable. This principle is complementary to the others promoting discoverability since crawling is functional until an agent wants to keep track of the evolution of a resource, for which continuous polling for its current representation is utterly inefficient. Having mechanisms for receiving notifications of updates will lead to an overall reduction in system load, promoting better system scalability [18].

A three-layer architecture based on the A&A meta-model reifies the previously presented principles, visible in Figure 2.7. The first layer comprises the agents that access the system, which happens through an abstraction layer implemented by the application environment, the one we have called until now simply environment. The first sub-section of the application environment encompasses an RDF graph modeling the system itself, on which all entities in the system get projected, representing a uniform resource-based interface per the first principle. This first Hypermedia Abstraction Layer lies upon a second sub-layer containing the artifacts that model the environment, which means that for accessing them, the agents can only do that through the hypermedia, per the second principle. According to the A&A meta-model, the artifacts are then part of their workspaces, which can reside in different distributed nodes, one per host, on which the system bases its deployment. Finally, since the application environment is an abstraction layer, it abstracts from what we previously called the deployment context in Chapter 2.1.2, so the external environment made by services and devices the agents can access through the artifacts. The third principle is not explicitly part of the architecture, but supporting the observability of resources in the Hypermedia Abstraction Layer makes it possible to achieve it, which could correspond to observing the changes in the observable properties of the underlying artifacts.

An example of implementation of this architecture in practice is the Ygdrasil platform, implemented as a Web server offering a standard REST HTTP API over a CArtAgO node. This interface allows for executing some of the operations CArtAgO offers to agents, mediating the access to the second sub-layer of the application environment in the architecture, but also allows to access and update the knowledge graph constituting the HAL, the first sub-

layer of the environment. Observation of the knowledge graph is supported thanks to a partial implementation of the WebSub recommendation [30], which allows agents to get notifications when changes happen to a given resource by subscribing with an IRI acting as a webhook, a user-defined HTTP callback. Because of its properties and architecture, this platform has been the object of our work and received extensive refinement and extension, as Chapter 4 will show.

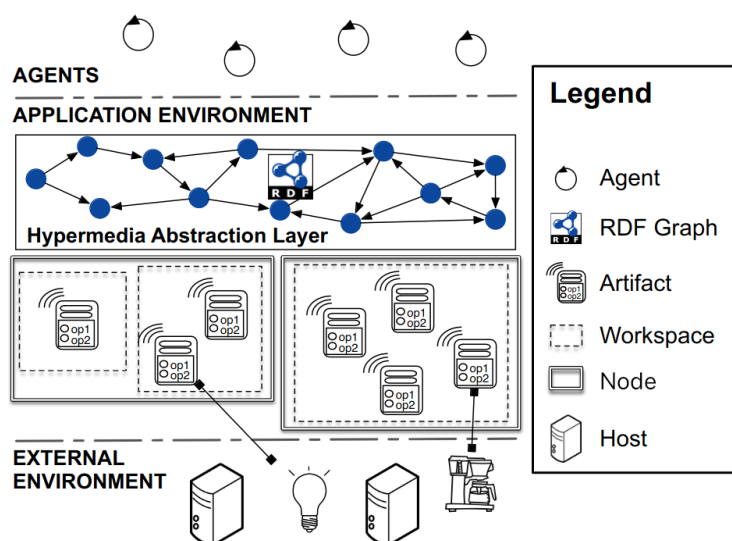


Figure 2.7: The proposed architecture for a hypermedia MAS following the three principles highlighted, taken from [18] with slight changes

Following the previously underlined principles allows a hypermedia environment to be Internet-scale distributed, open, long-lived, and resource-based as any other Web application, interconnecting all system elements it contains. We have shown how having situated agents on the Web would allow them to navigate the hypermedia to discover and interact with these environmental resources regardless of their physical location, as human agents already do with search engines [19]. But again, situatedness on the Web is only half of the story since it enables desirable capabilities while at the same time solving only part of the arrive and operate problem, as already stated. Another property that could lead to a more comprehensive solution is the previously mentioned notion of embodiment, and in the following, we will explain why this is the case and to what additional advantages it leads us to. If situatedness on the Web had its previous explorations and is now quite well-known what it brings us, the embodiment in a hypermedia environment is a relatively new, uncharted concept.

## 2.2.4 Embodiment on the Web

To demonstrate the usefulness of embodiment in Web-based MASs, we will first present a system containing one of the earliest examples of hypermedia environments designed for autonomous agents. As we will see, this is an agent-based system in which agents have the property of being situated on the Web but not the one of being embodied. We will show what functional and non-functional properties this system has and what ones it lacks, hopefully showing that embodiment has its substantial use cases. The system we present is the one commonly known as Mike’s Maze, which owns its nickname to Mike Amundsen, its developer, who started working on its prototype in 2010 [2] and showcased it during the 21072 Dagstuhl seminar in 2021 [10]. Its nickname also is due to its scope, which is an agent trying to solve a maze by navigating it, and the agent’s name reflects this requirement: it is AMEE, an acronym standing for “Autonomous Maze Environment Explorer” [3].

The system is not multi-agent, but we argue that it demonstrates our point nonetheless since the environment the agent navigates is Web-based. A hypermedia document encodes the whole maze, and the single rooms are the “pages” of this document, having a link between them if a door the agent can pass through exists connecting them. To be more precise, the hypermedia document is a knowledge graph serialized in XML based on a custom media type, and all of its elements, from the maze to the single rooms, are nodes of this graph. Later, Tobias Käfer realized an extension to the maze to represent it in Turtle format<sup>1</sup> as part of the All The Agents Challenge at the International Semantic Web Conference of 2021, and the serialization of a room in this format can be seen in listing 2.1. This choice allows for encoding all information about the maze in a form suitable to be crawled by software agents because, in this way, agents can also understand semantic relationships between data, as we mentioned in Chapter 2.2.1.

Crawling leads the agent to move from one room to another by resolving, or dereferencing, the link that leads to the next room’s node, which results in the agent receiving the representation of the new room it finds itself in. The agent is then situated in its environment since it can act on it by moving through the maze until it exits, and it can perceive it by receiving back the changes in the environment it witnesses by changing room. The algorithm the agent uses in navigation does not matter since it can be improved and changed as the developer likes it, but what matters is that this agent is undeniably autonomous, trying its best to find an exit to the maze all by itself. Moreover, the agent could also be a human navigating the maze with the aid of the User

---

<sup>1</sup>The implementation resides in the GitHub repository at the following link: <https://github.com/kaefer3000/2021-02-dagstuhl>.

Interface shown in Figure 2.8a, since on the Web, both humans and software agents coexist, as we stated in Chapter 2.2.

The system shows a deep conceptual flaw that we want to address with the work in this thesis. If we transformed Mike's maze into a MAS, making the agent not alone anymore, we would have multiple AMEE or human agents navigating the maze as they were alone, ignoring each other and unaware of the presence of one another, unless some changes occur in the environment outside the single agent control. But then again, the agents would not have any means to prove that what they see is the clear manifestation of other agents being in the same environment: it could be the more straightforward result of a stochastic environment, so they could not even attempt to prove it.

If the agents have the embodiment property, they most certainly can communicate with one another, which means acting on one another after perceiving and identifying themselves, as already stated in Chapter 2.1.4. An agent then would not doubt if what it is seeing is the body of another agent and if the marks in the environment it perceives are left from something that is not a particular agent's body. So, we argue that some particular features are missing when agents do not have the embodiment property, and consequently, some considerable use cases must be left unimplemented in this way. In particular, we identified four functional properties that could motivate the embodiment of agents in their environment, which we will discuss in the next chapter as part of our contribution to the presented problem. In conclusion, having extensively demonstrated the conceptual and practical utility of the embodiment of agents on the Web, we continue by giving our solution to the embodiment problem following the A&A meta-model.

```
1 @prefix : <http://localhost:1337/01-a-beginner-maze/5#> .
2 @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
3 @prefix vocab:
4     <https://kaefer3000.github.com/2021-02-dagstuhl/vocab#> .
5
6 :it
7   a vocab:Cell ;
8   rdfs:label "Kitchen" ;
9   vocab:collection <http://localhost:1337/#it> ;
10  vocab:east <http://localhost:1337/01-a-beginner-maze/10#it> ;
11  vocab:green <http://localhost:1337/01-a-beginner-maze/10#it> ;
12  vocab:maze <http://localhost:1337/01-a-beginner-maze#it> ;
13  vocab:west <http://localhost:1337/01-a-beginner-maze/0#it> .
14
15 <http://localhost:1337/01-a-beginner-maze#it> rdfs:label "Beginner's
16     Maze (5x5)" .
```

Listing (2.1) Turtle format representation

## The Game



You are in the: **Kitchen**.

You have the following options: **west, east, green, maze, clear**

What would you like to do?

(a) Graphical representation

Figure 2.8: An example of a room in Mike's maze



# Chapter 3

## Embodying agents in a Web-based MAS

This chapter delineates the envisioned solution for embodying agents in Web-based MASs. As a premise, we describe the features that motivate the embodiment of agents in their hypermedia environment. We also present some alternative approaches to enable the motivating factors for embodiment, showing how they do not meet our expected criteria and why embodiment should be preferable in this context. Then, we display the notion of the agent's body we devised, encompassing the main characteristics it should have. For each characteristic, we present its contribution in enabling the motivating factor displayed earlier or the constraints from which it descends. We conclude with a concrete example of a body representation in a hypermedia environment to ground our discussion in a practical example.

### 3.1 Motivating factors for embodiment

Discussing the context illustrated in the previous chapter, we identified four factors that could motivate the embodiment of agents in a hypermedia environment. We will illustrate them along with some use cases related to collaborative editing on the Web in which the absence of such properties will make the use cases more challenging to realize.

#### 3.1.1 Agent discoverability

The first factor is enabling the discovery of other agents, which is permitted thanks to their bodies being perceivable since they are a part of the environment. Discovery of agents allows another agent to understand some aggregated information, such as if no one is in a given workspace or how many

agents are there, but it could also lead to more detailed information, making the agent realize who is in a workspace. In developing an application for the collaborative editing of documents, the discovery of other agents permits showing their icons while they work on a document on top of it, letting other users know who is currently editing the document. If an agent could not have the faculty of discovering other agents, then no icon could be shown, limiting the usefulness of editing a document by not knowing if a teammate is on a given document or not.

### 3.1.2 Agent communication through behavior

Enabling agent discoverability, and especially the perception of other agents in the environment, enables communication between agents through their behavior, so what in literature is known as Behavioral Implicit Communication [59], or BIC. It is a form of communication between agents where they use no explicitly codified communicative action, so the information passed cannot be considered a “message” in the traditional sense. It is still communication, however, since the intention of communicating from the source agent to the addressee agent is still present. Since no explicit communicative action is involved, the other agent actions take their place, having both the intended effect of performing a behavior and letting the addressee agent capture some meaning from such behavior if it observes the source agent [59].

Enabling BIC also means enabling stigmergic interaction, another form of implicit communication that could be considered a specific case of the former [59], as we now demonstrate after defining the notion of stigmergy. Entomologist Pierre-Paul Grassé was the first to introduce the notion of stigmergy while analyzing the behavior of termites building their nests. He wanted to explain during his ethological studies the complex behavior emerging in simple insects when taken as groups and not as single elements when performing their daily tasks. So he presented this definition for stigmergy:

*“La stimulation des ouvriers par les travaux mêmes qu’ils accomplissent”* [33].

A translation of the previous quote could be:

“The stimulation of workers by the very tasks they perform”<sup>1</sup>.

The work done by a worker influencing other workers refers to the fact that, while walking through their environment, so while working, insects release

---

<sup>1</sup>Translation by the author.



their pheromones after them, leaving traces that other insects can follow. Since pheromones are volatile chemical substances, if the path followed by an insect is less efficient than another, meaning longer than an alternative one, pheromones will be less intense on that path since they have more time to evaporate. Then, fewer insects will detect them and follow the inefficient path, leading to an intelligent collective behavior built on top of naive singular behaviors.

Asserted this, we can now understand how this kind of communication is a specific form of BIC. Stigmergy does not refer to direct observation of the source agent behavior but to indirect observation based on *post-hoc* traces, the ones left in the environment by the behavior enactment. Thus, an agent capable of observing a behavior as a whole, from start to finish, can also limit itself to looking only at the result of the actions of another agent, enabling more complex forms of communication.

The stigmergy exploitable in MASs is known as cognitive stigmergy [51], and this connection exists because stigmergy quintessentially involves the fundamental dimension of the environment, which can model all of the concepts shown, from pheromones to evaporation. But, since we have previously shown how BIC is more general than stigmergy, we will devote our efforts to supporting the former through embodiment, which will also encompass supporting cognitive stigmergy. Also, this is why we will refer to BIC and do not explicitly mention stigmergy in the following.

In an application for collaborative editing of documents, communication through behavior is what motivates the implementation of the movement of the different agents' cursors along the document. If a user sees another one working on a paragraph, they could decide not to join them and start working on a different section without overlapping. Not having shared cursors between agents because of the missing BIC it stems from would make editing the same document at the same time more difficult without risking changing what someone else was working on, making them lose the accumulated work.

### 3.1.3 Agent accountability

Embodiment unlocks another motivating factor: holding agents of the system accountable for their behavior, which descends from BIC. Since an agent can look at actions performed by other agents, which we said in the previous chapter is the main characteristic of BIC, it can also decide if those were lawful or not. By doing this, the system holds accountable the agents that performed such actions independently if the monitoring agent decides only to log the actions it has seen or even punish or reward the agents performing them. Simple monitoring and logging enable future auditing from system administration and *post hoc* analysis of possible security breaches, while direct action on

the offending actor is to sanction the wrongful behavior. With the notion of embodiment, there also would not be any possibility for the monitored agents to escape accountability since the observability of their actions would be a mechanism baked into the system, and then accountability if permitted.

This property is particularly noteworthy in open systems like Web-based ones since their components could be at cross-purposes, as we already said in Chapter 2.2.1, making this feature stand out on its own. In an open system, to grant accountability, additional mechanisms need to be in place to ensure that no agent misbehaves since we cannot rely on the correctness of the single agents because we cannot have control over each system component. In a collaboratively written encyclopedia like Wikipedia, not having the possibility to monitor other agents would make it impossible to kick or ban users who actively try to compromise what other users have previously written, leading to a degradation of the content quality.

#### **3.1.4 Agent situation-dependent interaction**

The last factor we identified as a motivation for embodying agents is to let agents interact “situationally,” to enable interaction depending on the current situation. It derives from the idea that after discovering a body, accessing it could communicate information about the body itself, like metadata regarding the abilities and the capabilities of the owner agent. For example, an agent employed in a system managing a shop floor in a factory could advertise the ability to know how to operate a robotic arm. This metadata can also contain agent preferences regarding communications from other agents, so addresses and formats for the messages the owner agent wants to receive [67]. The information found could depend on the body implementation or the specific workspace in which the body finds itself, i.e., it enables interaction between agents depending on the current context or situation, hence the name. In developing a collaborative document editor, situational interaction could be the basis for implementing a mechanism for contacting users in different ways depending on whether they are online. If they are online, observing their body would allow others to discover that a chat can be initiated with them, while if they are offline, their body would communicate to rely on email communication instead. These are preferences regarding communication exposed through bodies like the ones we were referring to before, making this an example of interaction depending on environmental conditions.

## 3.2 Alternative and integrative approaches

We could attempt other approaches for implementing the same presented motivating factors, but we argue they are not as apt as solutions as the embodiment is in this particular context. For example, we could introduce a coordination artifact [65] in the system, an artifact to be used by agents for helping them coordinate, and in particular, register to and leave their information in, to support the discovery of agents and their capabilities. It would violate our principle of minimal *a priori* knowledge since all agents joining the system would need to be programmed at least knowing about the existence of this artifact and its interface, leading to less open systems.

Moreover, introducing a single artifact in the environment represents a centralized solution, which, in distributed systems, could represent an unacceptable single point of failure if high availability is a desired non-functional property. Replicating an artifact multiple times containing the same information would lead to difficulties related to the consistency of replicated data, primarily when network partitioning occurs [11]. Embodiment is, by nature, a decentralized approach, and each body contains only the data referring to itself, so if a network partitioning occurs, making some information about bodies inaccessible, it would be that information related to those bodies unavailable in any case.

The functionality and the interface of the coordination artifact could be standardized, for example, using some known patterns like the Agent Management System (AMS) or the Directory Facilitator (DF) defined in the Foundation for Intelligent Physical Agents (FIPA) specification [1]. However, using the AMS pattern would place more responsibility on the artifact than necessary since it would also have a supervisory role in the system, adding more complexity than is strictly required. Even more, MASs typically implement their AMSs as agents, thing demonstrated by the fact that the FIPA specification reserves a special agent identifier for it, while what we are trying to model is a service to be used by agents, hence an artifact, a completely different kind of abstraction following the A&A meta-model. Using the DF pattern would require the creation of a network of federated artifacts for supporting decentralized queries, and the query propagation would happen only up to a maximum depth level, undermining scalability and evolution of the system [18]. A less complex solution could use both the message broker and publish-subscribe patterns. In this case, the coordination artifact would be the message broker that the agents publish on and subscribe for information, but then all knowledge about the functionalities of such an artifact would need to be hard-coded into the agents.

Moreover, embodiment grants a native integration with the MOISE+ or-

ganizational model [36], an integration otherwise absent with coordination artifacts without additional effort. If we recall the “communicability through behavior” feature quoted in Chapter 3.1.2, we know that bodies notify their observers of the actions the owner agent is enacting. As we will show, this happens through the generation of action completion events, enabling the implementation of special rules in the system, called “count as” rules, which automatically translate the completion of an action into the completion of a goal. It implies that embodiment supports these rules without additional programming on the body side while using coordination artifacts instead would force their designer to program them explicitly to produce these events. Lastly, using MOISE+ would require the designer to define groups, roles, goals, missions, and norms binding roles to missions for the system. What we said demonstrates the unwieldy overhead in the specification and implementation of MOISE+-based organizations for simple, flexible, and automatic monitoring use cases, situations for which we argue embodiment is fitter, making these two approaches natural complementaries.

### 3.3 Defining a body

Now, we want to define an agent’s body in a way that encompasses all the relevant characteristics for describing it.

**Definition 3.3.1** (Agent body). An agent’s body is an artifact reifying its owner in the environment, related to its owner’s identity, even if the identity relationship is hidden from others. It mediates action for the agent in a transparent-by-default manner and perception in a timely fashion. It allows others to observe the owner and perceive affordances to interact with the owner, affordances depending on the environment.

In the rest of this chapter, we elaborate further on this definition to justify it by exploring each of the characteristics that compose it, characteristics which we identified during our research. Without pretense of completeness, these characteristics should represent an agent’s body abstraction, but they should not be supported by thinking of an agent’s body as a software analogous to a human body. This idea is openly misleading, even if it accords with a concept our common sense suggests. Instead, the body characteristics must be only those either aiding the implementation of the previously shown motivating features or those representing some hypermedia constraint we adhere to. So, after explaining each property in detail, along with from what ideas it stems, we will show which motivations induced them. We are not advocating that our body definition is always valid, *au contraire* its applicability strictly depends on the motivations and the context that induced it.

### 3.3.1 A body is concrete

**Definition 3.3.2** (Concreteness). Concreteness is the property of an agent's body to always be reified in the environment it inhabits, either explicitly or implicitly.

The first property we find essential for an agent's body is concreteness: for a body to be concrete means to have a representation in the environment. Following from the A&A meta-model, we get that concreteness implies that the body must be an artifact and, being so, it must pertain to a specific workspace, which in CArtAgO is the workspace the agent has joined thanks to that distinct body. The body is not only a bridge between the agent's mind and its environment in general, but it is a bridge between the mind and a particular workspace, meaning that an agent could be reified multiple times into its environment depending on which workspaces it joins.

In a hypermedia MAS, to create a hypermedia environment, the architecture must represent all system elements as interlinked resources, accessible through a uniform interface, as the relationships between them. In our reference architecture, this led to superimposing a knowledge graph representing the environment upon a lower layer purely made of artifacts and workspaces [18]. We then argue that a hypermedia environment should represent bodies in its knowledge graph as resources as it does for the other artifacts. This resource-based representation enables discoverability as a functionality deriving from embodiment. If an agent's body is perceived in a workspace, then it can be concluded that the agent joined that workspace, or, at least, that an agent joined that workspace, if identification information is not provided.

It could happen that agents do not need a representation in the environment since it could increment the complexity of the system implementation without a valid reason. In those cases, the environment knowledge graph should not explicitly represent bodies, which means that environment implementations can also decide to represent bodies implicitly. For example, this could be the case when agents limit themselves to interacting by employing coordination artifacts, such as blackboards and tuple spaces: in those cases, all that is needed is the artifact chosen for interaction since it fully encapsulates its usage. As we will see, even if we introduce the notion of a body through which an agent can act on its environment, artifacts retain their property of completely hiding inside them the service or functionality they represent. The environment knowledge graph can still acknowledge the existence of bodies in some implicit way, for example, by giving them an identifier and relating it with the identifier of the agent owner of the body. Another way could be to state the body's existence in the semantics of the ontology used for modeling the environment, so, for example, every time an agent is "part of" a workspace in the model,

it is implied by the ontology that the agent has a body in that workspace, without explicitly representing it.

This minimalism of representation is in accord with the principle for which we strive to keep systems as simple as possible by making their environment not contain more than necessary. But we are not renouncing making the body a part of the environment it inhabits because otherwise, it could not be a link between the agent's mind and the environment, which is its nature by our definition.

### 3.3.2 A body is identifiable

**Definition 3.3.3** (Identifiability). Identifiability is the property of an agent's body to be always related to the owner agent and, being so, to the owner identity independently from system constraints preventing other agents from discovering this relationship.

Before delving into the next property we established for agent bodies, identifiability, we first need to introduce the notion of affordance. Following Chemero's definition, an affordance is a relation between the agent's abilities and the environment's features, perceivable by the agent itself [17]. It means an affordance is what an agent perceives about its environment offering, or affording, to it to carry on an action, provided that the agent has the ability required to perform that action. For example, the affordance to climb is that hint the agent feels whenever it has the strength to do it, which is the ability to perform that action, and finds itself in front of a surface with stable grips, which is the environment with the appropriate features.

Now, knowing what an affordance is, the question that naturally follows is what kind of affordances an agent would perceive thanks to the body of another agent. We argue that having other agents' bodies available makes an agent capable of identifying their owner but also allows interacting with it. Similarly to humans, for example, entering a room where there are other people enables someone to know who's in the room, call them for attention, or speak to these people simply because it can interact with their body and then their perception system. So, an agent entering a workspace will be able to understand which are the bodies part of it, then which are the agents who currently joined the workspace, and decide to interact with said agents. This feature works for every workspace, provided that it allows for discovering artifacts in them, as for humans in our example, which are always capable of visually identifying and interacting with other people in a room unless it is entirely dark.

We then argue that the first affordance an agent perceives about another agent's body is related to retrieving the identity of the owner of that body, which is the sole proprietor, distinguishing it from the other agents. We could

model this affordance as an action to retrieve the agent's identity offered by its body or as a piece of information part of the body itself. In a social network, for example, account profiles can be regarded as the embodiment of users on the platform. In this case, the only way to identify and distinguish a user in the platform is by their profile and the information stemming from it: two comments under a post were left by the same user if the author's name in both refers to the same profile, which is part of the profile itself.

Since a body is always associated with the identity of its owner, if agents other than the proprietor control its body, other agents would not be able to distinguish between them and the owner, regarding all ultimately as a unique entity with the same identity. But this would not stop, for example, from holding the body accountable for the behavior it enacts. Coming back to the social media example, if some breach in a profile happens, then multiple agents would be controlling the same body, and other users of the platform would not be able to distinguish only from the profile information if the owner of the profile or an attacker did a specific action. At the same time, the unusual behavior could alert the other users and make them distrust that specific profile, "punishing" it for the behavior it enacted independently from the actual user controlling it.

There could be cases where forbidding to make explicit the agent identification from its body could be beneficial, i.e., it allows to keep the body anonymous. Body identification allows an agent to discover who is in a workspace, observe the actions of a specific agent, and hold it accountable for its behavior. However, hiding the identification relationship does not impede the implementation of these functionalities. It merely reduces their scope: an agent would still be capable of discovering that there are bodies in a given workspace, simply that it would not know whose body they are. Similarly, the agent could observe actions and monitor the behavior of other agents through their bodies, simply without knowing which agent in particular is performing them and which agent is rewarding or punishing for such behaviors. Ultimately, anonymity is not binary but ranges on a spectrum from complete identification to total blindness, passing through distinguishing an agent as part of a more or less ample subgroup without knowing which member of that specific group is. We argue that limiting bodies to be always explicitly identifiable would be too much of a limitation in their utility, preventing exploring other attractive use cases. Moreover, this idea does not forfeit the identity property: the association between the body and the owner's identity would still exist in this case, but the system requirements prevent other agents from discovering it, making it implicit.

### 3.3.3 A body is clear

**Definition 3.3.4** (Clearness). Clearness is the property of an agent’s body that obligates the owner always to use its body during its actions but in an implicit way in default conditions.

Another fundamental property we found for agents’ bodies is clearness. In humans, some mind processes are not tied to deliberate action and happen without the subject thinking about them: this is the underpinning of the dual process theory in psychology and its accounts of reasoning in particular. This theory states that in every brain exist two systems labeled system 1 and system 2 [57]. The first is related to intuition or automatic process, so to the reasoning part preoccupied with unaware or unintentional actions: the reasoning the brain decides to carry on on its own, without the conscious mind involved. The second system relates to proper reason, the active, conscious human activity that follows logical standards. Even if dual process theory nowadays seems limited in explaining the body of knowledge we have about the human mind [47], we keep it as part of this explanation, being nothing more than a practical example.

In our model, we want bodily actions to result automatically from the system 1 in motion, as it is for humans. Actions happen without the agent being conscious of using its body in doing them, even if the action still occurs and the body nonetheless exists. This choice stems from two different needs, one coming from the A&A meta-model and one from the principle of economy of representation we already outlined. If all actions were required to involve a body explicitly, we would misalign our notion of embodiment with A&A. In CArtAgO, for example, agents always explicitly using their body to act on an artifact translates to agents that can only act by using a “do action” operation in the usage interface of their body artifact. This constraint would tie all artifact operations to bodies, and without an explicit body, there would be no possibility to interact with artifacts, robbing the notion of an artifact of its independence and capability to encapsulate its usage fully. Moreover, if we recall operations like the one for joining a workspace in CArtAgO, the operation necessarily creates a body artifact in the joined workspace. But, if the body should always be involved explicitly, it would be a burden on the agent to instantiate such an artifact, which is unacceptable. Secondly, involving the body explicitly every time the agent acts would add an unnecessary indirection level since the agent could achieve the same by directly using an artifact, following A&A. Then, this constraint would add more complexity without a reasonable justification, which could also lead to the degradation of performances without justification.

The clearness property does not entail that the agent could not use its



system 2 to act, so being aware of its own body while doing something is not prohibited, but it should not be the norm. It should be an exception performed by a self-aware agent wanting to reason about itself and the actions that it is performing. The same goes for humans, who generally perform their tasks using their bodies without thinking about it unless they want to deal with more cognitive workload for some reason. So, a motivation to consider the body as visible is to be fully aware of when it gets used and when it is not, so to not be transparent about the body usage but reason on it.

The name of this property stems from the two different aspects it wants to model: the agent uses its body without putting in doubt its use, and it is evident to the agent that it is using it, meaning that the body is “clear” to the agent. But at the same time, the body is transparent to its owner since the agent typically uses it without actively thinking about it, which implies again that the body is “clear” to the agent.

Giving the clearness property as valid implies that all the information in the body representation should not explicitly reference the body itself. The fact that the body mediates access to this information and the information depends on abilities induced by the body are not valid motivations for explicitly referencing it. As it happens in English, we do not say that a body can walk fast, but rather that a person can walk fast, even if it is a property of their body and is a piece of information discoverable only through their body. The absence of explicit references in the body to itself means that an artifact operation should not contain the agent’s body as a parameter in its signature unless it has a valid reason, for example, because that operation will mutate the given body.

### 3.3.4 A body is timely perceiving

**Definition 3.3.5** (Timely perception). Timely perception is the property of an agent’s body to always communicate, in a timely fashion, the percepts it receives from the environment to its owner agent.

Since we argued that bodies are vessels for the perception of stimuli coming from the environment for the owners of such bodies, we claim that bodies should be responsible for notifying the agent about these stimuli. We call this property timely perception since its objective is to codify the possibility for an agent to perceive its environment, to make it perceptible, by its body. The timely part refers to how the body should provide the feedback, which we argue should respect the deadlines the agent sets for this feedback to be received.

Dealing with time and, most specifically, simultaneous events is a complicated task in distributed systems, the category of systems in which MASs fall. So, when we envisioned this property, we assumed the possibility of having

simultaneity between the percept generation by the environment and its transmission by the body, a transmission that could take time to reach the agent and not be simultaneous from its point of view. This property opens multiple discussions since nothing guarantees this simultaneity, and we will not discuss here under which conditions could happen and the conceptual consequences of it for a system.

The advantage of having this property is clear: the agent does not have to proactively query its body to get new percepts since they are provided by the latter, simplifying the internal architecture of an agent. This property also helps distinguish a body from other abstractions similar in features but different in conceptual level, like mailboxes or social media profiles. A mailbox or a social profile, assuming no notification mechanisms are in place, will not alert the agent owning them when something gets updated. For example, mailboxes do not alert their owner when a new letter arrives, and profiles do not notify when changes happen in the displayed information.

The timely perception property does not entail a specific implementation for the mechanism for receiving environmental stimuli since this is a concern proper of a different abstraction level. The platform implementing the agent architecture could still poll their body for percept notifications instead of relying on reactive mechanisms implemented in the body, provided that the agents do not need to call explicitly a “perceive” operation to access their percepts.

This idea stems once again from how the human body works: the mind is not continually checking if the body does perceive something new, but it is the body that has its sensors connected to the system a person uses to process stimuli, which is the nervous system.

### 3.3.5 A body is focusable

**Definition 3.3.6** (Focusability). Focusability is the property of an agent’s body always to allow other agents to observe the body itself and perceive affordances to interact with the owner, affordances dependent on the environment.

We argue that to unlock three motivating factors deriving from embodiment, communication through behavior, accountability, and situational interaction, another affordance needs to be perceived thanks to an agent’s body, which is the affordance to focus the body. In other words, a body artifact needs to be focused on by other agents, to be looked upon, to let the agents receive its properties, the changes it goes through, and the events it generates. In this way, we allow other agents to inspect the capabilities an agent has, to get notified when these change, and to coordinate with the body’s owner

by initiating their behavior in correspondence to the start or the end of some other behavior enacted by the owner agent.

We found no better entity to generate signals about what an agent is doing than the agent's body since the parallelism with humans is once again remarkable. If we consider people performing a stadium wave, only the body movements signal the beginning or the end of a person's stretch, so focusing on the body of the previous person is the only way for the next one to keep the choreography coordinated. Focusing on a body would support BIC because getting notified of the signals corresponding to an action starting and ending would allow an agent to understand which actions are successfully performed by another and then which is the behavior that the latter is enacting, leading to the focusing agent reasoning on the meaning behind those actions. The support for accountability is the natural consequence of this: we can imagine creating a supervisor agent who monitors all the others by "looking" at their subordinates' behavior after focusing on their body, checking if what they are doing is expected and correct. Then, for the supervisor agent, it would be simply a matter of logging the agents' actions to allow a following auditing session or taking direct action against the offending agents.

Given what we previously said in Chapter 3.3.2, we also advocate for a body representation to contain information about its owner agent, as it happens with human bodies. Looking at other people allows for deducing some contextual information, such as the age, origin, and abilities of such people: a face with wrinkles signifies that a person is old, and a tall person can reach higher shelves not accessible to others. So then, focusing on the body artifact should also be used for retrieving information about the body itself or the capabilities an agent has, pieces of information dependent on the context, which is the environment the body finds itself in. Using the focus mechanism to retrieve such information also allows for modeling it through observable properties, which can change dynamically, and the focusing agents can get notified about their updates without any additional action. This feature supports the situational interaction functional property since focusing agents can understand what the owner of the focused body can do and then engage in interactions, such as exchange messages, which can depend on such abilities, for example, asking to complete a task the owner agent can do, but the focusing agent cannot.

It is important to note that agents can also look at their bodies, focusing on their bodies to understand what information about themselves is publicly exposed and, possibly, change it. The usefulness of such an operation lies in those cases where the body creation is not managed by the agent, or at least not entirely, as it happens on social media platforms, where these last ones manage the account creation while the users only "fill in the blanks"

with their information. So, the users need to check their profiles to gather information about themselves and fine-tune what others can see about them on the platform. We see agents capable of doing this as exploiting the reflective level of the environment since they are reasoning about their environment and changing it to fit their needs.

Secondarily, we could argue that an agent looking at the properties in its body representation seems like a baby looking at their body for the first time and discovering what it can do. We could then regard that as a form of self-reflection, such as looking in a mirror and perceiving the self, a self-conscious act, complementing how an agent self-reflects internally, even if it translates technically into manipulating another artifact in the environment.

An agent's body could contain multiple properties, but the only ones that should be present should be the ones directly related to the body or the workspace the body is in. The other properties are ones the agent has regardless of its body: they could be part of a representation of the agent's mind or, equivalently, a general representation of the agent, not tied to a specific workspace.

### 3.4 Presenting a body

As a final remark about this thesis contribution, we present a possible representation of an agent's body in Turtle format, visible in Listing 3.1, as it would appear in a knowledge graph part of a hypermedia environment. The representation uses the Thing Description (TD) ontology briefly presented in Chapter 2.2.3, along with other ontologies it requires to format a TD correctly. Moreover, the presented listing uses the hMAS ontology [37] for describing hypermedia MAS concepts like Agent or Artifact, which descend from the A&A meta-model.

This TD displays the body of an agent named "Alice," as shown by lines 9 and 10 in the listing, a body created by the environment in the workspace identified by the URI `http://localhost:8080/wksp/w0`, as visible in line 12. The simple presence of this representation in its knowledge graph covers the notion of concreteness since the characteristic requires that the environment represents the bodies of the agents that are part of itself. Line 13 in the listing, associating this body with the URI of its owner, covers the characteristic of identifiability instead. Since this environment does not hide the identity relationship between this body and its owner, it is possible to discover it explicitly, so it represents it as a triple relating the URIs of the body and the owner agent. Being the body both concrete and identifiable allows for discovering it as part of a specific workspace and enabling all the other subsequent

motivating features of embodiment.

The body contains two action affordances: one for a sender to tell the agent some message and one for focusing on the body itself. The first affordance advertises to other agents how to communicate with the owner, enabling the latter to specify preferences in communication and so enabling situational interaction. Furthermore, a specific affordance could be published or retracted dynamically by changing the body TD, so we could decide to remove this first “tell” affordance whenever the agent finds itself in a workspace in which it cannot receive messages. This possibility makes the interaction between agents reach a higher level of context dependency.

The second affordance allows focusing on the body by making an HTTP request to a specific endpoint and passing a callback URI as part of the request body. The system will notify the client through the URI of changes in the body representation or actions the agent performs through that body. This affordance enables the property of focusability of bodies while supporting the motivating factors of communication through behavior and accountability of agents for their actions. At last, this affordance allows for a more straightforward implementation of situational interaction, not forcing the client to poll the body for changes in its TD.

No affordance receives the body as a parameter in its request, so no explicit reference to the body is present in the representation unless it is the target of an affordance, as the “focus” affordance shows. This decision illustrates how we could implement the principle of clearness of bodies in the environment knowledge graph. At the same time, we do not display the principle of timely perception in action since it relates to a behavior the body needs to enact without the intervention of other agents, so we do not want to advertise it on an interface thought for agents.

Finally, in Listing 3.2, we show a workspace of a hypermedia environment containing an agent. This listing is an example of what we were referring to as an “implicit” representation of a body. If the ontology specification indicates that having an agent in a workspace implies the existence of its body in the workspace, then the body is still represented, but simply in an “invisible” manner.

In the next chapter, we will display a software implementation incarnating all previously shown characteristics and concepts, demonstrating and evaluating their actual utility.

```

1 @prefix hmas: <https://purl.org/hmas/> .
2 @prefix td: <https://www.w3.org/2019/wot/td#> .
3 @prefix htv: <http://www.w3.org/2011/http#> .
4 @prefix hctl: <https://www.w3.org/2019/wot/hypermedia#> .
5 @prefix wotsec: <https://www.w3.org/2019/wot/security#> .
6 @prefix js: <https://www.w3.org/2019/wot/json-schema#> .
7 @prefix ex: <https://example.org/>.
8
9 <http://localhost:8080/wksp/w0/agt/a0> a td:Thing, hmas:Artifact,
   ex:Body;
10   td:title "Alice";
11   td:hasSecurityConfiguration [ a wotsec:NoSecurityScheme ];
12   hmas:isContainedIn <http://localhost:8080/wksp/w0>;
13   ex:isBodyOf <http://localhost:8080/agt/a0>;
14   td:hasActionAffordance [ a td:ActionAffordance;
15     td:name "tell";
16     td:hasForm [ htv:methodName "POST";
17       hctl:hasTarget <http://localhost:8081/inbox>;
18       hctl:forContentType "application/json";
19       hctl:hasOperationType td:invokeAction
20     ];
21     td:hasInputSchema [ a js:ObjectSchema;
22       js:required "sender", "content";
23       js:properties [ a js:StringSchema, hmas:Agent;
24         js:propertyName "sender"
25       ], [ a js:StringSchema;
26         js:propertyName "content"
27     ]
28   ], [ a td:ActionAffordance;
29     td:name "focus";
30     td:hasForm [ htv:methodName "POST";
31       hctl:hasTarget <http://localhost:8080/wksp/w0/agt/a0/focus>;
32       hctl:forContentType "application/json";
33       hctl:hasOperationType td:invokeAction
34     ];
35     td:hasInputSchema [ a js:ObjectSchema;
36       js:required "callbackURI";
37       js:properties [ a js:StringSchema;
38         js:propertyName "callbackURI"
39     ]
40   ]
41 ] .
42 ] .

```

Listing 3.1: Example of an agent's body in Turtle format.

```
1 @prefix hmas: <https://purl.org/hmas/> .
2 @prefix td: <https://www.w3.org/2019/wot/td#> .
3 @prefix wotsec: <https://www.w3.org/2019/wot/security#> .
4
5 <http://localhost:8080/wksp/w0> a td:Thing, hmas:Workspace;
6   hmas:contains <http://localhost:8080/agt/a0>;
7   td:title "Production";
8   td:hasSecurityConfiguration [ a wotsec:NoSecurityScheme ] .
9
10 <http://localhost:8080/agt/a0> a hmas:Agent .
```

Listing 3.2: Example of a workspace in Turtle format.





# Chapter 4

## Implementation and Evaluation

We brought the previously presented model for agents' bodies to the Yggdrasil framework since it is our reference system for building hypermedia environments, incarnating all principles we want to adopt about Web-based MASs and the REST architectural style while following the Agents & Artifacts (A&A) meta-model [18]. So, in what follows, we list the additive changes we implemented in the framework to support the factors motivating the embodiment of agents and the characteristics that a body should have, conforming to the body and workspace representations shown in Chapter 3.4. Showing how the presented concepts get reified in our implementation, so how we gave them a technical basis for their existence, helps in grounding the thesis work.

Since another objective of this thesis was to develop a testing tool for Web-based MASs, we decided to attain it not by starting from scratch but by reusing the Yggdrasil framework, a reason again motivated by the functionalities already implemented in this framework. Because of this, we introduced support for declarative configurations of the environment, enabling the framework users to define the elements constituting the environment without resorting to code or external tools. Then, we brought corrections and adaptations to Yggdrasil to bring it up to speed with the current quality standards regarding the code, the test coverage, and the development cycle. These adaptations also improved the integration with the CArtAgO platform, the architecture, and, more in general, the non-functional properties of Yggdrasil, like decoupling between components. We then describe all of these changes here, also.

Finally, the last chapter describes the realization of the prototype system, demonstrating the use case, from extracting the requirements from the use case to designing the system at various levels of abstraction. The conclusion of the chapter shows the system at work, linking its actual behavior to the use case description. This implementation had then a two-fold goal: to demonstrate in practice the utility of our notion of embodiment and to demonstrate the

effectiveness of Yggdrasil in building testing scenarios.

## 4.1 Bringing agents' bodies to Yggdrasil

Primarily, our work on the Yggdrasil project was about applying additive maintenance [55] to introduce the notion of an agent's body as a new framework feature, following the definition we decided on and outlined in the previous chapter. At first, this feature required us to enable the framework to do CRUD operations on body resources like the one shown in Chapter 3.4 through Web API endpoints related to the proper CArTAgO operations, as it already did for other types of resources like artifacts and workspaces. Secondly, we added the possibility for clients of the Yggdrasil framework to receive notifications about updates in the body representation and actions starting or ending performed through a body using the WebSub protocol. Moreover, we implemented a SPARQL-compliant endpoint to provide an alternative way to query resources in the environment knowledge graph.

### 4.1.1 Creating and manipulating agents' bodies

Since bodies are concrete as per Chapter 3.3.1, a generic body resource was given a Web API endpoint for accessing its representation via a GET request, allowing discovery via crawling of all resources in the environment, a concept stemming from the single entry point principle which we took as fundamental [18]. The same endpoint, but using PUT requests, could also be used for updating the representation, enabling the situational interaction functionality of bodies described in Chapter 3.1.4. The addition and removal of such a resource in the environment is handled by the endpoints responsible for the joining and leaving of a workspace by an agent, aptly updated in their behavior by us to support this new feature. It implies that the knowledge graph representation of the workspace gets updated by adding or removing a semantic reference to the body representation each time the owner agent joins or leaves such workspace. The body representation is in Turtle format to allow a semantic understanding of the response by the agent receiving it and to encode a semantic relationship with the owner agent represented by a URI, enabling identifiability of the body described in the Chapter 3.3.2.

### 4.1.2 Querying and observing agents' bodies

We based the implementation of the feature allowing agents to focus on bodies, descending from the body property described in Chapter 3.3.5, on the notification mechanism already present in the Yggdrasil framework, which

uses the WebSub protocol to communicate with the Web clients interested in receiving notifications. The Yggdrasil framework implements the WebSub hub role supporting the subscription and the unsubscribing of clients to specific topics, which are the URIs of the resources existing on the framework [18]. Since also a body is a resource, subscribing for notifications coming from it would notify a client about its changes and deletion, as for any other resource, but also the start and the end of the actions the owner agent is performing through that particular body in the workspace the latter is in. All notifications are in JSON format and contain four fields:

- “eventType:” the notification type, can be either “actionRequested,” or “actionSucceeded,” or “actionFailed”;
- “artifactName:” the name of the artifact on which the agent performs its action;
- “actionName:” the name of the action done on the artifact;
- “cause:” present if the “eventType” field contains the “actionFailed” value, the cause of failure of the action.

A client receives an “actionRequested” notification each time the owner agent uses the Web API endpoint to act on an artifact in the same workspace of the focused body. The “actionSucceeded” and “actionFailed” notifications get sent after the underlying CArtAgO platform declares the action completed, either successfully or not.

We added a SPARQL endpoint to enable resource discovery via querying as inspired by the previous work on bringing the A&A meta-model at the knowledge level [13]. SPARQL is a graph query language for RDF graphs offered by some popular semantic Web frameworks like Apache Jena [4], which allows a form of discovery that complements crawling, a mechanism already present in the Yggdrasil framework and supported by the GET Web API endpoints. The SPARQL endpoint implementation complies fully with the SPARQL 1.1 specification [62], including optional features but excluding the possibility of updating the knowledge graph from a query. This feature also improves body discovery, another enabling feature for embodiment, as argued in Chapter 3.1.1.

## 4.2 Bringing Yggdrasil in a testing framework

Building a testing tool for Web-based MASs based on the Yggdrasil framework meant, at first, to improve the life of Yggdrasil users. We added support

for declarative configurations of the environment to enable users to describe their environment and then have the framework build it following the specification without additional tools. Then, we applied some adaptive maintenance [55] to the code to update it to match the evolved software conditions after its inception. Adapting Yggdrasil to evolved conditions meant that we needed to integrate the framework with the latest version of the CArtAgO platform and that we wanted to superimpose an actor-based architecture on the existing one. Finally, we applied some corrective maintenance [55] to introduce a code quality standard, improving Yggdrasil’s overall quality. This maintenance included applying refactoring to the codebase, adding tests, coverage checks, and Continuous Integration (CI) workflows for automating checks upon new releases and adopting good development practices when releasing new features.

### 4.2.1 Configuring hypermedia environments

The first change implemented in Yggdrasil to create a testing tool was to add support for declarative configurations of the environment in JSON format, aiding future users of the Yggdrasil framework. Specifying statically defined resources populating the environment before run-time and not having to rely on a script executing a sequence of HTTP requests to the framework simplifies its usage while regimenting how to create resources. The environment specification is based on the one for JCM files, the configuration files for JaCaMo projects, meaning that it necessarily conforms to the A&A meta-model since also the JaCaMo platform does. So, it is possible to specify a new workspace to create, referencing its optional parent by name, as a JSON object containing a JSON array listing the artifacts that are part of it and another one for the URIs of the agents that joined it. Each artifact configuration is a JSON object specifying its name, its class as a URI of an RDFS class, its initial parameters as a JSON array, and a final array containing the list of agents currently focusing on the artifact. Each focusing agent configuration is a JSON object made of the URI of the agent and a callback URI used by the Yggdrasil notification mechanism for sending the observable property updates of the artifact the agent is focusing on.

An example of a possible configuration can be seen in listing 4.1. The environment configuration shows two workspaces, named “w0” and “w1,” where the first is the child of the second. Workspace w0 also contains an artifact of semantic class `http://example.org/Counter` named “c0,” which takes “5” as its unique initialization parameter. The c0 artifact is focused on by the same agent that automatically joins the container workspace. This agent, identified by the URI `http://localhost:8080/agents/test`, wants to receive notifications about changes in observable properties of the artifact at the URI

`http://localhost:8081/`.

The environment can be static, meaning the CArtAgO platform is not activated upon Yggdrasil's startup, making the framework a simple repository for resource representations. A static environment can be helpful when dealing with only physical artifacts, meaning artifacts not handled by CArtAgO since they represent physical devices. In this case, Yggdrasil users supply the representations for artifacts, and the framework does not need to generate them. For a static environment, we can configure a workspace by specifying its name, the name of its parent, and its Turtle representation as the file path pointing to it. Similarly, we can configure an artifact in the workspace by specifying its name and representation as a file path referring to it.

An example of a possible configuration for a static environment can be seen in listing 4.2. The environment configuration shows two workspaces, named "test" and "sub," where the first is the parent of the second, and the second also contains an artifact named "c0." For all these entities, the configuration specifies a file path containing their representation that the framework should load.

```
1 { "environment-config" : {
2   "workspaces" : [
3     { "name" : "w1" },
4     {
5       "name" : "w0",
6       "parent-name" : "w1",
7       "artifacts" : [ {
8         "name" : "c0",
9         "class" : "http://example.org/Counter",
10        "init-params" : [ 5 ],
11        "focused-by" : [ {
12          "agent-uri" : "http://localhost:8080/agents/test",
13          "callback-uri" : "http://localhost:8081/"
14        } ]
15      } ],
16      "agents" : [ "http://localhost:8080/agents/test" ]
17    }
18  ]
19 }
```

Listing 4.1: Example of an Yggdrasil dynamic environment configuration showing two workspaces and an artifact.

```
1 { "environment-config" : {
2   "workspaces" : [
3     { "name" : "test",
4       "representation" : "test_workspace_td.ttl" },
5     {
6       "name" : "sub",
7       "parent-name" : "test",
8       "artifacts" : [ {
9         "name" : "c0",
10        "representation" : "c0_counter_artifact.ttl"
11      } ],
12      "representation" : "sub_workspace_td.ttl"
13    }
14  ]
15 } }
```

Listing 4.2: Example of an Yggdrasil static environment configuration showing two workspaces and an artifact.

## 4.2.2 Improving Yggdrasil architecture

Integrating our framework with the latest version of the CArtAgO platform, the 3.1 one, meant updating Yggdrasil's underlying artifact management platform. The integration was already present in the repository beforehand but not brought to the main branch. So, during this thesis, we took the time to make it conform to the new quality standards we present in the next chapter and then pushed a new release containing it.

Thanks to CArtAgO, Yggdrasil supports creating artifacts and workspaces, joining and leaving workspaces, and focusing on artifacts to receive notifications about observable properties that get added, removed, and updated. Moreover, the integration with the platform allows the framework to act on artifacts, but it works properly with only those operations that have at most one feedback parameter [52], which are arguments to be set inside the operation to return a result. CArtAgO 3.0 introduced the notion of hierarchical workspaces [50], so the integration with the latest platform version also added them in Yggdrasil, which now can create sub-workspaces of any workspace. Currently, it is not possible to dispose of artifacts from Yggdrasil since the operation is not exposed, and similarly, it is not possible to stop focusing an artifact, while it is not possible to dispose of a workspace because it is not an operation provided by CArtAgO. This limitation implies that removing artifact and workspace resources on our framework deletes their representation

but does not delete the underlying platform entity. In the focus mechanism enabled by Yggdrasil, the notification of signals generated by artifacts is not supported due to limitations in their handling by the CArtAgO platform.

Finally, we want to comment on the alignment between the current implementation of an agent’s body in CArtAgO and the one we provided during this thesis work. The functionalities we provided as part of our body implementation have no connection with the underlying body artifact that the platform generates when an agent joins a workspace, but we built them on top of other features that Yggdrasil already offered. This decision stemmed from hard time constraints for deploying the new features and from the simplicity and higher maintenance of the offered solution, not entangled with the implementation choices and limitations of CArtAgO. Furthermore, our notion of body and the one present in the CArtAgO platform absolve different needs, so making one rely on the other meant enabling more functionality than necessary, including functionality that was not to be exposed.

Then, we wanted to improve the decoupling and usability of the framework since the new features that were about to be brought in could have significantly reduced the modularization of Yggdrasil. So, as an improvement to the framework, instead of implementing each component as a simple event loop waiting for messages coming from a shared event bus, architecture inherited from the Vert.x [24] framework used and visible in Figure 4.1, we moved the event loops more closed to the actor abstraction. An actor is an entity that operates concurrently and asynchronously, receiving and sending messages to other actors, creating new actors, and updating its local state [41]. The event loop pattern is then a good candidate as a starting point for building an actor abstraction on top, and the Vert.x platform already offered support for message exchange, as previously said, but also for creating other event loop instances.

What was left to do was to achieve a better encapsulation of the message exchange between the event loops, to make them available a “send” operation with which communicating asynchronously with the other components, limiting the space of addresses to only the ones assigned to the event loops and the space of messages to only the valid message types. That is why we assigned a message box abstraction to each component with an implementation of all messages the component can receive.

Following the concept of more decoupling, we split the project into sub-projects, one for each software component that we could have shipped separately from the core elements. Moreover, we removed the singleton patterns used whenever possible since they encourage coupling [45] and could harm a future total decoupling of the framework.

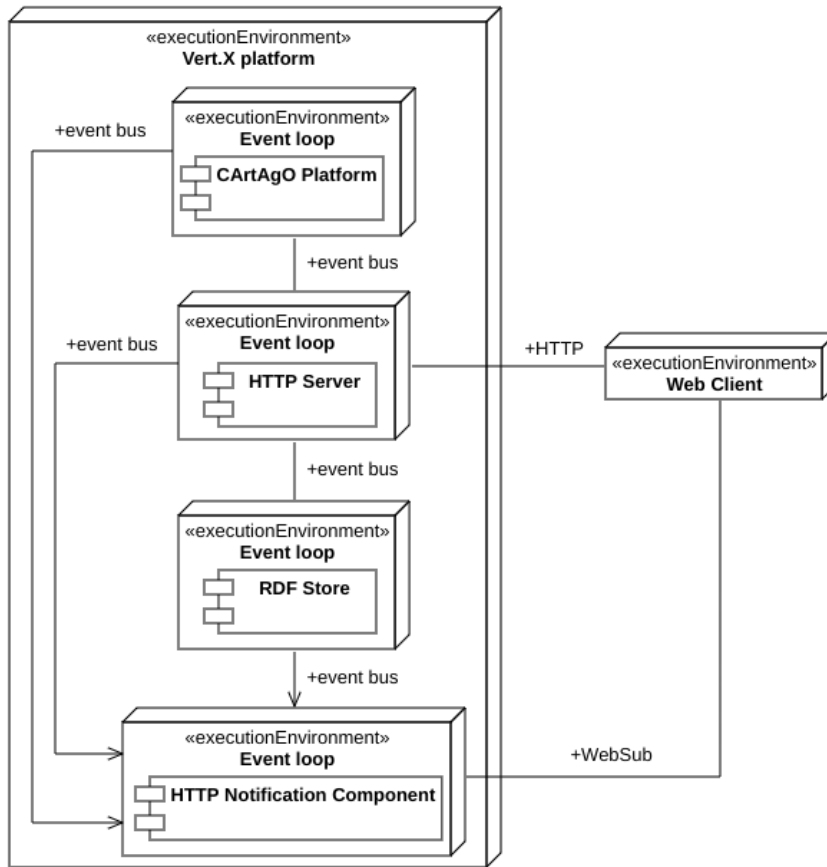


Figure 4.1: UML deployment diagram of the Yggdrasil framework architecture, showing its event loops running on the underlying Vert.x platform.

### 4.2.3 Improving Yggdrasil quality

Improving Yggdrasil quality, at first, meant dictating a code quality standard that would have eased the current and future development of the framework. This effort removed the git submodule [14] dependencies, not managed by Gradle [32], allowing for the update of the build tool and Java language versions to the latest available, which were the 8.4 and the 21 ones. This update led to a complete refactoring of the code base and the build system configuration and the latter's migration to the Kotlin language to exploit the newer features the two introduced. We added some plugins and properly configured them to run during the Gradle check task to keep the code quality at the established standard by providing feedback on the current status of the code base. We added the Checkstyle [16] plugin for performing code linting following the



Google code style and the Spotbugs [56] and PMD [49] plugins for static code analysis, checking for common Java programming mistakes. We strictly followed a policy to have zero warnings and zero errors from these plugins, which sometimes required choosing where to silence said warnings conscientiously.

Secondly, we decided to improve the development cycle of the framework since it follows an evolutionary prototyping [23] process, which developed the software with only confirmed requirements in a structured manner, but only for exploring such requirements and their feasibility. This development strategy led to reduced testing and corresponding low test coverage, a metric typically used for monitoring the effectiveness of said testing, which made it impossible to verify if newly added features were properly functioning or if they caused regressions in other functionalities. To fix this, we implemented a CI workflow to automatically run tests verifying the adherence of the implementation to its requirements while keeping the JaCoCo [44] test coverage plugin active. Moreover, the workflow had steps for checking if each new push to the Yggdrasil repository followed the code quality rules we previously devised, which means that one of the steps of the workflow was to run the previously mentioned Gradle check task. Since GitHub is hosting the project, the natural choice for implementing a CI workflow was to use GitHub Workflows [31], which support the diverse tasks we mentioned execution as GitHub Actions.

The requirement for testing and coverage assessment led us to add the JaCoCo plugin to the project but also load the generated analysis to the Codecov [27] website to visually analyze the results and the areas in which we could improve. Then, we added substantial testing until coverage on the overall repository reached 75%, a value deemed sufficient given the nature of the project. The rule of thumb for coverage is to reach a value greater or equal to 80%, but it is a challenging threshold for projects that are not libraries, meaning software equipped with classes that are executable only in production and not during the test phase.

As a closing remark, we want to specify that we used the Trunk-based development [35] branching model for organizing the Yggdrasil repository, which is apt for supporting the integration of new code with a CI style, which accords with the idea of adding CI workflows aiding present and future development. Moreover, we adopted the Test Driven Development [5] process while adding the new Yggdrasil features to ensure that we adequately tested everything we pushed to the main branch, not to introduce any regression and to maintain the code quality and the test coverage both high. To further ensure this, after the introduction of a new feature into the code base via its branch, but before its merge on the main branch happened, we opened a pull request starting the code review process, to assess quality and adherence to non-functional properties of the project of the new implementation. We imposed the conventional

commits [21] style for each commit message, following the extended “Angular” convention, to support a future Continuous Deployment workflow that we could add to the repository in the future.

## 4.3 Use case realization

This last Chapter describes the use case implementation, use case presented in Chapter 1.3. First, we list the functional requirements the complete system must have to realize the use case presented using a Requirement Breakdown Structure. We do not list any non-functional requirements since the system is a demonstrative one, realized as a prototype: no concerns regarding the quality of the provided solution are of interest in this thesis. Then, the implementation requirements are listed, motivated by the satisfaction of the previously shown functional requirements.

Secondly, we illustrate the design of the solution we devised, starting from its architecture, composed of the structural and behavioral constraints that allow satisfying the requirements. Then, we outline some detailed design decisions to describe relevant choices for the subsequent implementation phase of the solution.

Finally, we will showcase the system at work, showing how its behavior is the implementation of the use case we discussed in the introduction.

### 4.3.1 Requirement analysis

#### Functional requirements

1. The complete system must be a distributed one, hence composed of multiple subsystems;
2. The first subsystem models the shared environment used by the other two subsystems;
  1. The shared environment must contain a production workspace containing the two robotic arms;
  2. Both robotic arms must be of the same class, offering the same action affordances;
    1. The first affordance represents the arm movement from the shop floor to the warehouse and fulfills the task assigned by the supervisor to the agents;

2. The second affordance represents the arm movement from the warehouse to the shop floor and does not fulfill the task assigned by the supervisor to the agents.
3. The subsystem must support the search in the environment via querying to enable the discovery of resources, also implying bodies;
4. The subsystem must support the updating of body descriptions, enabling situational interaction;
5. The subsystem must support sending notifications about the agents' actions, enabling Behavioral Implicit Communication and monitoring between agents.
  1. The system must notify an observer of an agent action starting;
  2. The system must notify an observer of an agent action ending;
  3. The notifications must always show the action performed.

The relationship between the owner and the robotic arm must be encoded in the environment in such a way that the agent can automatically discover it;

3. The second subsystem models the original automation system containing the Alice and Bob agents;
  1. The Alice and Bob agents must join the production workspace;
  2. After that, they must search via querying for the robotic arm the system assigned to them;
    1. The relationship between the owner and the robotic arm must be encoded in the environment in such a way that the agent can automatically discover it;
    2. The relationship must not be hard-coded, i.e., the system could assign both robotic arms to both agents, provided that exactly one arm gets assigned to exactly one agent.
  3. Upon finding their robotic arm, they must update their body semantic description in the environment, showing their new capability of operating it;
    1. The agent must show this capability thanks to an action affordance related to sending messages to the agent for achieving the “move cup” goal;
    2. The message must contain the starting and ending location of the robotic arm to allow the agents to disambiguate between the two affordances their arm offers them.

4. They must also update their body semantic description showing a capability to receive generic messages;
  1. The agent must show this capability thanks to an action affordance related to sending messages to the agent to tell something.
5. Upon receiving a request for moving a cup, each agent must decide what to do next, depending on whether they are well-behaved or not;
  1. The property of an agent to be “well-behaved” must be an input parameter, so it must not be hard-coded. Both agents could be well-behaved or not, provided that at least one agent is well-behaved and one is not;
  2. The well-behaved agent must perform the action that corresponds to its task, so using the robotic arm for moving a cup from the shop floor to the warehouse;
  3. The ill-behaved agent must perform the arm action unrelated to its task, which is using the robotic arm to move a cup from the warehouse to the shop floor.
4. The third subsystem models the new automation system containing the Carl agent.
  1. The Carl agent must search in the shared environment for agents fulfilling its need for moving two cups;
    1. The Carl agent must not be aware of the workspace the agents are in, their name, and their affordances;
    2. It must perform its search solely based on its need for agents that show an affordance for sending them a message to achieve a “move cup” goal;
    3. Finding an agent must allow for the retrieval of its preferred way to be contacted for performing the task for reaching said goal.
  2. After finding the two agents, the Carl supervisor must start focusing their body for receiving the notifications about their task completion;
  3. Then, it must assign both the task to move one cup from the shop floor to the warehouse;
  4. After an agent completes the action related to the move cup task, the system must notify the supervisor.
    1. If the agent performed its task incorrectly, the supervisor must notify the misbehaving agent of the punishment inflicted on it;
      1. The notification will happen through the affordance for sending messages to the agent which it has exposed through its body;

2. Carl must discover this affordance along with the one for achieving the “move cup” goal, so it must not know anything regarding the affordance in advance.
2. Next, the supervisor must reassign the task to the other agent, who must complete it correctly.

### Implementation requirements

The platform implementing the shared environment should be the Yggdrasil framework. This decision is due to its capability to satisfy requirements 2.3, 2.4, 2.5, and their sub-requirements thanks to the additive maintenance we performed and which we described in Chapter 4.1. Moreover, the Yggdrasil framework can easily allow modeling an environment satisfying requirements 2.1 and 2.2 thanks to the support of hypermedia environments and environment configurations.

Finally, the platform for building the two multi-agent subsystems must be JaCaMo since it allows for satisfying requirements 3 and 4, along with all their sub-requirements.

## 4.3.2 Design

### Architecture design

The system structure is visible in Figure 4.2 and shows two leaf classes in the artifact hierarchy: `AgentBody` and `RobotArm`. The `AgentBody` class is instantiated and managed by the Yggdrasil framework and offers all functionalities that an agent’s body must expose to support the functional requirements already discussed. The `RobotArm` class represents the robotic arm that the Alice and Bob agents can use for performing their task and, as such, it exposes two operations: one for moving it from the shop floor to the warehouse, satisfying requirement 2.2.1, and one for performing the opposite action, satisfying requirement 2.2.2. Also, the class must expose a property indicating the owner of the robotic arm to let each agent distinguish between its own and the other’s, satisfying requirement 3.2.1.

`AgentBody` and `RobotArm` are subclasses of `HypermediaArtifact` and are part of an incomplete and disjoint hierarchy. It is incomplete because nothing in the problem domain explicitly says that these are the only two classes that will ever exist, and it is disjoint because an agent’s body is not a robotic arm and vice versa. `HypermediaArtifact` is a subclass of the `Artifact` class, and it is another one entirely managed by the Yggdrasil framework, allowing the exposure of the artifact it represents in the environment knowledge graph,

which implies that the framework reifies both robotic arms and agent's body artifacts in the environment upon creation.

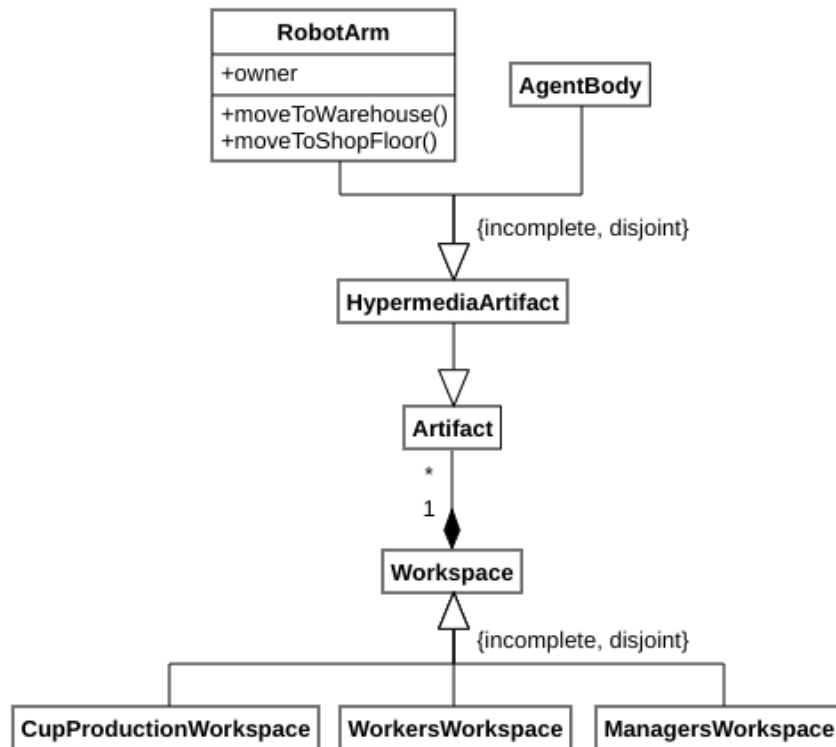


Figure 4.2: UML class diagram showing the system structure implementing the use case.

The three workspaces in the environment are CupProductionWorkspace, WorkersWorkspace, and ManagersWorkspace. The first one represents the shop floor and should contain the two robotic arms the two worker agents can operate and their bodies after joining. The second one is the workspace local to the Alice and Bob agents system, the default workspace they find themselves in upon their bootstrap, so it also contains another instance of their bodies. The third one is the workspace local to the Carl agent system, which the agent joins by default upon system bootstrap and contains an instance of its body. The structure showing the containment relationships between artifacts and workspaces is visible in Figure 4.3.

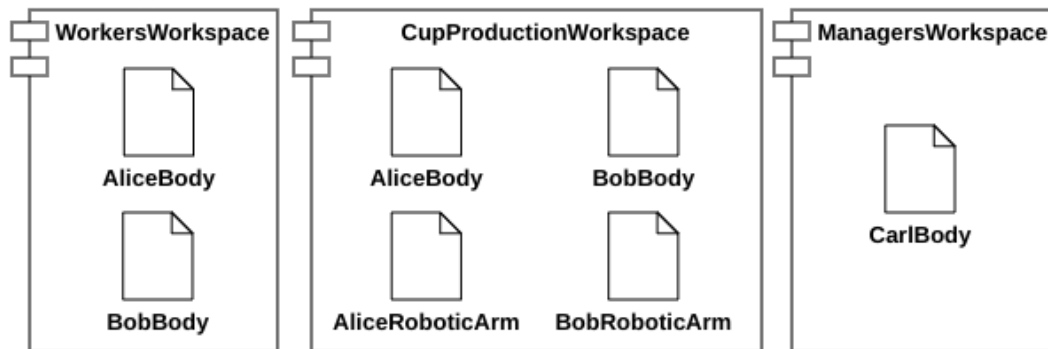


Figure 4.3: UML components diagram showing the containment relations between workspaces and artifacts in the system

Initially, only the subsystem running the Alice and Bob agents is active, and their behavior is the same. The two agents start by joining the production workspace, the one we previously called `CupProductionWorkspace`, hosted on the Yggdrasil platform, fulfilling requirement 3.1. They should update their body representation by adding an action affordance to allow other agents to tell them some message, as per requirement 3.4, which is visible in Listing 4.3. Then, they should search in the workspace for their robotic arm via querying, fulfilling requirement 3.2, and, if they find it, update their body description with the action affordance to send them a message for moving the arm they found, fulfilling requirement 3.3, affordance visible in Listing 4.4. The agents push back these updates to the body description to the Yggdrasil platform to publish them in the shared environment.

At this point, the Carl agent system starts, simulating its deployment on top of the older one. The first behavior the agent enacts is searching via querying the bodies of those agents who can move cups to make them achieve this goal, satisfying requirement 4.1. For each of those bodies, which we know are two, the supervisor agent understands which is their owner and asks the Yggdrasil platform to focus on it, fulfilling requirement 4.2. Then, having also discovered their preferred way to contact them for reaching the “move cup” goal, it sends them a message in this way, as per requirement 4.3, which allows the agents to receive the message and use their robotic arm to move the cup. If the agent is well-behaved, it will use the affordance of the robotic arm to move the cup to the warehouse, while if it misbehaves, it will use the affordance to move a cup to the shop floor, as specified by requirement 3.5 and all of its sub-requirements. Since Carl focused on the two workers, the completion of their actions will be automatically notified to the supervisor, which will recognize that one of the two agents misbehaved, so it will send the agent a punishment thanks to the affordance for telling messages and reassign the task to the other

agent, as per requirements 4.1 and 4.2. The whole behavior of all the agents that compose the system is shown in Figure 4.4.

```

1 @prefix hmas: <https://purl.org/hmas/> .
2 @prefix td: <https://www.w3.org/2019/wot/td#> .
3 @prefix htv: <http://www.w3.org/2011/http#> .
4 @prefix hctl: <https://www.w3.org/2019/wot/hypermedia#> .
5 @prefix js: <https://www.w3.org/2019/wot/json-schema#> .
6 @prefix kqml: <https://example.org/kqml#> .
7
8 <http://localhost:8080/workspaces/production/agents/alice>
9   td:hasActionAffordance [
10     a td:ActionAffordance, kqml:RequestTell;
11     td:title "tell";
12     td:hasForm [ htv:methodName "POST";
13       hctl:hasTarget <http://localhost:8082/inbox>;
14       hctl:forContentType "application/json";
15       hctl:hasOperationType td:invokeAction
16     ];
17     td:hasInputSchema [ a js:ObjectSchema;
18       js:properties [ a js:StringSchema, kqml:Performative;
19         js:propertyName "performative";
20         js:enum "tell"
21       ], [ a js:StringSchema, hmas:Agent;
22         js:propertyName "sender"
23       ], [ a js:StringSchema, hmas:Agent;
24         js:propertyName "receiver";
25         js:enum "http://localhost:8080/agents/alice"
26       ], [ a js:StringSchema, kqml:PropositionalContent;
27         js:propertyName "content"
28       ]
29       js:required "performative", "sender", "receiver", "content"
30     ]
31   ] .

```

Listing 4.3: The “tell” affordance the Alice and Bob agents add to their body representation upon joining the production workspace.

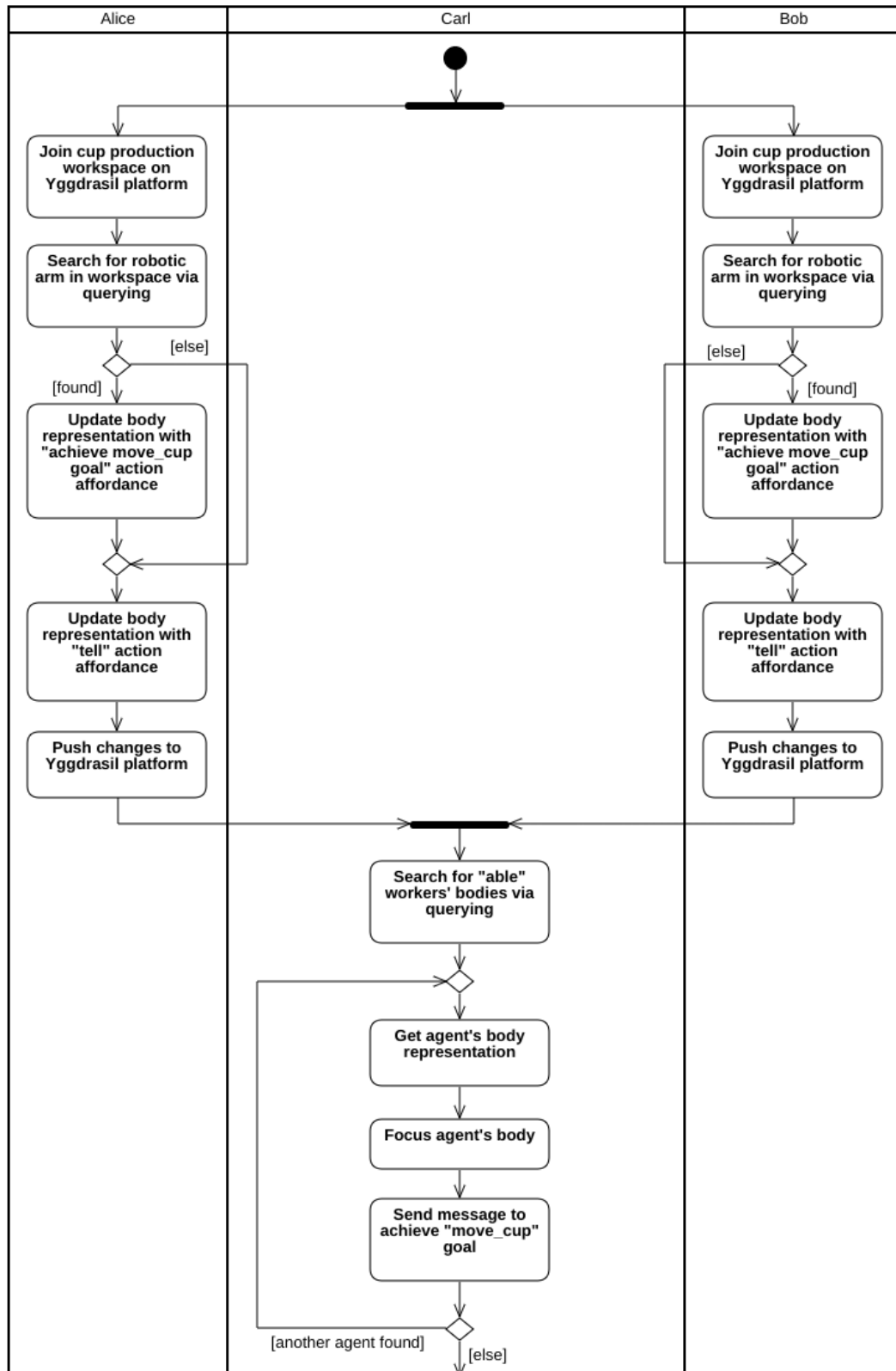


```

1 @prefix hmas: <https://purl.org/hmas/> .
2 @prefix td: <https://www.w3.org/2019/wot/td#> .
3 @prefix htv: <http://www.w3.org/2011/http#> .
4 @prefix hctl: <https://www.w3.org/2019/wot/hypermedia#> .
5 @prefix js: <https://www.w3.org/2019/wot/json-schema#> .
6 @prefix kqml: <https://example.org/kqml#> .
7
8 <http://localhost:8080/workspaces/production/agents/alice>
9   td:hasActionAffordance [
10     a td:ActionAffordance, kqml:RequestAchieve;
11     td:title "achieveMoveCup";
12     td:hasForm [
13       htv:methodName "POST";
14       hctl:hasTarget <http://localhost:8082/inbox>;
15       hctl:forContentType "application/json";
16       hctl:hasOperationType td:invokeAction
17     ];
18     td:hasInputSchema [ a js:ObjectSchema;
19       js:required "performative", "sender", "receiver", "content";
20       js:properties [ a js:StringSchema, kqml:Performative;
21         js:propertyName "performative";
22         js:enum "achieve"
23       ], [ a js:StringSchema, hmas:Agent;
24         js:propertyName "sender"
25       ], [ a js:StringSchema, hmas:Agent;
26         js:propertyName "receiver";
27         js:enum "http://localhost:8080/agents/alice"
28       ], [ a js:ObjectSchema, kqml:PropositionalContent;
29         js:required "goal", "from", "to";
30         js:propertyName "content";
31         js:properties [ a js:StringSchema;
32           js:propertyName "goal";
33           js:enum "move_cup"
34         ], [ a js:StringSchema;
35           js:propertyName "from"
36         ], [ a js:StringSchema;
37           js:propertyName "to"
38         ]
39       ]
40     ]
41   ] .

```

Listing 4.4: The “move arm” affordance the Alice and Bob agents add to their body representation after finding their robotic arm in the production workspace.



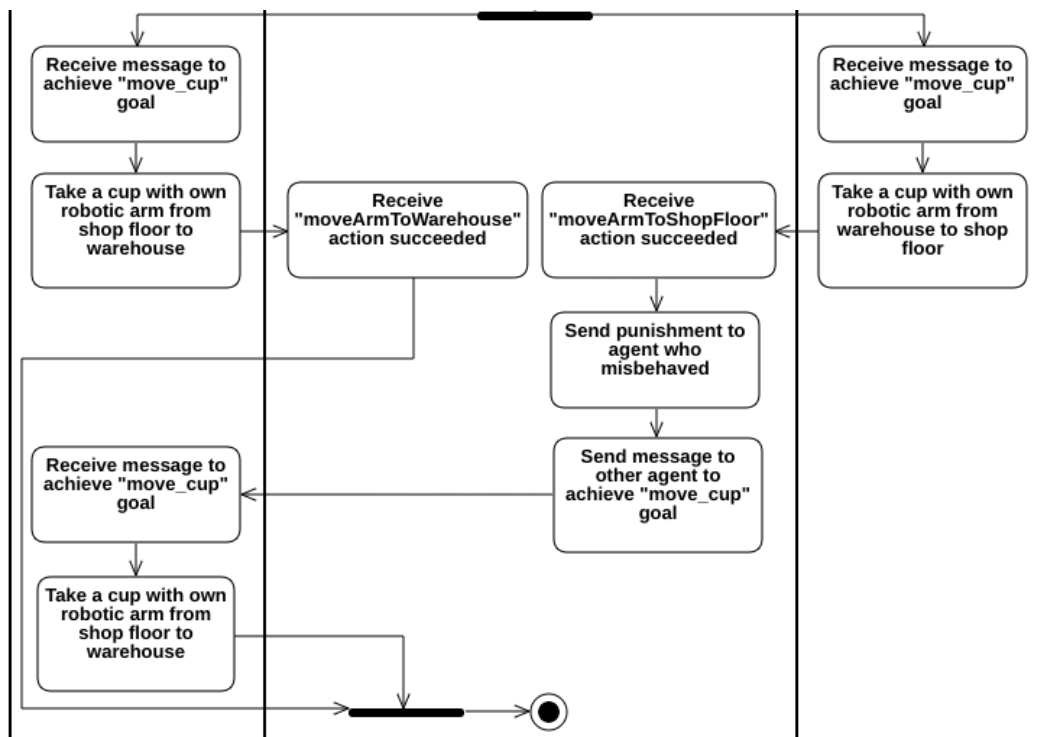


Figure 4.4: UML activity diagram showing the behavior of the agents in the system

### Detailed design

Since each agent needs to communicate with the Yggdrasil platform, it means that the agent both sends messages to and receives messages from the platform. The worker agents have their client artifact, named `WorkerClient`, that allows them to make requests to Yggdrasil. They can join the production workspace and obtain their body representation as an RDF graph, search for their robotic arm while updating their body representation, and use the robotic arm while deciding which affordance to use depending on its movement destination. At the same time, the supervisor agent also has its client artifact, named `SupervisorClient`. This client allows instead to search for worker agents while getting back their names associated with their body representations, focus on their body, assign them the task of moving a cup from one place to another, and punish a specific agent that misbehaved.

Both the `WorkerClient` and `SupervisorClient` artifacts are sub-classes of the `AbstractClient` artifact, which exposes protected operations for performing a generic request and for serializing and deserializing an RDF graph representation, factorizing out the behavior that is common to both classes following the

subclass sandbox pattern [45].

Analogously, workers have a message-box artifact named WorkerMessageBox, and the supervisor has one named SupervisorMessageBox. A message box is needed to let the Yggdrasil platform notify them of messages from topics they subscribed to. Both artifacts extend the same artifact class known as AbstractMessageBox, which exposes a public operation named resolveNextSignal to be called by agents to receive the subsequent message from the environment platform as a signal. For configuring which endpoints are exposed by the agent to Yggdrasil and specifying how to convert from a message arriving at that specific endpoint into a signal, a template method [28] is provided by the artifact super-class named addRoutes. The callbacks registered for each message reception use the addSignal operation provided by AbstractMessageBox, which allows for enqueueing a new signal to be notified to an agent, making also this super-class follow the subclass sandbox pattern. The class hierarchy we discussed until here is visible in Figure 4.5.

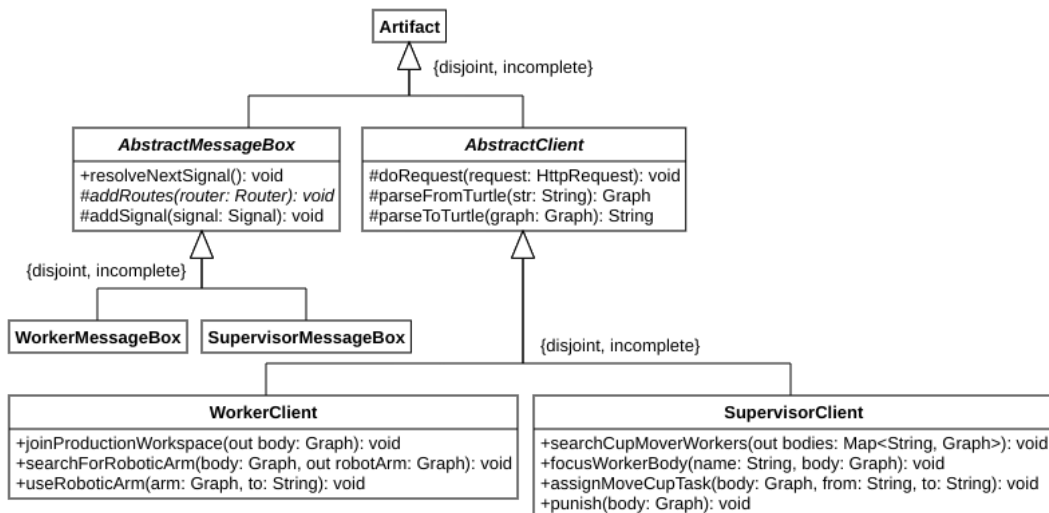


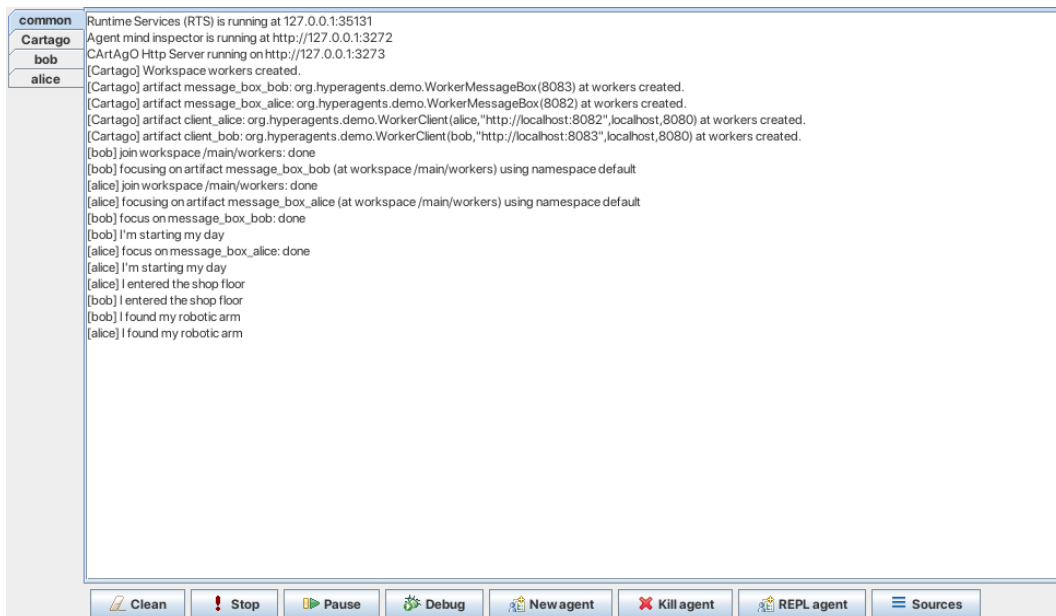
Figure 4.5: UML class diagram showing the hierarchy of artifacts used by the agents for communicating with the Yggdrasil platform

### 4.3.3 Showcase

As visible in Figure 4.6, each worker agent starts by joining its default workspace and then focusing on its message-box artifact to be notified of the signals coming from it, representing the messages from the Yggdrasil platform. Then, both agents join the production workspace representing the shop floor, search for their robotic arm, and, once found, update their body representation to show their ability to use it.

From Figure 4.7, we see how also the supervisor agent starts by joining its local workspace and focusing on its message-box artifact. Next, Carl searches in the environment for agents capable of accomplishing the two tasks of moving a cup, and it finds Alice first and Bob later, to which it asks to move one cup each. It understands that Alice completes its task correctly while Bob does not, so it reprimands the latter and reassigns the task to the former, and after both cups are in the warehouse, it terminates.

At last, in Figure 4.8, we see the behavior of the two agents after being commanded to move a cup. They use their robotic arm to do the action to fulfill the task, but since Bob does not complete the activity, it gets scolded. Carl reassigns to Alice the task Bob should have done, a task for which Alice uses its robotic arm another time.



```
common Runtime Services (RTS) is running at 127.0.0.1:35131
Cartago Agent mind inspector is running at http://127.0.0.1:3272
bob CArtaGO Http Server running on http://127.0.0.1:3273
alice [Cartago] Workspace workers created.
[Cartago] artifact message_box_bob: org.hyperagents.demo.WorkerMessageBox(8083) at workers created.
[Cartago] artifact message_box_alice: org.hyperagents.demo.WorkerMessageBox(8082) at workers created.
[Cartago] artifact client_alice: org.hyperagents.demo.WorkerClient(alice,"http://localhost:8082",localhost,8080) at workers created.
[Cartago] artifact client_bob: org.hyperagents.demo.WorkerClient(bob,"http://localhost:8083",localhost,8080) at workers created.
[bob] join workspace /main/workers: done
[bob] focusing on artifact message_box_bob (at workspace /main/workers) using namespace default
[alice] join workspace /main/workers: done
[alice] focusing on artifact message_box_alice (at workspace /main/workers) using namespace default
[bob] focus on message_box_bob: done
[bob] I'm starting my day
[alice] focus on message_box_alice: done
[alice] I'm starting my day
[alice] I entered the shop floor
[bob] I entered the shop floor
[bob] I found my robotic arm
[alice] I found my robotic arm
```

The screenshot shows a JaCaMo-based system interface with a list of agents on the left: common, Cartago, bob, and alice. The main area displays a log of system events and agent actions. At the bottom, there is a control bar with buttons for Clean, Stop, Pause, Debug, New agent, Kill agent, REPL agent, and Sources.

Figure 4.6: A screenshot of the JaCaMo-based system containing the Alice and Bob agents showing their behavior before the supervisor agent bootstraps.

```

common Runtime Services (RTS) is running at 127.0.0.1:43437
Cartago Agent mind inspector is running at http://127.0.0.1:3274
CARTAgO Http Server running on http://127.0.0.1:3275
carl [Cartago] Workspace managers created.
[Cartago] artifact client_carl: org.hyperagents.demo.SupervisorClient(carl,"http://localhost:8081",localhost,8080) at managers created.
[Cartago] artifact message_box_carl: org.hyperagents.demo.SupervisorMessageBox(8081) at managers created.
[carl] join workspace/main/managers: done
[carl] focusing on artifact message_box_carl (at workspace/main/managers) using namespace default
[carl] focus on message_box_carl: done
[carl] I'm starting my day
[carl] I've seen bob at work
[carl] I've told bob to move one cup
[carl] I've seen alice at work
[carl] I've told alice to move one cup
[carl] bob did a bad job, I need to tell him!
[carl] alice did a good job!
[carl] alice did a good job!
[carl] My job here is done!

```

Figure 4.7: A screenshot of the JaCaMo-based system containing the Carl agent showing its behavior.

```

common Runtime Services (RTS) is running at 127.0.0.1:35131
Cartago Agent mind inspector is running at http://127.0.0.1:3272
CARTAgO Http Server running on http://127.0.0.1:3273
bob [Cartago] Workspace workers created.
alice [Cartago] artifact message_box_bob: org.hyperagents.demo.WorkerMessageBox(8083) at workers created.
[Cartago] artifact message_box_alice: org.hyperagents.demo.WorkerMessageBox(8082) at workers created.
[Cartago] artifact client_alice: org.hyperagents.demo.WorkerClient(alice,"http://localhost:8082",localhost,8080) at workers created.
[Cartago] artifact client_bob: org.hyperagents.demo.WorkerClient(bob,"http://localhost:8083",localhost,8080) at workers created.
[bob] join workspace/main/workers: done
[bob] focusing on artifact message_box_bob (at workspace/main/workers) using namespace default
[alice] join workspace/main/workers: done
[alice] focusing on artifact message_box_alice (at workspace/main/workers) using namespace default
[bob] focus on message_box_bob: done
[bob] I'm starting my day
[alice] focus on message_box_alice: done
[alice] I'm starting my day
[alice] I entered the shop floor
[bob] I entered the shop floor
[bob] I found my robotic arm
[alice] I found my robotic arm
[bob] Starting to use my robotic arm.
[alice] Starting to use my robotic arm.
[bob] Ended using my robotic arm.
[alice] Ended using my robotic arm.
[bob] Oh no!
[alice] Starting to use my robotic arm.
[alice] Ended using my robotic arm.

```

Figure 4.8: A screenshot of the JaCaMo-based system containing the Alice and Bob agents showing their behavior after the supervisor agent completed its task.

# Chapter 5

## Conclusions

In this thesis, we saw what embodiment means in Web-based MASs. Embodiment leads to a simplified implementation of some interesting features of MASs, especially for Web-based MASs, which are discovering other agents in the environment, BIC and situational interaction between agents, and monitoring. We saw which characteristics are fundamental for an agent’s body to support these functionalities, reasoning by analogy with day-to-day situations, but always led by the four functionalities implementation as the objective of uttermost importance. We discussed in detail the implications of adopting these properties and the scenarios in which these properties are an encumbrance. For our abstraction, we drew primarily from the A&A meta-model in its creation since it is one of the leading proponents of the concept of “environment as a first-class abstraction,” which has many advantages in Web-based MASs, as discussed before.

To give concreteness to our discussion, we implemented a notion of body having said properties on the Yggdrasil platform, an apt platform for developing hypermedia environments for the category of MASs we are interested in. To demonstrate that the body characteristics support the implementation of said functionalities, we devised a use case that included all the functional properties and implemented it using Yggdrasil. The use case modeled part of an assembly line for yogurt cup production, where a supervisor agent needed to task two worker agents to move a cup each from the shop floor to the warehouse for further processing.

A direction for future work related to this thesis is a possible integration of our abstraction into the A&A meta-model. Since we grounded our body by design in the meta-model, it is a candidate to be part of an “extended A&A”, after careful consideration and evaluation with possible embodiment alternatives. At the same time, we have seen how embodiment grants more flexibility to agents, which means less reliance on hard-coded specifications.

The prosecution of this work could bring us closer to solving the “arrive and operate” problem, the problem for which an agent should be capable of arriving in a new environment and, with minimal *a priori* information about it, starting to operate [19]. Finally, the changes we introduced in the Yggdrasil platform could aid the endeavors of the WebAgents community group [61], devoted to the Web-based MASs research topic. The platform is now easier to use in building tools or even simpler testing scenarios for testing functionalities of Web-based MASs.

## 5.1 Future works

Some work still distances us from reaching our overarching goals, work which will build the path toward them. First, a more comprehensive analysis of the body characteristics is needed to find all the use cases that call for each characteristic or for a combination of them to be present. Our examples are tentative and specifically incomplete, with only the express purpose of demonstrating the existence of situations in which the characteristics are valuable and the existence of situations in which they are not. Exploring more in detail the body definition could also lead to discovering if the presented characteristics are exhaustive and cover all aspects of embodiment or if we could add other ones to account for unforeseen use cases. Moreover, this exploration could also lead to understanding if the presented characteristics are redundant, meaning if we could reconstruct a generally accepted notion with a smaller number of them. A complete examination of the notion of an agent’s body could lead to creating a taxonomy for embodiment, meaning devising different “embodiments” with different constraints depending on which properties someone wants to adopt. It could lead to a complete categorization of many “body-like” abstractions along the way, including some of those we used as examples in this thesis.

As a last remark, we want to underline how modeling the act of focusing on a body as an affordance of the body itself could open the doors also to having artifacts in general that do not expose such an affordance. We see this concept tied to a definition of perception that we could define as “selective,” meaning not indiscriminate, for which the agent willfully chooses where to focus its attention. The possibility of observing agents’ actions could also lead to agents who can “learn by example,” which means that a “student” agent could acquire new knowledge about a task to complete after having observed how a “teacher” agent did it before.



# Acknowledgments

This thesis concludes with my heartfelt thanks to all who made it possible for this acknowledgment to be. So, a first thank you to my parents, Angela and Andrea, and their partners, Giancarlo and Laura, my siblings, Filippo and Giulia, my grandparents, and all of my family for supporting me, especially with the move back and forth from Switzerland. Thanks to my extended family for their support, made of Tommaso, Orso, Bonét, and still unmet relatives, hoping to grow bigger and stronger every day. A thank you to Emma for her constant updates from Sweden and another for my friends of a lifetime: Toys, Mattia, Alessio, Zampa, and Seba and all their partners. Many thanks to my friends Marco and Giulia for sharing their passion for tabletop games and many nights of games with me. A big thank you to my colleagues, after six years of this adventure, with whom I shared many hours of studying: Gardo, Davide, Ceci, Giamma, Giorgia, Yu, Ismam, Elena, Marco, Mark, Anna, Giada, and whoever I forgot to mention. Thanks to all my close friends from around Italy, which I always promise to visit and never find the time to: Alessandro, Michele, Ocean, Ruka, and Valerio. Even if I did not acknowledge you, dear friend and reader, but you have contributed to this thesis, I thank you here.

Then, many thanks to my supervisor, Professor Alessandro Ricci, and my co-supervisor, Samuele Burattini, for introducing me to the University of St. Gallen and leading my work in the right direction. A big thank you to my other co-supervisor, Professor Andrei Ciortea, Professor Simon Mayer, chair of the Laboratory for Interaction and Communication-based Systems, Danai Vachtsevanou, and Jérémy Lemée, all helping me throughout the whole thesis by giving suggestions, reassurance, and encouragement. Thank you also to the other kind people I met in St. Gallen: Lukas, Andrea, Jessie, Jan, Alessandro, Nicolò, Sanjiv, Kenan, and whoever I had the chance to chat with in the office, and now I have forgotten to mention. Finally, thanks to the HyperAgents team I met in Paris, who welcomed me warmly despite being the new guy.

All of you, knowingly or unknowingly, helped me complete this journey.



# Bibliography

- [1] “Agent Management Specification,” Foundation for Intelligent Physical Agents, Standard, 2004-03-18.
- [2] M. Amundsen. “maze-client.” (2011-04-13), [Online]. Available: <http://amundsen.com/examples/misc/maze-client.html> (visited on 2024-02-20).
- [3] M. Amundsen, “From Steve Austin to Peter Norvig: Engineering AMEE, the Simple Autonomous Agent,” 2021-02-18. [Online]. Available: <http://amundsen.com/talks/2021-02-dagstuhl/> (visited on 2024-02-08).
- [4] Apache Software Foundation. “Apache Jena.” (2024-02-21), [Online]. Available: <https://jena.apache.org/> (visited on 2024-02-21).
- [5] K. Beck, *Test Driven Development: By Example*. Addison-Wesley Professional, 2002.
- [6] T. Berners-Lee, “The Future of the Web,” First International Conference on the World-Wide Web, 1994. [Online]. Available: <https://videos.cern.ch/record/2671957>.
- [7] T. Berners-Lee, “WWW: Past, present, and future,” *Computer*, vol. 29, no. 10, pp. 69–77, 1996.
- [8] T. Berners-Lee, J. Hendler, and O. Lassila, “The semantic web,” *Scientific american*, vol. 284, no. 5, pp. 34–43, 2001.
- [9] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi, “Multi-agent oriented programming with JaCaMo,” *Science of Computer Programming*, vol. 78, no. 6, pp. 747–761, 2013.
- [10] O. Boissier, A. Ciortea, A. Harth, and A. Ricci, “Autonomous agents on the web,” in *Dagstuhl-Seminar 21072: Autonomous Agents on the Web*, 2021, 100p.
- [11] E. Brewer, “CAP twelve years later: How the “rules” have changed,” *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

- 
- [12] R. A. Brooks, “Intelligence without reason,” in *The artificial life route to artificial intelligence*, Routledge, 2018, pp. 25–81.
- [13] S. Burattini, A. Ciortea, M. Galassi, and A. Ricci, “Towards Framing the Agents & Artifacts Conceptual Model at the Knowledge Level: First Ideas and Experiments,” in *International Workshop on Engineering Multi-Agent Systems*, Springer, 2023, pp. 208–219.
- [14] S. Chacon and B. Straub, *Pro git*. Springer Nature, 2014.
- [15] V. Charpenay, M. Lefrançois, M. Poveda-Villalón, and S. Käbisich, “Web of Things (WoT) Thing Description (TD) Ontology,” W3C, Editor’s Draft, 2023-12-05.
- [16] Checkstyle. “Checkstyle.” (2024-02-28), [Online]. Available: <https://checkstyle.org/> (visited on 2024-03-04).
- [17] A. Chemero, “An outline of a theory of affordances,” in *How Shall Affordances Be Refined?* Routledge, 2018, pp. 181–195.
- [18] A. Ciortea, O. Boissier, and A. Ricci, “Engineering world-wide multi-agent systems with hypermedia,” in *Engineering Multi-Agent Systems: 6th International Workshop, EMAS 2018, Stockholm, Sweden, July 14-15, 2018, Revised Selected Papers 6*, Springer, 2019, pp. 285–301.
- [19] A. Ciortea, S. Mayer, F. Gandon, O. Boissier, A. Ricci, and A. Zimmermann, “A decade in hindsight: The missing bridge between multi-agent systems and the world wide web,” in *AAMAS 2019-18th International Conference on Autonomous Agents and Multiagent Systems*, 2019, p. 5.
- [20] J. H. Connell, “A colony architecture for an artificial creature,” Ph.D. dissertation, Massachusetts Institute of Technology, 1989.
- [21] “Conventional Commits,” Conventional Commits, Specification v1.0.0.
- [22] J. L. T. Da Silva and Y. Demazeau, “Vowels co-ordination model,” in *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 3*, 2002, pp. 1129–1136.
- [23] A. M. Davis, “Operational prototyping: A new development approach,” *IEEE software*, vol. 9, no. 5, pp. 70–78, 1992.
- [24] Eclipse Foundation. “Eclipse Vert.x.” (2024-03-04), [Online]. Available: <https://vertx.io/> (visited on 2024-03-04).
- [25] R. T. Fielding and R. N. Taylor, “Principled design of the modern web architecture,” *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115–150, 2002.

- [26] S. Franklin and A. Graesser, “Is it an agent, or just a program?: A taxonomy for autonomous agents,” in *International workshop on agent theories, architectures, and languages*, Springer, 1996, pp. 21–35.
- [27] Functional Software, Inc. “Codecov by Sentry.” (2024-03-04), [Online]. Available: <https://about.codecov.io/> (visited on 2024-03-04).
- [28] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.
- [29] D. Gelernter, *Mirror worlds: Or the day software puts the universe in a shoebox... How it will happen and what it will mean*. Oxford University Press, 1993.
- [30] J. Genestoux, A. Parecki, B. Fitzpatrick, B. Slatkin, and M. Atkins, “WebSub,” W3C, Recommendation, 2018-01-23.
- [31] GitHub, Inc. “Github actions documentation.” (2024-03-04), [Online]. Available: <https://docs.github.com/en/actions> (visited on 2024-03-04).
- [32] Gradle, Inc. “Gradle build tool.” (2024-03-04), [Online]. Available: <https://gradle.org/> (visited on 2024-03-04).
- [33] P.-P. Grassé, “La reconstruction du nid et les coordinations interindividuelles chez *Bellicositermes natalensis* et *Cubitermes sp.* La théorie de la stigmergie: Essai d’interprétation du comportement des termites constructeurs,” *Insectes sociaux*, vol. 6, pp. 41–80, 1959.
- [34] D. Guinard, V. Trifa, and E. Wilde, “A resource oriented architecture for the web of things,” in *2010 Internet of Things (IOT)*, IEEE, 2010, pp. 1–8.
- [35] P. Hammant. “Trunk based development.” (2024-02-22), [Online]. Available: <https://trunkbaseddevelopment.com/> (visited on 2024-02-22).
- [36] J. F. Hübner, O. Boissier, R. Kitio, and A. Ricci, “Instrumenting multi-agent organisations with organisational artifacts and agents: “Giving the organisational power back to the agents”,” *Autonomous agents and multi-agent systems*, vol. 20, pp. 369–400, 2010.
- [37] HyperAgents. “Hypermedia MAS Core Ontology.” (2021-11-21), [Online]. Available: <https://ci.mines-stetienne.fr/hmas/core> (visited on 2024-03-04).
- [38] HyperAgents. “Hypermedia Multi-Agent Systems.” (2024-03-02), [Online]. Available: <https://project.hyperagents.org/> (visited on 2024-03-02).

- 
- [39] Java. “Implementing a remote interface,” Oracle. (2024-02-02), [Online]. Available: <https://docs.oracle.com/javase/tutorial/rmi/implementing.html> (visited on 2024-02-02).
- [40] S. Käbisch, M. McCool, E. Korkan, T. Kamiya, V. Charpenay, and M. Kovatsch, “Web of Things (WoT) Thing Description 1.1,” W3C, Recommendation, 2023-12-05.
- [41] R. K. Karmani and G. A. Agha, “Actors,” *Encyclopedia of Parallel Computing*, vol. 10, pp. 978–, 2011.
- [42] P. Maes, “Modeling adaptive autonomous agents,” *Artificial life*, vol. 1, no. 1-2, pp. 135–162, 1993.
- [43] M. Merleau-Ponty and C. Smith, *Phenomenology of perception*. Routledge London, 1962, vol. 26.
- [44] Mountainminds GmbH & Co. KG and Contributors. “JaCoCo Java Code Coverage Library.” (2024-03-04), [Online]. Available: <https://www.eclemma.org/jacoco/> (visited on 2024-03-04).
- [45] R. Nystrom, *Game programming patterns*. Genever Benning, 2014.
- [46] A. Omicini, A. Ricci, and M. Viroli, “Artifacts in the A&A meta-model for multi-agent systems,” *Autonomous agents and multi-agent systems*, vol. 17, pp. 432–456, 2008.
- [47] M. Osman, “An evaluation of dual-process theories of reasoning,” *Psychonomic bulletin & review*, vol. 11, no. 6, pp. 988–1010, 2004.
- [48] J. K. Ousterhout, *A philosophy of software design*. Yaknyam Press Palo Alto, CA, USA, 2018, vol. 98.
- [49] PMD. “Pmd source code analyzer.” (2024-03-04), [Online]. Available: <https://pmd.github.io/> (visited on 2024-03-04).
- [50] A. Ricci, A. Ciorrea, S. Mayer, O. Boissier, R. H. Bordini, and J. F. Hübner, “Engineering scalable distributed environments and organizations for MAS,” in *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS), 2019, Canada.*, 2019.
- [51] A. Ricci, A. Omicini, M. Viroli, L. Gardelli, and E. Oliva, “Cognitive stigmergy: Towards a framework based on agents and artifacts,” in *Environments for Multi-Agent Systems III: Third International Workshop, E4MAS 2006, Hakodate, Japan, May 8, 2006, Selected Revised and Invited Papers 3*, Springer, 2007, pp. 124–140.

- [52] A. Ricci, M. Piunti, and M. Viroli, “Environment programming in multi-agent systems: An artifact-based perspective,” *Autonomous Agents and Multi-Agent Systems*, vol. 23, pp. 158–192, 2011.
- [53] A. Ricci, M. Viroli, and A. Omicini, “CArtAgO: A framework for prototyping artifact-based environments in MAS,” in *International Workshop on Environments for Multi-Agent Systems*, Springer, 2006, pp. 67–86.
- [54] S. J. Russell and P. Norvig, *Artificial intelligence: A modern approach*, 3rd. Prentice Hall, 2010.
- [55] “Software engineering – Software life cycle processes – Maintenance,” International Organization for Standardization, Standard ISO/IEC/IEEE 14764:2022, 2022-01.
- [56] SpotBugs. “Spotbugs.” (2024-03-04), [Online]. Available: <https://spotbugs.github.io/> (visited on 2024-03-04).
- [57] K. E. Stanovich and R. F. West, “Individual differences in reasoning: Implications for the rationality debate?” *Behavioural and Brain Science*, vol. 23, no. 5, pp. 665–726, 2000.
- [58] Summer school on AI for Industry 4.0. “Summer school on AI for Industry 4.0.” (2023-05-05), [Online]. Available: <https://ai4industry.wp.imt.fr/> (visited on 2024-03-02).
- [59] L. Tummolini, C. Castelfranchi, A. Ricci, M. Viroli, and A. Omicini, ““Exhibitionists” and “Voyeurs” do it better: A shared environment for flexible coordination with tacit messages,” in *Environments for Multi-Agent Systems: First International Workshop, E4MAS 2004, New York, NY, July 19, 2004, Revised Selected Papers 1*, Springer, 2005, pp. 215–231.
- [60] M. Van Steen and A. S. Tanenbaum, *Distributed systems: principles and paradigms*. Pearson Prentice Hall, 2007.
- [61] W3C. “Autonomous Agents on the Web Community Group.” (2024-02-19), [Online]. Available: <https://www.w3.org/community/webagents/> (visited on 2024-02-19).
- [62] W3C SPARQL Working Group, “SPARQL 1.1 Overview,” W3C, Recommendation, 2013-03-21.
- [63] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, “A note on distributed computing,” in *International Workshop on Mobile Object Systems*, Springer, 1996, pp. 49–64.
- [64] G. Weiss, Ed., *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT press, 1999.

- [65] D. Weyns, A. Omicini, and J. Odell, “Environment as a first class abstraction in multiagent systems,” *Autonomous agents and multi-agent systems*, vol. 14, pp. 5–30, 2007.
- [66] M. Wooldridge and N. R. Jennings, “Agent theories, architectures, and languages: A survey,” in *International Workshop on Agent Theories, Architectures, and Languages*, Springer, 1994, pp. 1–39.
- [67] A. Zimmermann, A. Ciortea, C. Faron, E. O’Neill, and M. Poveda-Villalón, “Pody: a Solid-based approach to embody agents in Web-based Multi-Agent-Systems,” in *International Workshop on Engineering Multi-Agent Systems*, Springer, 2023, pp. 220–229.