ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS OF CESENA

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
Second Cycle Degree in Computer Science and Engineering

AGENT-ORIENTED VISUAL PROGRAMMING FOR THE WEB OF THINGS

Thesis in
PERVASIVE COMPUTING

Supervisor
Prof. ALESSANDRO RICCI

Co-supervisor
Prof. SIMON MAYER

Presented by
SAMUELE BURATTINI

Academic Year 2020 – 2021

to Giada

Index

Introduction

In the context of a Pan-European project aimed at defining the next generation of intelligent IoT systems, this thesis proposes the idea of Visual Agent-Oriented Programming as the enabling tool for non-technical users to configure and program such systems.

The thesis work was carried out while being hosted by the University of St. Gallen, contributing to the exploration of the Interaction and Communication-based Systems research group in the field of engineering autonomous systems capable of controlling Web of Things (WoT) environments with a human-in-the-loop philosophy.

Motivated by the ever-increasing demand for interfaces designed for workers to be able to keep the pace of the fast digitalization of business activities, a new tool was designed and developed to provide a seamless interface to control the digital representation of physical environments exploiting agent programming.

To our knowledge, this is also the first attempt at building a user-friendly interface on top of the agent-oriented paradigm that targets people with no previous programming experience. This can be seen as the first step in exploring whether the assumptions made when defining agents as entities to build software in a way that was more understandable for humans since taking inspiration from models of human behaviour and reasoning still hold.

From a WoT perspective, providing a uniform programming interface is crucial to allow to create or modify software featuring a different degree of autonomy in flexibly performing tasks, dealing with open, dynamic and distributed environments. Users can leverage the agent paradigm to define complex behaviour at a very high level of abstraction leaving space for all sorts of reconfiguration to happen in the lower layers such as on-demand machine allocation.

Chapter 1 goes deeper into the motivations that brought to the definition of this research proposal, Chapter 2 presents the state of the art that was taken as reference for the three different fields involved which are: Agent-Oriented programming, Visual Programming and Web of Things.

In Chapters 3, 4 and 5 the design and development process is described highlighting the requirements envisioned for the system and how they were

technically achieved to produce a working prototype.

Finally, Chapter 6 describes the evaluation process that consisted in testing the implemented prototype with a user study to gain qualitative feedback that can drive future iterations of the solution.

Chapter 1

Context, Motivations and Research Proposal

This project was born from the collaboration of the PSLab [1] of the University of Bologna and the Interaction and Communication-based Systems [2] research group at the University of St. Gallen, Switzerland.

Since both groups shared interest in similar topics concerning the augmentation of smart environments through the use of software agents, while at the same time trying to keep "humans in the loop", the idea was to find an interesting research proposal that was coherent with this themes.

Coincidentally, the St. Gallen group was working on a European project named IntellIoT[3] that funded research projects on the development of systems that could make the existing IoT technologies, more intelligent and autonomous. The project ended up shaping the requirements of the thesis work itself and creating the opportunity for an internship to develop the thesis work abroad.

In the following sections first, the general objectives of the IntellIoT project are presented to provide the context in which the thesis was conceived. Secondly, motivations for the increasing need and interest in end-user programming for domain-specific applications are shown, to further validate the proposed work. In the end, the general proposal for the thesis is described, this then shaped the requirements that are listed in Chapter 3.

[1] `https://apice.unibo.it/xwiki/bin/view/PSLab/`
[2] `https://ics.unisg.ch/chair-interactions-mayer/`
[3] `https://intelliot.eu/`

1.1 The IntellIoT Project

IntellIoT defines itself as a Pan-European project focusing on developing integrated, distributed, human-centred and trustworthy IoT frameworks, applicable to agriculture, healthcare and manufacturing.

With its thirteen partners scattered around nine countries, the project explores the applicability of new enabling technologies such as 5G connectivity, distributed technology, Augmented Reality and AI to real-world fields in combination with IoT sensor networks.

In the context of this broad project, the University of St. Gallen is focusing on managing Web of Things enabled intelligent robots using multi-agent systems. The work of this thesis is exploring how humans can define the behaviour of such systems effectively.

1.1.1 Key Mission

The overall goal of the IntellIoT project is to develop a reference architecture and framework for IoT semi-autonomous applications and environments that present evolving intelligence through the interaction with human experts and a reference communication and computation framework that adapts to changes in the environment and has built-in security, privacy and trust[18].

Three pillars emerge from this mission, that are the central research topics on which the project revolves around:

1. **Collaborative IoT**: multiple semi-autonomous entities will have to cooperate in order to reach the system goal. These entities will be aware of themselves and their surroundings through sensing technologies and will have a different amount of knowledge about the task they need to perform. Since providing complete knowledge is almost impossible in an open evolving scenario, they will also need to improve their knowledge through Machine Learning capabilities either by discovering new details through the environment or by interacting with the other entities within a secure communication network.

2. **Human-in-the-Loop**: the human is considered a source of invaluable experience and knowledge. With this in mind, the idea is to keep the human in a central role. Instead of removing him/her from the system, the plan is to use the human experience to overcome unknown situations when the system does not have the knowledge (yet) to solve a problem and have the system learn from it to overcome future similar situations. The human is still at the centre of the system, and to do that he/she needs new efficient tools to interact with the machines.

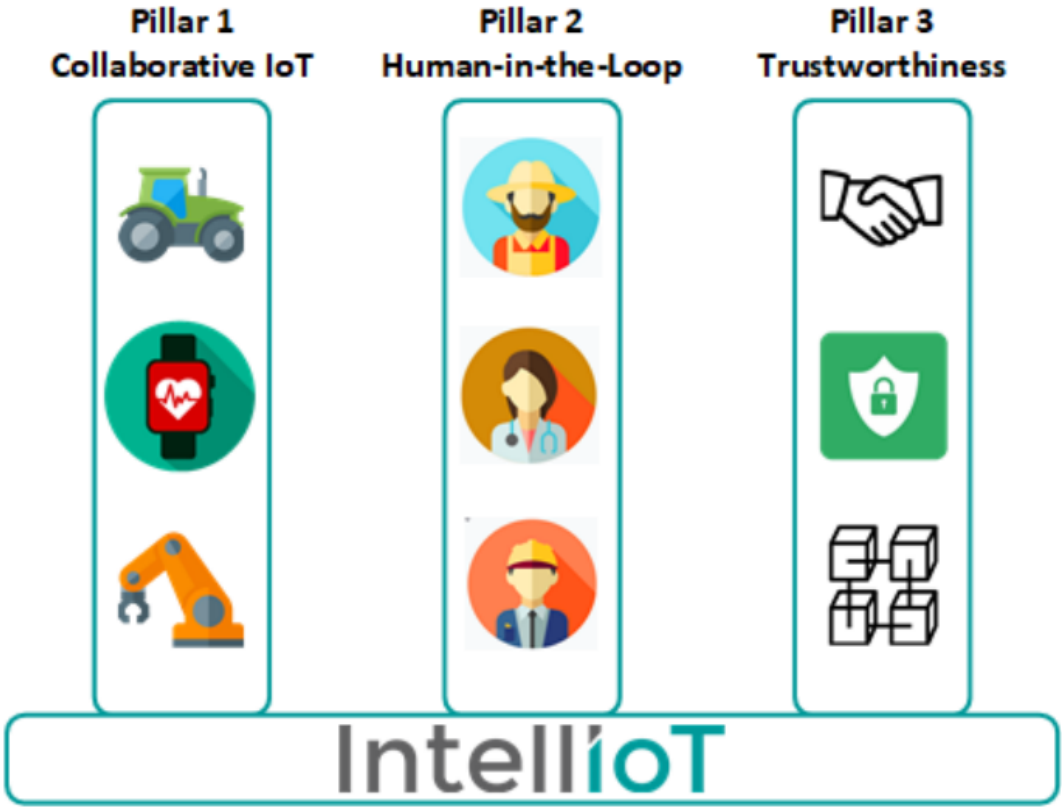


Figure 1.1: The three pillars of IntellIoT

3. **Trustworthiness**: Security, Privacy and Trust are considered essentials for the broader acceptance of IoT systems and applications. To realize reliable systems trust must be ensured from the autonomous components as well as it is nowadays required from the human members of an organization. As well as applying all the known techniques to ensure the treatment of data safely, the system needs to be monitored and transparent in its decision process.

These pillars are not only objectives of the system, but also help shape the requirements of the different use cases. In each use case, the domain is exploited to create a unique scenario to work on each of these pillars independently exception made for the third one (Trustworthiness) that by nature has to be pervasive in every solution that the project will deliver.

1.1.2 Use Cases

The project targets three selected use cases coming from real-world applications of current IoT technologies. These areas have been selected because

they feature heterogeneous IoT enabling technologies, device types, network deployments and performance requirements.

The three chosen use cases are:

1. **Agriculture**: in the agriculture domain the concept of "smart farming" is already a thing, with careful detection of individual plant needs through sensing technologies and deployment of specialized treatments instead of investing large quantities of substances uniformly over large fields. The next step that IntellIoT wants to address is towards automating processes by employing self-driving tractors equipped with sensors and able to perform tasks that are usually tiring for human operators and thus potentially unsafe. Of course, farmers will be away from the field but still important to the management of the farm, being able to remote control the tractor in case of unknown obstacles and in defining the overall goals of the farming system leveraging their knowledge and experience.

2. **Healthcare**: in the field of healthcare IntellIoT is interested in dealing with remote patient monitoring: an interesting application of wearable IoT devices that could help relieve the social (and financial) burden of chronic diseases. The idea is to improve treatment by enabling constant monitoring of the patient health state, but letting the patient carry on with regular life activities. With a system like that in place, the AI-powered monitoring system will still rely on human experts to assess and judge potential health threats and understand unknown situations or errors. The critical point of the system is to be able to filter efficiently just the right amount of data to be sent to the professionals so that they can gain proper insights without being overwhelmed.

3. **Manufacturing**: Industry 4.0 aims at the complete digitalization of manufacturing processes governed by IoT technology and supported by as little human intervention as needed in the whole cycle from a customer order submission to the realization of a piece and its delivery. Following these requirements, this use case aims to enable individualized customization and realization of a simple product. This goes towards an idea of manufacturing-as-a-service using shared machines on a single shop floor. This requires the system, once data from the customer is received, to identify needed machines, plan a path for the piece moved by self-driving vehicles around the plant and instruct machines with the required steps to build the product. The whole process can be queued for approval from a human supervisor to teach the AI and improve the solutions as well as ensure safety to operate in the environment.

The use cases help ground the project down in specific situations that can be immediately perceived as beneficial for future IoT systems. They also provide easily understandable domains that can be explored and used for testing during the development phase of the project so that the proposed solutions are always challenged with specific domain problems.

For this thesis work, some tests were carried out in a simulated farming scenario deliberately inspired by the agriculture use case of the IntellIoT project.

1.2 Domain-Expert Programming

As seen in the mission of the IntellIoT project, keeping humans in the fast-paced machine-controlled world is a very important challenge for the days ahead. The expertise of well-trained workers and the ability of people to adapt quickly and develop alternatives is not (at least for now) matched by Artificial Intelligence and is a valuable resource that can't be neglected in the process of digitalization.

Also, one of the oldest known problems when dealing with software development is the gap between the programmer's comprehension of the domain and the domain itself. This often causes misunderstanding and errors when creating software solutions for real-world problems and research is always looking for tools and processes to try to make this gap thinner.

This is usually accomplished either by training programmers and analysts to extract the domain knowledge from the experts and reflect it into the code (like with Domain-Driven Design techniques) or by directly empowering the user with tools that require the expert himself to "code" or configure some parts in a complex system often making use of domain-specific languages or even visual tools that are considered more user friendly.

We envision a kind of end-user programming that may be defined as *domain expert programming* where the programmer is not necessarily just any final customer of a software solution, but a person with deep knowledge of the domain in which the software may be useful.

In this project context, domain experts are further identified as people whose experience in the field of modelling business processes using smart machines is extremely valuable, but whose lack of programming training is limiting the capability to configure such systems and schedule such configurations for execution.

This creates the need for a tool that could help these people to build a solution from scratch. Low-code (or even no-code) environments are powerful tools designed to enable people without a proper computer science background to program parts of a system. These environments often offer an intuitive vi-

sual way of assembling programs to relieve an inexperienced user from the challenges given by syntax, scope and other well known technical details associated with almost every programming language.

To this purpose, experimenting with new tools that can provide a graphic interface with both close to domain abstractions and high programming flexibility is a key point to keep the human expertise and at the same time increase the level of autonomy and intelligence of complex systems.

1.3 Proposing Visual Agent Programming

The proposal for this thesis merged the needs of the IntellIoT project with the ever-increasing interest in end-user programming for industrial domains and the research interests of both the involved groups in agent-managed systems of smart things.

The resulting idea is the design and implementation of an agent-oriented visual language and a supporting development environment to *agentify* the control of smart things.

While both visual tools and multi-agent systems have been used in the context of IoT applications (as seen in [27] or [11]), the idea of approaching agent-oriented programming with a visual abstraction is instead, to our knowledge, a novelty.

In this context, the WoT is kept as a domain constraint while the exploration of how to port agent programming to a visual level is conducted. The core thesis work will be based on studying the existing agent programming tools and developing a credible mapping that could leverage both the expressiveness of agent-oriented languages and the ease of use of graphic programming interfaces.

In particular, the chosen agent architecture will be the Belief Desire Intention (BDI) model, because originally intended to lower the gap between *human practical reasoning* and code. In this sense, by designing an effective visual tool, we expect to be able to use it going forward to study whether the original claim is valid and people actually *think* as agents do.

A qualitative study was also planned to gain feedback for the developed interface and provide the first insights in this direction by evaluating how non-technical users responded to solving problems with the visual agent-oriented programming language after little to no training about either “classical” programming or the adopted agent model.

Before continuing with the technical description of the implemented solution, the next chapter introduces the state of the art of the main cited enabling technologies used in the project.

Chapter 2

State of the Art

To better understand this thesis project, an excursus of existing literature about the three main axes around which the project is developed is presented.

The main contribution of the thesis project itself is indeed the seamless integration of these three orthogonal research fields that are: Agent-Oriented Programming, Visual Programming and Web of Things technologies.

The following sections aim at providing a basic understanding of the research done so far on these topics and what is considered the state of the art of each field as well as presenting the technologies used to develop the thesis project itself.

2.1 Agent-Oriented Programming

Software agents have their roots in the Distributed AI community and can be traced back as far as the 1970s with the first definition of the Actor model. The definition of an Actor was that of a computational *agent* which has a mail address and a behaviour. Actors communicate by message-passing and carry out their actions concurrently[17].

In the first years of their conceptualization research focused on the definition of agent models and architectures and on how agents could cooperate to solve problems. The term quickly became so broad that it's hard to define properly what an agent is since many categories and applications exist and the word 'agent' is really an umbrella term for a heterogeneous body of research and development[24]. This can lead to confusion about what to expect from an agent and how to classify a software system as agent-based or not.

Since agents became such an important concept in the community, the idea of having agent-oriented programming as a new paradigm emerged. The name is derived from Object-Oriented programming since the core idea of agent programming is that it's a specialization of the first model in which instead of

having objects defined by their properties interacting through method invocation, we have agents defined by their *mental state* interacting through semantic message exchange[30].

For the scope of this thesis though, a clear definition of agents is needed. In the following sections, the notion of agent adopted in this thesis is presented alongside the reference architectural models used to realize both agents and multi-agent systems. Lastly, the supporting technologies used to implement the software solution are shown.

2.1.1 What is an Agent

To give a definition in just one sentence of what is now generally accepted to be considered an agent, this quote by Wooldridge[37] can be used:

> *An agent is a computer system that is situated in some* ***environment****, and that is capable of* ***autonomous action*** *in this environment in order to meet its* ***design objectives***

This (apparently) simple definition encapsulates all the basic aspects that need to be taken into consideration when thinking about agents. First of all, an agent is situated in an environment, which means that a notion of such environment must be defined for the agent to live within it and observe it to then be able to act upon it *autonomously*. Finally, an agent must act to meet its design objective which means that agents are created with a specific goal in mind to be achieved and their autonomy is merely intended as a means to reach that objective.

To further refine this definition it's important to introduce the notions of *weak* and *strong* agency as well since they determine the properties that are expected to be satisfied by gent systems.

Weak Agency is recognized to any software system that exhibits the following properties:

- **Autonomy** which means that agents can operate without the need of direct human intervention and have control over their actions and internal state;
- **Social ability** which means that agents are capable of interacting with other agents (and possibly humans) with some form of communication language;
- **Reactivity** which means that agents perceive their environment, and respond to the events that arise from that environment;

- **Pro-activeness** which means that agents do not only react to the external stimuli coming from the environment but can exhibit goal-directed behaviour and take the initiative.

Strong Agency is recognized to systems in which "weak agents", that have all the properties listed above, are conceptualized or even implemented with concepts that are usually applied to humans, for example, is often the case to use notions of beliefs, knowledge, intentions etc.[38]

For the scope of this thesis when the agent term is used it is referring to an agent under the strong agency notion. In particular, the thesis work is revolving around agents designed with the Belief Desire Intention model, discussed below.

2.1.2 The BDI Agent Model

The Belief Desire Intention model is an agent architectural model based on human practical reasoning. Its definition was motivated by the need of having a resource-bounded intelligent agent capable of both means-end planning and weighting of competing alternatives[7].

To achieve the desired properties, the agent architecture was inspired by how human behaves, in particular, to reduce the time spent on deliberation, the model introduced the idea of *intentions*, alongside beliefs and desires which were already commonly used, to indicate that, once chosen, an agent should persist with a plan to some degree instead of continuously reconsider all the possible routes, just like people do.

First of all, for most, it might sound strange to give to a computer system the definition of a *mental state*, but if we qualify the entities composing such mental state it becomes clear how this helps in modelling an agent-based system:

- **Beliefs** are defined as information that the agent has about the environment. This information, since it's internal to the agent, can be outdated or inaccurate but the agent believes it's true. They are not so dissimilar from variables holding data in any other computer program;

- **Desires** are all the state of affairs that the agent *might* want to accomplish. This does not imply that the agent acts upon a desire, they just potentially influence the deliberation process. These are the options the agent can choose to pursue;

- **Intentions** are the state of affairs that the agent has decided to work towards. From all the possibilities that the agent has considered either them coming from the outside or his internal desires, the agent chooses to *commit* to a specific intention. Note that this process can be recursive, every time narrowing down the options available for the agent that is following a specific intention[6].

The powerful idea behind using these concepts to model computation of autonomous entities is that they are both very high-level and easily understandable since used by people in everyday life to reason about other people's behaviour (e.g. we can say that a person buying a ticket intend to take the train and vice versa).

To then traduce these conceptual entities in a computational model, it's possible to envision a control-loop for the agent that can keep a *rational balance* between beliefs, goals, actions and intentions[12]. The loop sequence can be summarized as:

1. Observe the environment to perceive changes;
2. Update the set of beliefs and determine the current desires based on beliefs and intentions;
3. Choose among the desires which ones to pursue that become intentions
4. Generate a plan to achieve the intentions.
   (a) Execute a step of the plan
   (b) Observe the environment to perceive changes
   (c) Update the set of beliefs and determine the current desires based on beliefs and intentions;
   (d) Check if it's worth reconsidering its current intention to deliberate again
   (e) Check if the current plan is still sound to achieve the intention or if it needs to find another one
   (f) Repeat the inner cycle if the plan is still ok
5. Repeat the outer cycle if the plan is successful or if the intention is now impossible to reach for some reason.

Of course, the crucial elements are the continuous observation of the external environment and the capacity of deciding whether it's worth reconsidering an intention or keeping following it. This kind of control structure allows the

agent to be sufficiently complex to act "freely" while keeping a commitment to an intention and not getting lost in continuous deliberation.

One of the most famous implementations of the BDI model is the Procedural Reasoning (PRS) System[13] which is based on the fact that the agent doesn't need to plan since it's equipped with a *plan library* manually constructed by developers. This simplifies of course the whole process, reducing the time needed by the agent to plan since it's not necessary to compute a new one but just choose whether an applicable one is present in its library.

Plans in PRS are defined by a *goal* a *context* and *body*. The goal is needed to define what the plan accomplishes, the context determines when a plan is applicable whereas the body is a sequence of actions. The interesting approach of PRS is that plan bodies can have new goals as actions, this implements the behaviour of deferring a decision at the last moment since when a new goal is encountered during the execution of a plan the agent stacks a new intention on top of the previous one and works to accomplish it as usual before resuming the rest of the plan (or concurrently depending on the semantic of the goal itself).

When we speak of plans in this thesis we refer to this model since the PRS implementation of the Belief Desire Intention model is the one adopted for the implementation of the enabling technology to program agents that was used for this project: *Jason*[5].

2.1.3 Multi-Agent Systems and the A&A Metamodel

In agent-related literature, the notion of a multi-agent system (MAS) is pervasive since, although systems composed by a single agent are possible, it is often more useful and interesting to build systems where more than one agent work together (or competitively) in the same environment.

As it can be seen in Figure 2.1 the structure of a multi-agent system is composed of the environment and the agents that exist within it. Agents have a sphere of influence on the environment which means that they either observe and/or have control over it. Of course, these portions might overlap which means that the effect of one agent's actions upon the environment can influence the behaviour of another agent.

Agents in a MAS are often collaborating to the achievement of a high-level goal, this creates the need of defining some organizational relationships that define roles among agents that regulates how they can interact with each other. In a multi-agent system, then, three dimensions are intertwined: the agent dimension, the organization dimension and the environment dimension.

Regarding the latter, we defined agents as situated in an environment and we've seen how this environment is fundamental in the BDI architecture to

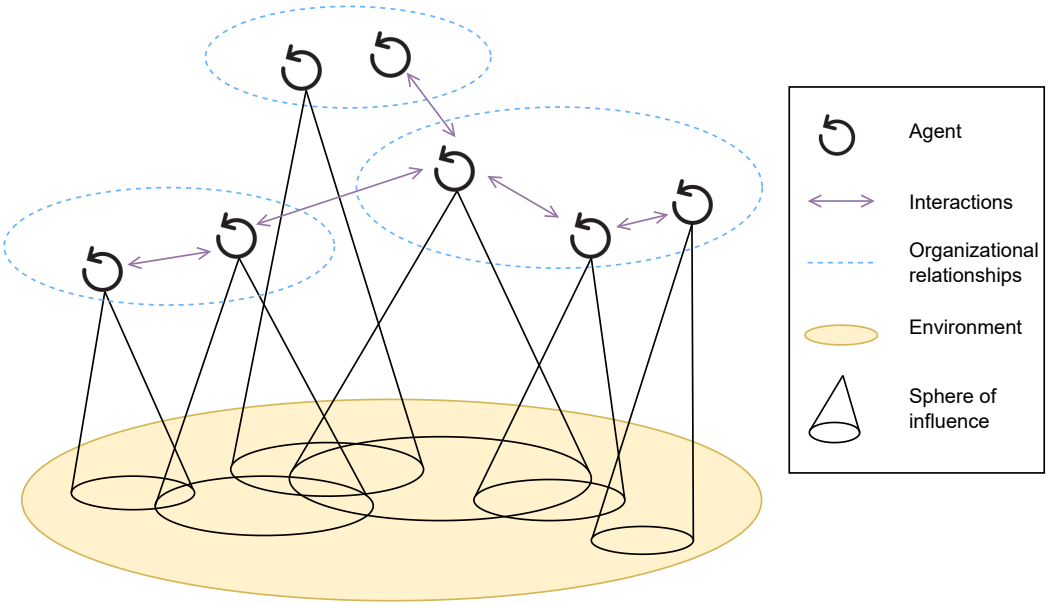


Figure 2.1: The visualization of a multi-agent system showing the relationships among agents and with the environment[19]

let the agent acquire new perceptions and change its beliefs. It's easy to understand that how the environment is shaped has an important influence on the whole agent system.

One of the proposals for describing and defining environments is the Agent & Artifacts metamodel. This conceptual framework introduced the idea of having the necessity to program the environment for effective multi-agent software engineering with the ultimate goal of providing agents with a proper *working environment* where they could find, create and share appropriate tools that could help them achieve their goals in the form of artifacts.

The notion of *Artifact* is an abstraction meant to represent an entity belonging to the environment, thus external from the point of view of an agent. Such entities can be created, shared, used and disposed of by agents to carry out some tasks. Artifacts types should be defined by the MAS engineer encapsulating a specific function and purpose and they can also be placed in the environment to design its initial state[28]. Artifacts should be used to model only passive entities, that become useful only through the interaction with agents.

To provide a notion of locality artifacts can be organized into *workspaces* that define the scope of visibility of an artifact for the agents in the MAS. To get access to an artifact agents must join the appropriate workspace in which artifacts are shared.

The artifact model exposes an interface made of an observable state and operations and events. This mimics the way humans interact with their tools and, in the same way, agents can inspect the state of an artifact (e.g. watch how much time is left on a timer), invoke operations on it (e.g. activate the timer) or react to changes notified by the artifact itself (e.g. listen to the ring signalling the end of the timer).

A framework to build MAS environments named CArtAgO was developed implementing the Agent & Artifacts metamodel to be used with different kinds of cognitive agents.

2.1.4 The JaCaMo Platform

When it comes to tools to engineer and implement multi-agent systems the most notable is the so-called JaCaMo platform which was designed to incorporate in a coherent solution the three orthogonal dimensions typically present in MAS shown in Figure 2.1.

To achieve so the JaCaMo platform fuses together three technologies from which its name originate, namely:

- **Jason** as the BDI agent programming language to code autonomous intelligent agents for the multi-agent system;
- **CArtAgO** as the framework based on the A&A metamodel to define the environment in which agent will be placed;
- **Moise** as the organizational specification model to manage agent organizations and roles within the MAS[16].

The JaCaMo platform managed to achieve its goal of proving that there is a benefit of building a multi-agent system taking into consideration these three fundamental dimensions and having them fully integrated into a single tool[4].

The JaCaMo platform was chosen as the enabling tool supporting the execution of the agents built with the visual language developed. For this reason, as explained in further details in section 4.1 the visual language is based on Jason also due to its seamless integration with the execution platform.

2.2 Visual Programming and End-User programming

Visual Programming is defined as the action of programming using more than one dimension to convey semantics[9]. Traditional text-based program-

ming is considered mono-dimensional since, although visually organized on a 2D screen, code can be seen as a single string of characters.

In the field of Visual Programming two main concepts are identified:

- **Visual Programming Languages** (VPL) are programming languages designed in a way that leverages the multi-dimension space to create new syntactic expressions visually;
- **Visual Programming Environments** are tools that use visual expressions to generate code that may have a different syntax from the code used to edit it. This commonly means creating visual interfaces on top of traditional text-based languages to simplify the syntax and leverage the power of a VPL.

Historically, Visual Programming started out trying to change the entire software development process but was heavily criticized since the proposed tools were interesting for "toy" software projects and failed to generalize to more complex scenarios. This is partially due to a methodological problem that makes it hard to effectively test newly developed tools because of [35].

The scope of Visual Programming tools was then changed to either focus on parts of the development process (e.g. building GUIs with Visual Basic), or on specific domains, creating some kinds of Visual Domain-Specific Languages that since working with restricted domain could leverage the visual abstraction to a different level and gain more from the positive impact of it.

In general, the main goals of Visual Programming are:

1. to make programming more accessible to some particular audience;
2. to improve the correctness with which people perform programming tasks,
3. to improve the speed with which people perform programming tasks.

It's then intuitive to see how and why Visual Programming has been often paired with the idea of end-user programming, specifically in the connotation of visual domain-specific applications.

This field is extremely relevant if we think about the fact that most software (in terms of quantity) is actually written by people with expertise in different domains than computer programming but who might need to support their activities with computers, rather than by professionals.

End-user programming can be defined by the *goal* that the programmer has when writing software, often this goal is simply to solve a problem on the fly with a computer and maybe forget about it later, thus ignoring all the features typical for industrial software engineering such as maintenance or testing[20].

In different definitions, the same term is also used to indicate programming done by everyday people that are not programmers in their regular work life[22]. In this thesis, the term is used with this meaning even if *domain-experts programming* might be the best-suited term to indicate the activity of automating a task through programming done by professionals in a specific field.

2.2.1 Block-based Visual Programming

One of the most successful examples of Visual Programming Environments applied to end-user (or novice users) is block-based visual programming.

The core idea is to present the user with a primitives-as-puzzle- pieces metaphor as the main visual abstraction to convey visual cues to users indicating where and how commands may be used. Writing a block program consists of dragging-and-dropping instructions together and writing small portions of text for values, names etc. The possibility of syntax error is prevented since the programming environments don't allow to snap together blocks that shouldn't be connected[32].

Block-based languages support the user not only removing syntax problems but also making available all the possible instructions in easily browseable block palettes so that they don't need to know the language in advance. Blocks are also often differentiated by colour, to convey groups of similar functions and the possibility of including text in the block helps describe the function that the block encapsulates better than classical programming languages (e.g. a variable increment which might be written as `x=x+1` in Java can be expressed as a block with the wording `increment x by 1`).

The idea of block-based visual programming is not new, although it gained increasing popularity in the last few years with projects like Scratch[21] that pushed block programming as a learning tool for children to introduce them to the world of programming and algorithmic reasoning and AppInventor[25] that brought many smartphone users close to the world of mobile application development by giving them a fast integrated tool to design both the user interface and the behaviour in an online editor.

Since this rise in popularity, studies have been conducted to assess what makes this approach so successful[33]. When directly comparing text-based languages with block environments the majority of users answered that blocks were easier to use thanks to the visual cues, and the controlled environment that prevents the frustration of mistakes and felt more natural and understandable as well (*"Java is not in English it's in Java language, and the blocks are in English, it's easier to understand."*).

This of course opens up the question of whether it's interesting to use such

languages not only as a learning tool but as proper programming environments for end-users. That's why this thesis will focus on block languages as well as other technical motivations discussed in Section 4.1.

2.3 Web of Things

First introduced in 2007[15] the Web of Things is now a set of W3C standards to improve the interoperability and usability of the Internet of Things (IoT). The idea originated trying to reduce the "silo effect" coming from the industrial development of IoT technologies. The proposal is to apply the architectural principles and standardized technologies of the World Wide Web as a way to reproduce the fully connected network experience that we have with the *regular* Internet with the Internet of Things as well.

This can open the door to interesting scenarios for both human and machine interaction with smart things and harness the full potential of complex distributed sensor networks.

2.3.1 Origins and Motivations

To better understand what the Web of Things is and what is the current point of development of the project it's important to give a definition of what it's considered to be a Smart Thing and what is the Internet of Things and why it's important to have connected networks of devices.

As it is described in what can be considered one of the Internet of Things manifesto[2] this idea was born as the miniaturization of computers made it possible to embed devices in common objects. The core idea is to let computers track, produce and reason about data in a typically human-centred Internet since machines do not suffer the limitation in observation attention, time and fatigue that people have. This meant producing data with much more quality and in much more quantity.

The idea of a smart thing was not novel and for a long time theorized in the ubiquitous computing field[34] that envisioned a world where computation would "disappear" in the environment.

To define a Smart Thing we can think of any regular object and give it the ability to communicate about its current state, and possibly its capabilities, giving access to remote control as well.

First, it was possible, using passive RFID tags, to append a piece of digital information to any object to be able to track it, then the ability to embed small sensors and processing units within basically anything became a reality. Giving each of these devices an IP address and the ability to be on its own on the network was the final step towards the definition of a new Internet[14].

This resulted in an explosion of different technological stacks composed of communication protocols, architectures, and creating a panorama of not just one Internet of Things but several silos of connected devices, missing the original point of interoperability.

In this panorama, the Web of Things initiative saw in the historical evolution of the "Internet of Humans" and the World Wide Web the solution to such problem, creating an application-level standardization based on open and shared technologies to connect things and realize true *physical mashups*.

2.3.2 Design Principles

As stated multiple times already the main focus of the Web of Things is true interoperability among different IoT devices, working on different technological stacks and produced by different vendors.

This brings several benefits in both the consumer and the industrial domain thus the W3C is pushing towards the adoption of this standard to allow the creation of interesting applications on top of the internet-connected sensing and actuation devices scattered around the world.

One of the main architectural properties that the Web of Things takes from the Web is the idea of RESTful APIs [36]. By giving Unique Resource Identifiers (URI) to things and their properties they are effectively fully integrated into the Web and can be treated as any other Web resource. It also inherits the idea of different representation formats for the same resource, enhancing interoperability through the classical Content-Type negotiation approach.

Although the RESTful API approach is strong in the WoT community, the architecture does not impose any architecture and does not force a client or server implementation of system components.

In general the Web of Things architecture is based around four pillars [39]:

- **Flexibility**: There are a wide variety of physical device configurations for WoT implementations. The WoT abstract architecture should be able to be mapped to and cover all of the variations;

- **Compatibility**: There are already many existing IoT solutions and ongoing IoT standardization activities in many business fields. The WoT should provide a bridge between these existing and developing IoT solutions and Web technology based on WoT concepts. The WoT should be upwards compatible with existing IoT solutions and current standards.

- **Scalability**: WoT must be able to scale for IoT solutions that incorporate thousands to millions of devices. These devices may offer the same capabilities even though they are created by different manufacturers.

- **Interoperability**: WoT must provide interoperability across device and cloud manufacturers. It must be possible to take a WoT enabled device and connect it with a cloud service from different manufacturers out of the box.

In general, this sums up the philosophy of WoT. The conceptual architecture to implement these principles is based on the definition of Things that are described with *Thing Descriptions* in order to provide a machine-readable manual on how to interact with a device.

2.3.3 The Thing Description Model

The fundamental building-block of the WoT architecture is the Thing Description (TD). Every *thing* (or any virtual entity representing the aggregation of things) must be described by a TD that act as a machine-readable manual for the interaction with the thing itself.

Thing descriptions are based on the notion of *interaction affordances*. This term originates in ecological psychology and was first adopted in the Human Computer Interaction field basing on the definition by Donald Norman:

> *"Affordance" refers to the perceived and actual properties of the thing, primarily those fundamental properties that determine just how the thing could possibly be used.[23]*

An affordance in the real world is, for example, the handle of a door, which is an affordance of the thing door suggesting that it can be opened. Not only, but, usually, the shape of the handle also suggest how to open it.

Similarly, an interaction affordance of a thing is intended in the hypermedia fashion of presenting information and control, thus suggesting TD consumers the possible choices of how to interact with a thing in the form of hyperlinks that can be followed to navigate the Web of Things.

The Thing Description model defines three types of interaction affordances:

- **Property Affordances** An Interaction Affordance that exposes the state of the Thing. This state can then be retrieved (read) and optionally updated (write). Things can also choose to make Properties observable by pushing the new state after a change;

- **Action Affordances** An Interaction Affordance that allows invoking a function of the Thing, which manipulates state (e.g., toggling a lamp on or off) or triggers a process on the Thing (e.g., dim a lamp over time);

- **Event Affordances** An Interaction Affordance that describes an event source, which asynchronously pushes event data to Consumers (e.g., overheating alerts).

Affordances are described defining a `title`, a `@type` which is usually defined using a JSON Schema and one or more hypermedia `forms` defining the operation semantic, the target IRI of the affordance and any additional information on how to use the affordance itself[40].

The standard representation of a thing description is in JSON-LD which is an extension of JSON that supports semantic annotations so that Thing Descriptions are directly interoperable with the semantic Web and can be enriched with RDF triples.

An example of a ThingDescription can be seen in Listing 2.1. The showcased thing is a smart lamp with a boolean property representing the state of the lamp and an action that toggles the state and returns the new value. Even from such a basic example, it's possible to see how the thing description model can describe everyday objects and their affordances to external users navigating the Web of Things.

```
{
    "@context": "https://www.w3.org/2019/wot/td/v1",
    "@type": "HueLamp",
    "id":
        "http://localhost:3000/workspaces/lamp-wars/things/huelamp",
    "title": "huelamp",
    "base": "http://localhost:3000/affordances/lamp-wars/huelamp/",
    "securityDefinitions": {
        "nosec_sc": {
            "scheme": "nosec"
        }
    },
    "security": [
        "nosec_sc"
    ],
    "properties": {
        "state": {
            "type": "boolean",
            "forms": [
                {
                    "href": "state",
                    "op": [
                        "readproperty"
                    ],
                    "contentType": "application/json"
                }
            ],
            "readOnly": false,
            "writeOnly": false
        }
    },
    "actions": {
        "toggle": {
            "output": {
                "type":"boolean"
            },
            "forms": [
                {
                    "href": "toggle",
                    "op": "invokeaction",
                    "contentType": "application/json"
                }
            ],
            "safe": false,
            "idempotent": false
        }
    }
}
```

Listing 2.1: An example of Thing Description, showing a simple lamp used in the project.

Chapter 3

Requirements

As stated in chapter 1, the formulated proposal was to first design a user-friendly tool domain experts could use to program and run an agent system and then develop a prototype implementation to qualitatively assess the effectiveness of the resulting IDE.

This fitted the IntellIoT mission well so the project and the use cases on which it's based were used to ground down the proposed application not just as a simple visual development environment, but a tool where agents and things in the real world are integrated seamlessly using the WoT Thing Descriptions as a way to model a specific domain that experts can understand.

When designing the requirements for the system, some assumptions were made to better identify the target and constrain the required functionalities. Some non-functional requirements were identified as well, leading to technological locks on the components of the system.

3.1 Assumptions and Constraints

The first assumption made when designing the requirements of the system is that all the required smart things were available through the WoT Thing Description standard using the JSON-LD 1.1 representation.

From within the tool itself, users won't be able to edit the descriptions and organize them, this setup has to be carried out either manually or by another tool (or a new version of the prototype).

The other strong assumption made was that things were able to communicate over HTTP. This can often not be the case when dealing with legacy IoT systems that may require different communication protocols supported by the WoT standard, but since it was always meant to be a prototype this was ignored to simplify the implementation.

To constrain the complexity, *event affordances*, present in the Thing Description specification, were also ignored for the time being. This choice limits the capabilities of agents of observing the thing state asynchronously and force both the TD designer and the agent programmer to support active polling for properties.

These choices were made since the thesis scope was less on the best possible integration between agents and the Web of Things and more on the relationship between users and agent programming. Stronger integration with the WoT is delegated to future works.

Regarding users, some assumptions were made to design the system around a specific target. The programming ability of users was considered to be either none or very little, but no hard constraints on this were put in place to keep the system as general as possible.

The hypothesis was instead that users have a strong knowledge of the domain of course since they are considered to be experts. In this specific context, that means that they know and understand deeply the things available in the environment and how to interact with them. If not the designer of the TDs themselves, since that can require some deeper knowledge of computer science, they understand at least the concept of *properties* and *actions* of a thing and the input/output flow when interacting with it.

3.2 Non-Functional Requirements

The main non-functional requirement for the system was to be as much user friendly as possible. This meant having a nice clean interface with a few easy to understand features for users to not feel overwhelmed by the tool.

Other than that, the system was required to be easily accessible and unobtrusive. This was translated in a technological lock of using standard Web technologies to realize the interface to the system as a Web application since the execution of the multi-agent system was never meant to run on the user machine, but on some dedicated server anyway.

Web applications have the benefit of being portable on a set of different machines and devices, although the application was not required to have a smartphone version and was designed to be used with wide screens.

The last non-functional requirement was yet another technology lock on the reference agent-oriented abstraction. BDI agents were chosen primarily since they should be easier to understand for novices, but also because of the tools that the research community has built to support the programming and execution of such agents.

The BDI model has been given support with a programming language

named *Jason* and a Java-based execution platform, *JaCaMo*, that mixes pure agent programming with the *Agents and Artifacts* metamodel for environment programming. The integration of the BDI model with the A&A metamodel is crucial as the use case for this system has the communication with the environment through the WoT at its core.

Since this was a prototypal implementation there was no expectation on performances of the system in general.

3.3 Functional Requirements

The very high-level requirements for the tool were identified after the definition of the core idea was finalized, keeping the application use cases in mind.

The planned system is required to have:

- A visual programming language to develop BDI agents;
- A Web-based application to use the visual language, program agents and submit them for execution;
- An interface to discover and test the behaviour of available smart things interacting through Thing Descriptions;
- Seamless integration of agent to thing interaction in the visual language;
- An execution environment to run a desired multi-agent system configuration;
- Dynamic instantiation and removal of agents in an already running system;
- Agent to thing interaction support from the execution environment;
- Permanent storage of the agent's code to allow for future editing and maintenance of an "agent library".

It's clearly understandable from this list how the main focus of the project is to create a comprehensive tool that, having non-technical users as targets, can mask most of the complexity concerning the integration with the real world and the execution of complex multi-agent systems.

Providing this integration seamlessly and naturally is one of the main challenges in this project scope.

The next chapter will detail how these requirements were translated into software components and how these components are organized to build the system that supports all the required functionalities as well as the motivations behind the design choices.

Chapter 4

Design

In this chapter, the design process of the system is described in detail.

Given that the high-level requirements and objectives identified for the system were stable and complete, there was no need for a detailed plan of each different component before starting the implementation.

The whole process was carried out incrementally, building prototypes first and moving on to design new components when needed. This led to a system built organically, with constant refactoring to improve the quality of the solution and no problems of integration.

A step by step approach was used: starting from the integration of the WoT into a Web UI, later going towards the definition of the block language and finally concluding with the execution support of agents built with the block language.

The integration of the WoT communication from within the agent code was introduced as the last link, building a simulation environment as well to test out the features of the whole system without necessarily needing a physical smart object.

This development process proved successful in adjusting the complexity and scope of the whole project to fit within the hard time constraint of the internship while preserving all of the core features required for the project.

Some parts of the system were designed knowing that they might have needed a full project focused only on them and they were simply supporting the realization of a prototype. This leaves space for future innovation in those fields, while proving the point that the combination of these technologies can bring interesting results.

4.1 From Agent Code to a Visual Language

The main design challenge in this project was the creation of the visual language that could support both an easy way to develop agents and a seamless interface to the WoT affordances to be used within the agent code.

Since this is the core of the whole system, the final product went over multiple iterations over the lifespan of the project, and will probably continue to evolve gaining feedback from real users trying out the system.

Although constant change from this part of the system is expected, the design process was carried out methodically, first deciding on the visual abstraction that suited best the use case, then analyzing the reference language to understand syntax and programming constructs and finally providing a map focusing on enabling users to compose their agents and hopefully learn about the model while doing so.

4.1.1 Choosing the Visual Abstraction

There are multiple ways to support the development of a program making use of a graphic interface. Of course, because different tools highlight different aspects of the program itself, choosing the right one is a crucial point when designing a new platform.

When choosing the visual abstraction three factors were taken into consideration:

- Ease of use for novices;
- Coherence with the agent paradigm;
- Development support.

The main classes of visual programming languages applied when novice users are involved are block-based and flow-based languages.

When considering the two approaches, because of the nature of the agent-programming model it seemed more natural to go with a block-based approach. Flow-based languages are usually applied in the modelling of how data must flow during the execution of the program, but when dealing with agents users define the program in terms of behaviour more than data. BDI agents deal with a set of predefined plans to apply in specific situations.

This kind of reactive behaviour can be easily modelled with blocks, having for each plan a separate “chunk” of blocks triggered by some sort of event. This has been already proven effective by the notorious MIT App Inventor[25] project that allows users to develop Android mobile apps using blocks in a reactive programming fashion.

The App Inventor project was originally backed up by Google which is still supporting a JavaScript library to create, customize and manage blocks on a canvas. This is a strong base model on top of which lots of custom other block languages were developed.

The presence of this tool settled down the block abstraction as the best for this prototypal project since it could help build the language faster relying on a well known and appreciated template.

Of course, although this is the conclusion reached for this project given the requirements and the similarities within languages, it might still be relevant to explore different possibilities as well as try designing a fully customized visual paradigm from scratch.

4.1.2 Reference Syntax and Constructs

To begin the construction of the visual language, it was necessary to study and understand the syntax and identify the "building blocks" of the agent-oriented programming language chosen as a reference.

The language of choice was the BDI agent language *Jason*, an extended version and implementation of the conceptual language *AgentSpeak*. This reference was chosen because it's among the most popular agent-oriented languages also due to the support given by the Jason interpreter and the integrated *JaCaMo* platform.

The AgentSpeak language is a BDI agent language based on logic programming [26]. This means that, contrary to other previous approaches, the representation of beliefs and goals is not implemented using classical data structures, but taking the shape of a first-order logic predicate such as in Prolog. This was done as an effort to provide a stronger connection between the theoretical BDI architecture and its actual implementation and to simplify the model-checking of agent programs.

The analysis of the constructs available in the language, to be ported as visual elements, first involved looking at the Jason syntax and grammar shown in listing 4.1.

An agent is composed by an *initialization* section where the programmer can establish the knowledge that the agent has since the beginning of its execution by defining a set of beliefs, a set of goals to pursue and a set of deductive rules that can be used to simplify the checking of logic conditions. After the initialization, the programmer can define a *plan library*, where all the procedural knowledge of the agent is stored. This defines what the agent can do and eventually how to handle failure.

The difference between the two sections though, is that goals and beliefs in the initialization can not have variables since they would not be grounded.

```
agent → ( init_bels | init_goals )* plans
init_bels → beliefs rules
beliefs → ( literal "." )*
rules → ( literal ":-" log_expr ".")*
init_goals → ( "!" literal "." )*

plans → ( plan )*
plan → [ "@" atomic_formula ] triggering_event [ ":" context ] [
    "<-" body ] "."
triggering_event → ( "+" | "-" ) [ "!" | "?" ] literal
literal → [ "~" ] atomic_formula
context → log_expr | "true"

log_expr → simple_log_expr
    | "not" log_expr
    | log_expr "&" log_expr
    | log_expr "|" log_expr
    | "(" log_expr ")"
simple_log_expr → ( literal | rel_expr | <VAR> )

body → body_formula ( ";" body_forumula )*
    | "true"
body_formula → ( "!" | "?" | "+" | "-" | "-+" ) literal
    | atomic_formula
    | <VAR>
    | rel_expr

atomic_formula → ( <ATOM> | <VAR> ) [ "(" list_of_terms ")" ] [
    "[" list_of_terms "]" ]
list_of_terms → term ( "," term )*
term → literal
    | list
    | arithm_expr
    | <VAR>
    | <STRING>

list → "[" [ term ( "," term )* [ "|" ( list | <VAR> ) ] ] "]"

rel_expr → rel_term ("<"|"<="|">"|">="|"=="|"\\=="|"=") rel_term
rel_term → (literal|arithm_expr)
arithm_expr → arithm_term [ ( "+" | "-" ) arithm_expr ]
arithm_term → arithm_factor [ ( "*" | "/" | "div" | "mod" )
    arithm_term ]
arithm_factor → arithm_simple [ "**" arithm_factor ]
arithm_simple → <NUMBER>
    | <VAR>
    | "-" arithm_simple
    | "(" arithm_expr ")"
```

Listing 4.1: Grammar of Jason, as reported in [5]

Notation	Triggering event type
`+`	Belief addition
`-`	Belief deletion
`+!`	Achievement-goal addition
`-!`	Achievement-goal deletion
`+?`	Test-goal addition
`-?`	Test-goal deletion

Table 4.1: Types of triggering events and the corresponding notation

From a syntactical point of view beliefs and goals are represented as logic predicates (`literal`) that can have zero or more terms. A term can be either:

- an `<ATOM>` which is any lowercase string with no spaces;
- a `<STRING>` which is a string surrounded by quotes;
- a `<NUMBER>` which is any digit and floating-point number;
- a `<VARIABLE>` which is a string starting with an uppercase letter and no spaces;
- or they can even be lists, arithmetic formulas or an atomic formula that allow a higher level of composition.

Plans are defined by their `triggering_event` an optional `context` and the plan `body`.

The triggering event is a literal preceded by a combination of symbols that can express one of the six possible events in table 4.1 that can be used to decide when a plan should be considered relevant.

A plan context is an optional logic expression whose runtime evaluation can determine whether a plan is applicable or not in a given time. Again, a logic expression can have standard boolean conditions mixed with predicates that check against the agent's belief-base and can ground variables.

Finally, plan bodies are a sequence of instruction among expressions (e.g. to assign a variable or do some math), addition/removal of beliefs and goals that can trigger the execution of other plans and also invocation of actions both internal or of an artifact. Invocation of actions shares the same syntax as predicates.

4.1.3 Mapping Principles

Once the main concepts were extrapolated from both a syntactical and semantic perspective, the design process involved choosing a way to map each construct with a specific block.

The approach followed a few core design principles that guided how to develop the block language. These guiding principles can be summarized as:

- **Single Responsibility**: each block should map only one concept of the language;
- **Composition**: complements the single responsibility principle by trying to avoid creating complex blocks that merge multiple functions instead each simple block should be used in composition with other blocks to create valid constructs;
- **Convey Semantics**: blocks should be designed to let the user understand easily what makes them unique and how they can be combined with other blocks;
- **Avoid Block Explosion**: when blocks with similar meanings can be grouped together or reduced to the same block, sacrifice the single responsibility to avoid a huge number of different blocks that would be confusing.

The overall goal was to make a language that was simple enough for novice users to be understood intuitively, but also not limiting for more experienced programmers. Keeping the right balance was definitely a challenge, but the guiding principles helped with that when choices needed to be made on how to represent a specific concept.

To simplify the implementation, though, some advanced features of the language were left out namely lists and annotations. This choice was made to not overload the block language with these features for this first implementation, they could always be added in future by extending the project. Higher-order variables are also limited since they require a much deeper understanding of the language to be used effectively.

4.1.4 Core Blocks

The block language design process culminated in actually defining all the blocks needed to construct an agent program. As stated before blocks followed an iterative design process to be more polished and understandable, but that mainly concerned the UI side (displayed text and colours) more than the actual

structure since the methodical mapping process starting from the grammar and following the listed principles brought immediately to a solid base start.

To better understand the outcome of the whole process and ground the considerations done so far, in this subsection some of the main building blocks are shown and explained.

As stated before, the main thing noticeable when looking at an agent program is the separation between the initialization of the agent's mental state and the actual plan library, so these categories were used as the main groups for blocks as well to make the distinction clear for users as well.

Initialization blocks The initialization part of the agent was wrapped in a block acting as a root. This was modelled as if it was a special plan executing every time the agent is started to use the same semantics as the plan library and also to not leave any "dangling" isolated blocks that could cause confusion or get lost in the canvas space.

Colours were used to make initialization blocks feel different from plan blocks and let people understand that they could only stack together.

As in Jason, users can initialize agents by adding beliefs which to let people familiarize themselves with the concept are called *notes* as if the agent is writing down post-its containing all the information he needs to remember to convey the behaviour and life cycle of a belief.

Users can also tell their agents to achieve a goal. Or define a logic rule that can be later used to simplify the code (the agent knows __ is true when __).

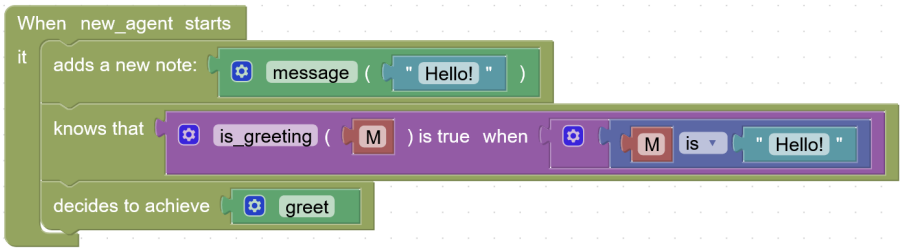
Goals and beliefs, since sharing the same syntax, are represented with the same block that has the syntax of a predicate and can accept only grounded terms as arguments.

An example of an agent initialization block can be seen in Figure 4.1a.

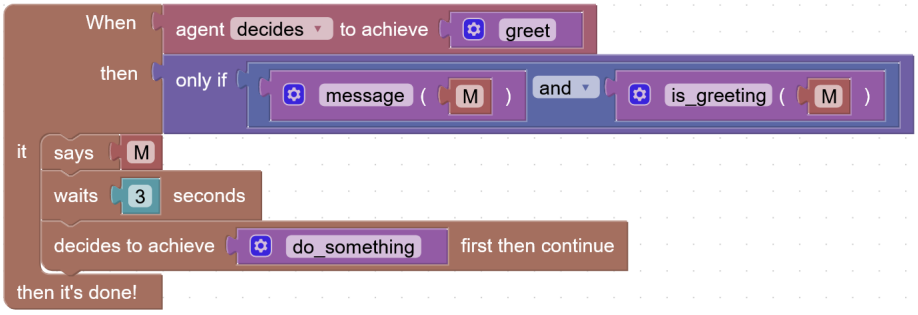
Plan blocks Plans were modelled with a root block that served as a container for the plan triggering event, the context and the actual plan body. Each plan can easily be identified by an agglomerate of blocks once again to keep the workspace as clean as possible. One thing that was noticed, but not addressed, is that plans rely on the order of definition when matching the same triggering event so the user should pay attention to where the plans are placed in the visual interface (but an automatic ordering system could definitely be implemented and help the overall clarity).

Again colours were used to emphasize which blocks could be put together, and they are purposefully quite different from initialization blocks.

Triggering events were modelled coupling addition and deletion of the same concept as a selectable option of the same block to limit the number of blocks.



(a) Initialization blocks



(b) Plan blocks

Figure 4.1: Portions of an agent created with the visual language.

Different types of goals and beliefs triggers were still kept separated since semantically different but they do share the same block shape and colour.

Contexts were modelled as expressions using classic logic operators. They can be combined to be as long as needed mixing predicates with boolean expressions. To avoid leaving an empty context since it is an optional part in the definition of a plan, a special *always* block was created so the user could check if there are blank connections left to consider their agent complete.

Blocks similar to the ones used to represent beliefs and goals are used with a different colour to symbolize the fact that they can accept variables. They share the same shape since they share the same syntax but have a different meaning and can be used in different contexts. Instead of using the same blocks to represent actions, a few relevant actions are directly mapped to specific blocks in order to enforce basic typing and guide the user when choosing to insert them in the code.

Interacting with things is considered an agent action, more on that in the next section.

4.1.5 Integration with the WoT

Of course, part of the design process was dedicated to understanding how to make agents interact with smart things and especially how to fit this naturally within the visual language. To focus as much as possible on the natural integration of concepts, blocks were designed first and later the actual implementation of the execution support was done adapting to the design.

This lead to a very clear design that hides all the unnecessary technical concerns from the eyes of the final user, to which interacting with a Web Thing is not so different from any other agent's action. Blocks were designed by keeping as reference the Thing Description model, mapping interaction affordances into simple operations.

Of course, blocks needed to present a grade of flexibility since, in order for the final solution to be effective, they needed to be automatically generated from the TD itself. As said before, only property and action affordances were considered for this project scope. Of course, since they represent semantically different operations they were modelled as different blocks.

An important difference between these different kinds of affordances is that actions can have some form of input whereas property must always have output. This can be further defined by providing the JSON Schema description of the expected object to be sent or received through the affordance API. JSON Schemas are considered optional in the Thing Description model, but it's needed for the correct use of the tool so that the automatic generation of blocks can show what are the expected values to be used when interacting with a thing.

This posed the need for blocks to represent and manipulate JSON objects. Again, following the same mapping principles described above, the choice was to split the actual affordance blocks from the JSON manipulation since they represent different functions of the language although usually coupled.

The actual implementation of agent-to-thing interactions required some work at the infrastructure level, using the tools provided by Jason and JaCaMo to extend agent capabilities. How this was concretely achieved is described in the next chapter, the main point from a design perspective is that at the user level no details about the underlying technicalities should be shown.

4.2 Identifying Components and Architecture

During the design process, it was necessary to identify all the software components needed to realize the requirements for the system.

This was done trying to understand and separate the responsibilities, to maintain a clean, and expandable, architecture. In total, six modules were

identified to compose the system:

Smart Environment TD Repository This module is responsible for storing and serving the Thing Descriptions (TD) of all the available things in the environment. This is necessary to provide access to the TDs for the user to be able to understand them, try them out and use the affordances in the agent code.

Thing Explorer This module is an interface to the TD Repository, it provides an easy way for the user to test the affordances masking the details of composing the right request to the Thing itself behind forms and buttons generated from the description.

Web IDE This module allows the user to program agents through a visual language and submit them for execution. Since the technological lock was already imposed by the requirements this was designed to be a web application.

Storage Manager This module is responsible for the persistence of the user-created agents' code and of the designed runtime configurations specifying which agents to execute together in a multi-agent system.

Runtime Orchestrator This module is an optional module that schedules the execution of different runtime configurations each time on a new Runtime Environment. Although this was not in the requirements when it came to planning the support for execution of multi-agent systems the possibility of having multiple separated systems running was considered an interesting feature to add.

Runtime Environment This module executes a multi-agent system given the generated source code and a runtime configuration specifying how many agents needs to exist in the system. The environment is also extended to natively support agent to thing interactions.

A simulation environment, the **WoT Simulator and Proxy** was designed as well. It supports testing of the agent system on a simulated smart environment both for development purposes and for checking the soundness of the programmed agents before deploying in a real-world scenario. This component is outside of the project scope itself but complements the solution providing a much-needed testing option when dealing with programming smart environments.

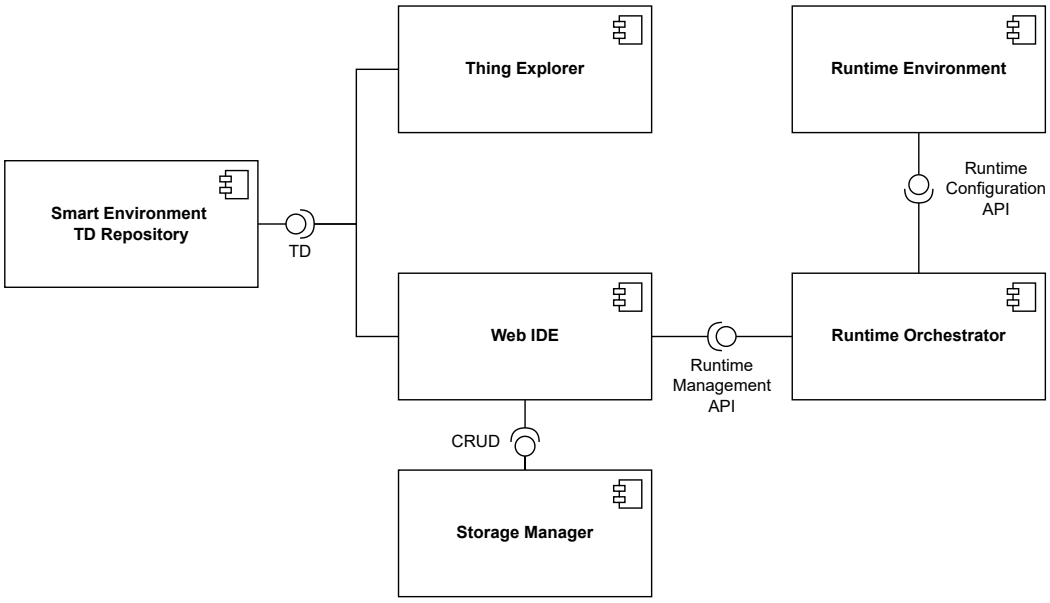


Figure 4.2: Components and their relationships to implement the systems' requirements

As it can be seen from Figure 4.2, the components presented above collaborate to implement a comprehensive system implementing the requirements: the **Smart Environment TD Repository** exposes an interface to serve Thing Descriptions to the rest of the system. The **Thing Explorer** consumes the descriptions to generate a user-friendly interface, whereas the **Web IDE** does the same to provide programming constructs that users can insert in the agent visual language.

The **Web IDE** persists the defined agents through the CRUD (Create Read Update Destroy) operations provided by the **Storage Manager** and interact with the **Runtime Orchestrator** to schedule the execution of a multi-agent system using the **Runtime Environment** as the execution platform.

4.2.1 Designing the Agent Runtime Infrastructure

The Runtime Infrastructure design was heavily influenced by the actual target runtime platform which is the JaCaMo platform.

Given that the block language can generate correct agent code, agents could have simply been exported as source files and used in a regular JaCaMo project.

What the Runtime Infrastructure aims at, is to allow for this exchange to happen over the network and submit a configuration specifying which agents

to execute together in a multi-agent system as well. Of course, it also supports natively the extensions needed to allow agents to interact with the WoT.

The planned design involves a two-step process with the configuration first submitted to the Runtime Orchestrator that then can schedule the actual execution to one of the Runtime Environment nodes available.

The inspiration for the runtime configuration came from the *.jcm* file used in the JaCaMo platform to define a multi-agent system. For this example, just names and source reference of the agents to be executed is needed but since the *.jcm* allows defining some fine details about the environment and the agent organization as well this can be later implemented to extend the control and possibilities of configuration.

4.2.2 Simulation Environment Requirements

The Simulation Environment was the latest addition to the project, coming from the desire and need to test the solution independently from the lab setup consisting of real smart objects.

Usually, in these situations, mocked APIs are the quickest solution, they usually are implemented in a "static" way, always replying in the same way and that wouldn't have allowed seeing if the effects of the actions were actually executed triggering changes in the thing state.

Also, not having a comprehensive view of the whole environment would have made the debugging process hard to follow and impossible for different mocked things to behave according to the same environmental constraints. Let's say for example we're building a temperature monitoring agent, if we're mocking a temperature sensor and a heater we would like to see that the action of turning on the heater actually results in a rise in the temperature readings from the sensor, otherwise, the agent might think that the system is failing. This is practically impossible to achieve if the two mocked devices are not placed within the same simulation.

The idea was further developed to realize a companion simulation environment that could act as the deployment environment for the agents built with the visual tool. The simulation environment was envisioned to have a visual interface as well for users to quickly be able to see if the programmed multi-agent system was behaving in the desired way just as they would do by looking at the real things.

Since this component was realized for the immediate needs, no intensive research on the existing literature on the topic was done and a fully customized option was designed and developed instead.

The simulation environment was planned to satisfy the following requirements:

- Offer a simulation environment to test applications that need to operate over web-things;
- Act as a repository for the Thing Descriptions of the simulated things;
- Be easily configurable to support different scenarios with multiple things;
- Display an intuitive graphic interface to debug and observe the behaviour of the system;
- Offer the possibility for customization for a better graphic representation of the things;
- Have a robust infrastructure that allows to simply add the simulation models and the Thing Descriptions to simulate a new thing.

As it can be seen, the main focus was on ease of customization acting like a framework where new user-defined models can be added to simulate different scenarios.

Since it was meant to be a companion application for the Web IDE, the simulator was also planned as a web application so users could keep both applications open on their browser and switch easily between them.

Chapter 5

Development

In this chapter, the development process that brought to the creation of the prototype of the whole system is described.

Technological locks were imposed from the interface side by the requirements to have a light-weight Web application that didn't need installation on the end-user machine, and from the agent infrastructure side by the support offered by the JVM environment with the implementation of JaCaMo as the main framework for executing BDI agents written in the Jason programming language.

The development process was organized to first realize the end-user interface to manage and program agents with the designed block language and later build the runtime environment infrastructure to support the remote execution of multi-agent systems capable to interact with the Web of Things.

Interesting implementation details are discussed in the following sections following the realization of the main components while motivating the choices of the enabling technologies.

5.1 Smart Environment TD Repository

To better understand how the system is implemented the best way is to follow the main data flow between components, starting with the source which is the Smart Environment TD Repository.

This component was the first present in the system since it was based on the already available implementation of a discovery service for Thing Descriptions developed by the team in St. Gallen.

The project, whose name Yggdrasil comes from the mythological tree of life, aims at providing uniform interaction among heterogeneous agents by proposing the notion of *Hypermedia MAS*[10].

The idea is to model an agent environment based on the Agents & Artifacts metamodel through hypermedia and standard Web technologies, achieving scalability and uniform access to resources in the environment.

The API exposed by Yggdrasil helps navigate an agent environment and all the defined workspaces where artifacts and their operations are described with the Thing Description standard since the model perfectly overlaps with the one of A&A.

This allows to present both digital artifacts and actual things under the same interface and the service can act as a discovery platform for both.

Some work was initially done to extend the capability of Yggdrasil of generating Thing Descriptions and serving them in JSON-LD 1.1 format to be better consumable in Web applications.

The Smart Environment TD Repository doesn't necessarily need to be an instance of Yggdrasil though, since Yggdrasil itself was exploited for this functionality out of its many features.

The WoT Simulator developed for testing purposes acts in fact as a TD servient as well, replacing Yggdrasil for a lighter deployment setting.

The main point is that any service capable of exposing Thing Descriptions is ok as long as it does still keep them organized using the structure of the Agent & Artifacts metamodel made of environments and workspaces since this was used as a constraint in the front-end applications namely the Thing Explorer and the Web IDE to keep things neatly organized and provide the concept of scope for agents interacting with them.

5.2 Creating the Web User Interface

The UI is what target users of the system will see and interact with. Of course, this is one of the most important parts of the system itself given that the main goal is to directly empower people to use the platform easily for their needs.

Users should have a clear idea of what each component of the interface do without any detailed explanation and the interface should allow them to understand and use the tools they have available to create their solution through agent programming.

Two main use-cases were identified when designing the UI:

- Observing and testing the smart things' behaviour in reaction to commands sent through the Thing Description interface;

- Coding to replicate the desired behaviour of the smart things.

These two phases are of course not necessarily sequential but can be repeated in multiple iterations when writing an agent program so the idea was to model two different interfaces to support each use case in parallel.

The Thing Explorer is the component of the system supporting discovery and testing of the capabilities offered by the different smart things whereas the Web IDE is the one providing access to the block language and runtime management of multi-agent systems.

Both are implemented as Web pages served by a Node.js server. The whole Web application that implements the system UI is developed using standard Web technologies without the aid of modern front-end frameworks.

There is no strong motivation on not using frameworks other than the implementation of the Thing Explorer started on top of a simple JavaScript-based prototype and there was never the need to refactor the whole infrastructure since the application is self-contained with a small number of pages and reusable components.

5.2.1 Thing Explorer

The Thing Explorer is the component of the system that helps users navigate the environment modelled in the TD Repository. As stated before this is crucial since users need to be aware of what are the things available for interaction and which affordances they expose to develop a solution to their domain problems.

The explorer is implemented as a Web interface showing on one side all the available things in a workspace and revealing a list of affordances, separated by kind, when clicking on a thing.

Each affordance is parsed from the Thing Description, generating a graphic component that shows the description of the affordance itself and allows to test it out by sending the correct request to the thing at the press of a button. To compose the request, if parameters are needed like in the case of actions, an HTML form is used to input the required values. The form is interpreted by composing the syntactically correct JSON object to send as the body of the request following the schema defined in the TD.

Results of requests to the thing are displayed as raw JSON messages in a notification-like component to signify that the thing is answering the request.

The interface is clean and simple since this system component is designed to be easily accessible and should require no instructions to be used. The tool is thought to be complementary to the Web IDE, the same colour scheme is used to help users familiarize themselves with the different kinds of affordances. The resulting interface can be seen in Figure 5.1

The explorer was seen as a necessary extension to deal with the complexity

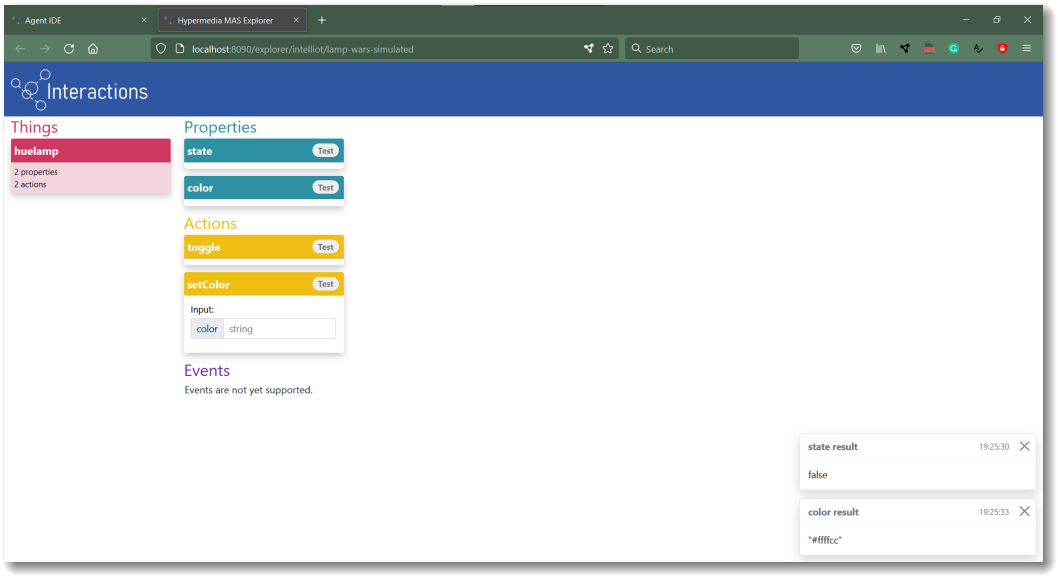


Figure 5.1: With the Thing Explorer UI, users can see available things, their affordances and test them out, results are displayed on the bottom right as notification pop-ups.

of programming smart things with a non-trivial behaviour. It was proven quite useful when preparing demos of the whole system with the devices available in the laboratory such as robot arms or mobile robots. Being able to test out some affordances and identify the right parameters before putting them into the agent code, reduces the time needed to create a working solution and fine-tune the values to be sent to the machine.

5.2.2 Web IDE

The Web IDE is the core component since it is the interface to use the visual agent language, manage the library of saved agents and schedule the execution of multi-agent systems.

The goal of creating a user-friendly Integrated Development Environment (IDE) comes from the advantages that such systems give to programmers when designing their solutions. The interface should then offer a comprehensive view of the code, give the programmers tools that can be used to produce code more efficiently and support them in the execution and deployment.

One key feature of traditional IDEs that here is missing is debugging support which was left to future works since the infrastructure was missing to achieve it and to provide a good visual explanation of the agent's decision process is definitely a challenge worth another project.

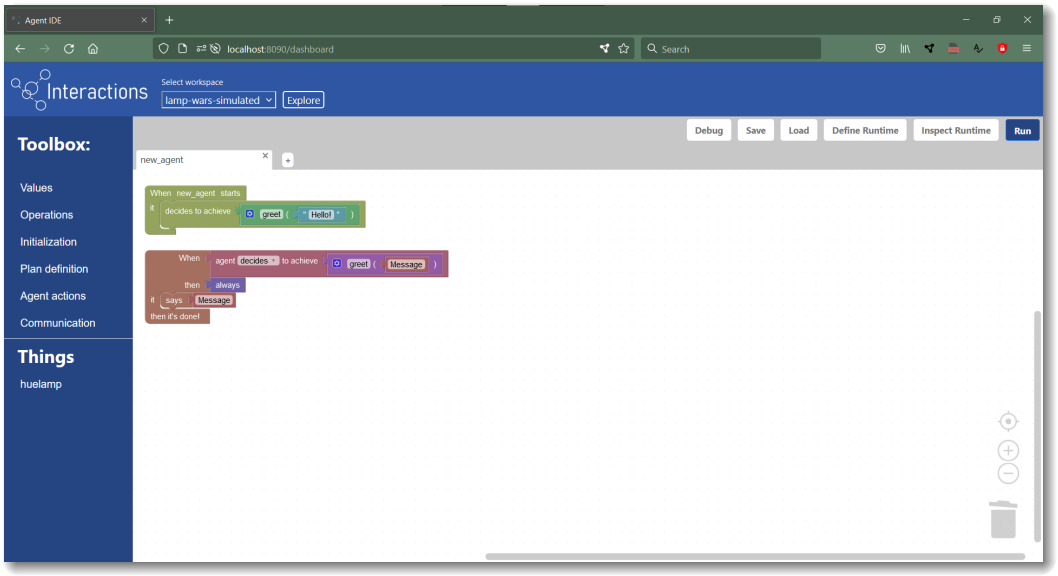


Figure 5.2: The Web IDE - on the left the toolbox, in the centre the canvas to place blocks and on the top right the buttons to manage agents and execution.

The final interface can be seen in Figure 5.2. In the following sections, different features of the tool are presented to show how they were implemented and integrated towards the goal of assisting users in the development cycle.

Development tools

The structure used by the TD Repository of environments and workspaces to group things is used by the IDE to give *scope* only to a specific workspace when programming agents. The first thing the user has to do is select which workspace to load so that TDs can be parsed and blocks related to such things can be generated. This limitation is purely meant for the manageability of the number of things visible to the agent, it's up to the designer of the repository to organize things efficiently so that all the desired interactions can take place.

All the tools concerning the visual language are already offered by the library of choice to implement it which is Blockly. The library not only supports the definition of custom blocks but comes with a development environment styled as an infinite *canvas* to drag blocks in.

In the canvas, blocks can be moved, manipulated, deleted and composed to create programs. The syntactical check generally offered by an IDE is a consequence of the usage of blocks as visual abstraction since each block defines to which other it can be linked allowing only correct compositions.

The canvas is complemented by a *toolbox*, a menu where blocks (or pre-

aggregated block structures) can be organized in categories, for users to easily access them and use them in the canvas.

In total six categories were defined to group blocks up:

- **Values** is the category containing blocks for raw values that can be used in combination with other blocks such as atoms, strings, numbers, booleans and variables;

- **Operations** is the category for operations between values such as math operations and boolean expressions;

- **Initialization** is the category for blocks that can be used to define the initial knowledge of the agent in the form of beliefs, goals and rules. Note that the root block for the "initialization plan" is not present since only one can exist and it's generated by default when a new agent file is created and can never be deleted;

- **Plan definition** is the category containing blocks for defining plans given their triggering events and context;

- **Agent actions** contains all the actions available to be used in plans, can be extended to support more of the internal actions that Jason supports;

- **Communication** contains some actions that are related to agent-to-agent communication and thus separated since they are not needed if the user is programming an agent that does not need to directly interact with other agents.

Additionally, a category is dynamically created for each smart thing in the workspace to easily group and distinguish blocks related to the use of affordances on different things.

On top of the functionalities offered by Blockly, some additional control was given to the user by enriching the interface.

As any typical IDE, a tab-bar to keep multiple *files* open at the same time was added alongside the possibility of defining a name for the agent source file. Since there can be only one canvas in each Blockly instance, this is actually implemented by removing all the blocks on the canvas when the user selects a tab and load the corresponding ones from an in-memory data structure where blocks are temporarily stored.

A button grid is also displayed on top of the canvas to manage the storage and execution of agents.

Persistent storage

Agents can be saved persistently to be later recovered and edited. This of course is a fundamental feature to have in any IDE since writing code is generally an iterative process that could require the user to go back and edit some parts of an agent program either for bug-fixing or evolving the behaviour.

Since the agent visual programming environment is developed as a Web application, central server-side storage was seen as the most sensible option. The **Storage Manager** component inside the Web IDE backend server implements this functionality. For the time being, it does not keep separate libraries for multiple users, but, for a production environment, some form of authentication might be needed to avoid mixing agents created by different people.

The central storage uses a MongoDB database to store both the generated Jason code and the XML-based serialization used by Blockly to describe the block structure so that the visualization can be quickly regenerated on the Web interface.

The interaction with the storage is enabled by *Save* and *Load* buttons on the button grid on top of the canvas.

Runtime Management

After users have developed one or more agents they might want to deploy them. Execution management is usually a feature of IDEs so the Web IDE must support users in the process of actually submitting a multi-agent system configuration to the **Runtime Environment**.

Since for the time being the configuration is simply a list of agent names and their corresponding sources, when pressing the *Define Runtime* button users are prompted with a simple interface to pick from the library of agents that they saved on the persistent storage and choose which one they want to be executed in the same multi-agent system.

A configuration must be then persistently saved with a name and later submitted for execution on the actual Runtime Environment using the *Run* button.

When a multi-agent system is running, users may like to get some information about which agents are deployed and possibly edit them. This is done with the *Inspect Runtime* button that shows the user a menu with all the multi-agent systems currently running and can provide information about which agents are present in them. Users can dynamically add or remove agents selecting a name and a template from the agent library.

This allows to edit the behaviour of an agent, and update it by removing only agents with that source without interfering with the running instances of

other agents in the system. In future, it might be interesting to provide this update functionality directly over running instances of agents by swapping the plan library with the new one in order to keep the agents *alive* and thus maintain their mental state.

Code generation

The core functionality of the Web IDE is of course generating valid in the Jason syntax. Once the design phase of the different blocks to map the concepts and abstractions available in Jason to develop agents was completed, the actual code generation was implemented in the IDE once again thanks to the support of Blockly.

Since the library offers the possibility to define custom blocks it is also possible to develop custom code generators. This is done by defining functions that convert each type of block into a string. The approach resembles one of a classical parser but as if the parsing is already done by the actual block structure itself. This leads to a very clean definition of how blocks connections should be interpreted when composing the actual code, delegating the responsibility of the representation to each singular unit.

When defining blocks is possible to include what are called *mutations* in Blockly, which are values embedded in the block to keep extra information that is not necessarily visible in the graphic representation. This mechanism was heavily used to generate affordances blocks and store values needed in the code generation phase without showing the users technical details such as affordances URIs.

5.3 Implementing the Runtime Infrastructure

Through the visual language, users can build agents and generate valid Jason source code that could be exported and used in any Jason application. The goal of the project though was to not only support people in the building phase but also in the execution phase like any modern IDE would do.

The visual language is also managing interaction with the Web of Things and thus requires a specific runtime environment properly configured and equipped with the enabling tools.

On top of all these requirements, since the IDE interface is Web-based the execution environment was planned to be accessible with standard Web technologies such as REST APIs as well. The **Runtime Environment** is then implemented as a Web Server, exposing APIs to receive both agent source code and runtime configurations to start the execution and to later stop it at any given moment.

When designing the overall system, a level of indirection was added to support the execution of possibly more than one multi-agent system at the same time. The Web IDE communicates with the so-called **Runtime Orchestrator** that deals with the concrete rerouting of requests to the Runtime Environments nodes.

Although planned, for simplicity, this last feature was not implemented and the orchestrator is directly rerouting requests to a single known runtime environment node.

It is worth noting that in building the runtime infrastructure the goal was to have immediate support for the execution and not to create a solution to last in the long period. One of the most critical aspects that emerged is that the runtime environment where agents are deployed is completely ignorant of the existence of the smart things whereas it might be interesting to reflect the structure used in the TD Repository in the environment as well. Further work on the topic is already in the planning working on top of the previously cited Yggdrasil project.

Anyway, since this was not the main scope of the project the proposed infrastructure is a valid solution to work in a concrete scenario even if the integration with the Web of Things is treated as with any other external entity instead of in a more inclusive way.

5.3.1 Wrapping JaCaMo

As stated before, JaCaMo is the reference execution platform for Jason agents. When building the Runtime Environment the first task was to find an efficient way to wrap JaCaMo in a Web Server application that could add the API logic needed to be interoperable with the rest of the system.

The integration of JaCaMo with REST has been already studied by the original creators of the platform in a recent paper[1] that brought to the creation of JaCaMo-REST: a resource-oriented Web-based abstraction for the multi-agent programming platform JaCaMo implemented in Java.

In its essence, JaCaMo-REST is a JaCaMo application that directly exposes APIs to modify all the relevant entities in a multi-agent system from agents to artifacts and organizations. The idea at the core level is to have a multi-agent system always running and allow an external application to edit which agents are present, the environment as well as directly change the behaviour of agents by communicating with them or injecting new plans.

Since the requirements of the Runtime Environment planned for the system were different JaCaMo-REST was wrapped and slightly modified to work as needed.

The resulting implementation architecture is shown in Figure 5.3. The

architecture was planned to be JaCaMo independent, so that it would be possible to change the agent execution platform. A newly created REST server implemented with Vert.x in Java exposes the following endpoints to manage the execution of a multi-agent system:

- GET `/agents` to retrieve all the agent sources saved on the filesystem;
- GET `/agents/:id` to retrieve a specific agent source by its id which is the name of the source file;
- PUT `/agents/:id` to update the source file of an agent;
- GET `/runtime` to get information about the running MAS;
- POST `/runtime` to submit a new configuration and start the execution of the MAS;
- DELETE `/runtime` to stop the current executing MAS;
- GET `/runtime/agents` to get the list of agents currently running in the MAS with their names and source templates;
- POST `/runtime/agents` to create and add a new agent to the currently executing MAS;
- GET `/runtime/agents/:name` to get information about a specific agent running identified by its name in the currently executing MAS;
- DELETE `/runtime/agents/:name` to remove an agent from the currently executing MAS;

The `MasManager` implements the business logic used by the API controllers through delegation to a `PersistenceManager` and a `RuntimeManager`.

Agents and configurations received through the API are converted to the right kind of files for the JaCaMo platform and saved persistently on the file system that acts as a shared memory between the Web server and the JaCaMo-REST instance. The latter is executed as a subprocess managed by the `RuntimeManager` that starts it when needed passing as an argument the path of the *.jcm* file with the configuration that needs to be loaded.

Communication with the running instance of JaCaMo-REST is achieved using the APIs directly exposed by the process over the local network through the `MasBridge`. This allows for example to kill running agents and add new ones. For this last feature in particular, the original API was changed to behave so that when the name of the source file to be used as a template for

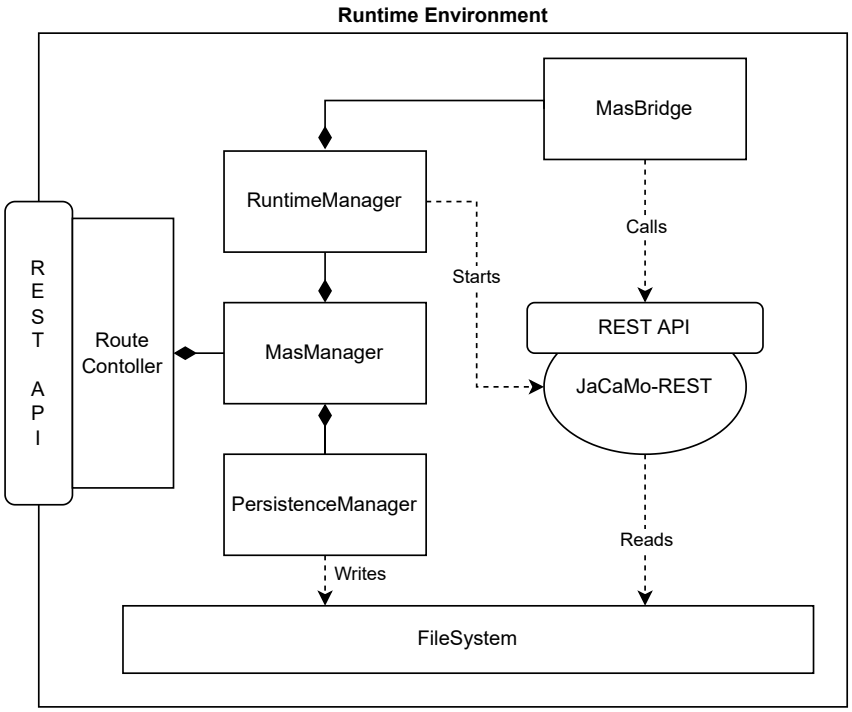


Figure 5.3: Architectural diagram of the wrapper built around JaCaMo-REST

a new agent is passed as a parameter to the API that agent is instantiated since the original implementation always deployed agents based on an empty template expecting plans, goals and beliefs to be sent later and fill the agent through the APIs.

Again, this was a prototypal implementation, further insight might be gained in integrating JaCaMo-REST by maybe expanding its API, but since the time was limited it was easier to use it sort of like a black-box and wrap it in a custom made interface for better compatibility.

5.3.2 Support for Agent to Thing Interaction

The very last piece of the puzzle in building a runtime infrastructure for the WebIDE was the integration of the Web of Things inside the agent environment.

Both Jason and JaCaMo were built knowing that, depending on the specific application, developers might have needed additional control over what agents can achieve especially when interacting with external services and systems so there is an already built-in mechanism to expand the capabilities of the multi-agent system. This can be done either by creating new internal actions of the

Jason language using Java or by defining CArtAgO artifacts (once again in Java) that agents can use.

The two methods, although quite similar in practice, have a substantial semantic difference that was considered when choosing how to support proper interaction with things. Actions can be seen as something that an agent can do, they are *internal* by definition thus they are not shared between agents and they can only edit the agent state. Artifacts, on the other hand, can be seen as tools providing functionalities to an agent, they have a life cycle, internal state and they can be shared among agents in the same environment so an action invocation on an artifact can modify the environment for all the other agents.

For these reasons both methods were used to realize the integration with the Web of Things, in particular, a `WoTHttpClientArtifact` was defined providing functionalities for agents to invoke affordances on things and retrieve the result as a JSON object. Internal actions were used instead to add JSON parsing and assembling capabilities to agents.

WoT HTTP Client Artifact

Since the things considered for this project scenario operates over HTTP, agents need a way to make HTTP calls to communicate with them. At first, this was considered a capability of an agent, so the idea was to implement it with a Jason Internal Action.

Web of Things though can use authentication flows to implement security and access control to the thing's internal state and actions. This would have meant that agents would have needed to constantly remember to use the proper authentication header when making requests to each thing. As human users, we are not accustomed to this, since usually Web clients are designed to make this task easier by asking the user to authenticate once and then remembering his identity.

Since agents should not be dissimilar from people in their interaction with external services, the implementation of this flow was achieved with a Web client artifact. In this way, agents could authenticate first for a given thing and then use it seamlessly through the instance of the client. The abstraction of an Artifact is then the most appropriate since the client has a life cycle and some internal state concerning authentication.

As it can be seen in Listing 5.1 the artifact is implemented extending the base CArtAgO class and it provides operations to set authorization, read properties and invoke actions both with or without sending input. The operations match the semantics of the Thing Description model, they accept the URL of the affordance and return the result as a serialized JSON payload. The seri-

alization is needed for compatibility with Internal Actions that need to parse the fields and convert them into values usable within the agent code.

The intended use of the artifact is for each agent to instantiate a client for each Web thing it needs to interact with. Although artifacts are effectively shared among agents, this simplifies the management and each agent may have its own method of authentication with the thing. Of course, having shared authenticated clients could open the possibility of malicious behaviour, but for the purpose of this implementation, we assume to have a multi-agent system where trust is ensured.

```
public class WotHttpClientArtifact extends Artifact{

    private WotHttpClient client;
    private String name;

    void init(String name) {
        this.name = name;
        this.client = new BasicHttpClient(name);
    }

    @OPERATION
    void authorizeWithKey(String location, String tokenName, String
        tokenValue){
        TokenLocation tokenLocation =
            TokenLocation.valueOf(location.toUpperCase());
        this.client = new StringTokenAuthenticatedHttpClient(this.name,
            tokenLocation, tokenName, tokenValue);
    }

    @OPERATION
    void readProperty(String url, OpFeedbackParam<String> result) {
        try {
            result.set(client.readProperty(url).toString());
        } catch (WotClientException e) {
            failed(e.getMessage());
        }
    }

    @OPERATION
    void invokeAction(String url, String method, JsonElement obj,
        OpFeedbackParam<String> result) {
        try {
            result.set(client.invokeAction(url, method, obj).toString());
        } catch (WotClientException e) {
            failed(e.getMessage());
        }
    }
```

```
    @OPERATION
    void invokeAction(String url, String method, OpFeedbackParam<String>
        result) {
        try {
            result.set(client.invokeAction(url, method, null).toString());
        } catch (WotClientException e) {
            failed(e.getMessage());
        }
    }
}
```

Listing 5.1: The Java code implementing the WoTHttpClientArtifact

The management of artifacts is seamless to the user that does not need to know the internal details of the implementation when using the visual language. To this purpose, custom shared plans were developed and appended to each agent so that they are able to manage the creation of the artifact.

In Listing 5.2 the two test-goal triggered plans for the creation of a client are shown. The first one checks in the plan's context if the user has provided some form of authentication for that thing. The authentication information has to be set as a belief of the agent before invoking affordances and it is used to directly authenticate the client artifact upon creation.

If no authentication is provided the second fallback plan is executed. Both plans save the artifact ID as a belief for the agent for it to be retrieved when needed since the instruction `?xx_wot_client(Thing, ID)` is prepended to each affordance invocation so that the artifact is lazily created when needed for the first time and later its ID is retrieved to be used again.

```
+?xx_wot_client(Thing, ID) : x_thing_login(Thing, Scheme, Location,
    KeyName, Value)
<- println("Thing with login");
    .my_name(N);
    .concat(N, "_", Thing, "_client", A);
    makeArtifact(A,"wot.WotHttpClientArtifact",[N],ID);
    authorizeWithKey(Location, KeyName, Value)[artifact_id(ID)];
    +xx_wot_client(Thing, ID).

+?xx_wot_client(Thing, ID) : true
<- .my_name(N);
    .concat(N, "_", Thing, "_client", A);
    makeArtifact(A,"wot.WotHttpClientArtifact",[N],ID);
    +xx_wot_client(Thing, ID).
```

Listing 5.2: Shared plans appended to each agent source file to manage Artifacts

Internal Actions for JSON management

To make the integration with the Web of Things complete, agents need to be able to extract information from the JSON result of the invocation of an affordance and use it within the agent code. They also need to be able to assemble JSON objects to be used as inputs for affordances that require so.

These JSON manipulation capabilities are best modelled as Internal Actions since they are part of the manipulation that an agent does without the need for any interaction with the environment. Actions in Jason can interact with regular Java Object-Oriented code to do any sort of computation and return values wrapped within the classes that represent Jason's valid terms. Object references as treated as terms and allow to share Java objects between different actions.

In total seven actions were developed as a library for agents to use: the notation used below with the angle brackets indicates the so-called *feedback parameter*. Since Jason's actions need to have always only a boolean return value indicating if the plan can continue or the action has failed, this is the intended method of returning the functional result of an action and the feedback parameter must be an unbound variable that gets grounded with the return value.

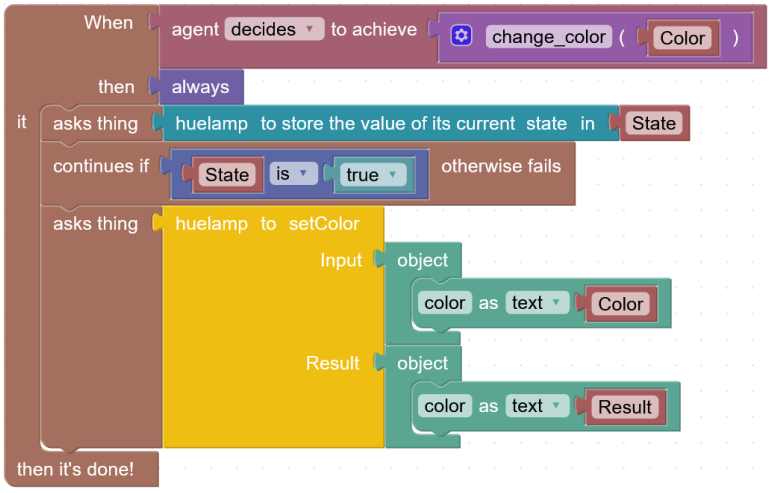
Here are the actions for JSON management, again the idea was to create simple "atomic" actions that could express all the required functionalities by composition:

- `json.create_empty_object(<Obj>)` to create an empty object that can be later filled with fields;
- `json.create_empty_array(<Obj>)` to create an empty array that can be later filled;
- `json.parse(string, <Obj>)` to create a JSON object, array or value from its string representation (needed to interact with the client artifact);
- `json.get(obj, type, key, <Value>)` extracts a value from a json object if the key is a string otherwise an element from an array if the key is a number;
- `json.get(obj, type, <Value>)` extract the typed value in agent terms from a JSON primitive value;
- `json.set(obj, type, key, value)` insert a new key or change the value of an existing one. As before if the key is a string works with object otherwise with arrays;

- `json.print(prefix, obj)` print the JSON object to the agent console for debug purposes.

As it can be seen, type information is needed when extracting the value from the JSON structure in order to perform the correct cast.

Again JSON management is seamless to the user using the visual language, all the necessary parsing logic is automatically generated from the blocks given the data coming from the input and output schema of the affordance as shown in figure 5.4.



(a) A plan using two affordances on a lamp thing.

```
+!change_color(Color) : true
<- ?xx_wot_client("huelamp",X_var_1);
    readProperty("http://localhost/huelamp/state",
        X_var_2)[artifact_id(X_var_1)];
    json.parse(X_var_2, X_var_3);
    json.get(X_var_3, "boolean", State);
    (State == true);
    json.create_empty_object(X_var_4);
    json.set(X_var_4,"string", "color", Color);
    ?xx_wot_client("huelamp", X_var_5);
    invokeAction("http://localhost/huelamp/color","POST", X_var_4,
        X_var_6)[artifact_id(X_var_5)];
    json.parse(X_var_6, X_var_7);
    json.get(X_var_7, "string", "color", Result).
```

(b) The code generated from the blocks above

Figure 5.4: The integration with the Web of Things in the block language and its corresponding translation in Jason code.

5.4 WoT Simulation and Proxy Environment

The Simulation Environment was developed as a supporting system to the main one to give the ability to test agents onto simulated smart things environments.

As stated before the main features desired for this system were ease of configuration and a simple yet effective graphic interface to show the state of the simulated things. On top of that, for the needs of the user study, the Simulation Environment was also extended to work as a proxy, mapping an application-level Thing Description to the raw APIs of a thing, allowing for more interesting and complex affordances to be exposed for the users (and their agents) to use.

To accomplish these requirements a Node.js server application was built, to work at a fast pace and produce a working prototype in the least amount of time. For future investigation on this kind of system, a typed language would be better since Object Oriented languages are usually better suited to build simulations due to their expressiveness in modelling objects and their behaviour.

To simulate a thing, its Thing Description must be defined and saved in a folder. The application then parses the TD and exposes both the TD itself and all the endpoints corresponding to the affordances. The TD must declare in the `@type` field the name of the class implementing the simulation model.

All the simulation objects must extend a `ThingWrapper` abstract class and implement two methods that map the property and action affordances to the actual state changes on the simulation model. When a request is sent to the affordance URL the application finds the simulation object implementing that thing by matching its unique id, it then invokes the mapping methods passing any optional input data and waits for a result to be returned after the thing logic is applied. A fixed tick simulation is also run to emulate time passing in the system, meaning that at each update a thing can change its internal state without having this triggered by an affordance (e.g. a smart light bulb might overheat with time if kept on).

Every tick a thing might push its new state on a WebSocket to keep the front-end application updated with the current simulation state. Invocation of affordances is also logged which makes it possible to track the behaviour of agents interacting with the system and have some form of basic debugging.

If a thing logic implements remote calls to a smart object API the simulated thing is acting as a proxy, this is extremely powerful since it allows to both test an application over hybrid systems when not all the things are available and because, as stated before, it might be necessary to give the agents a higher-level set of operations over a thing and this system can act as a way of prototype

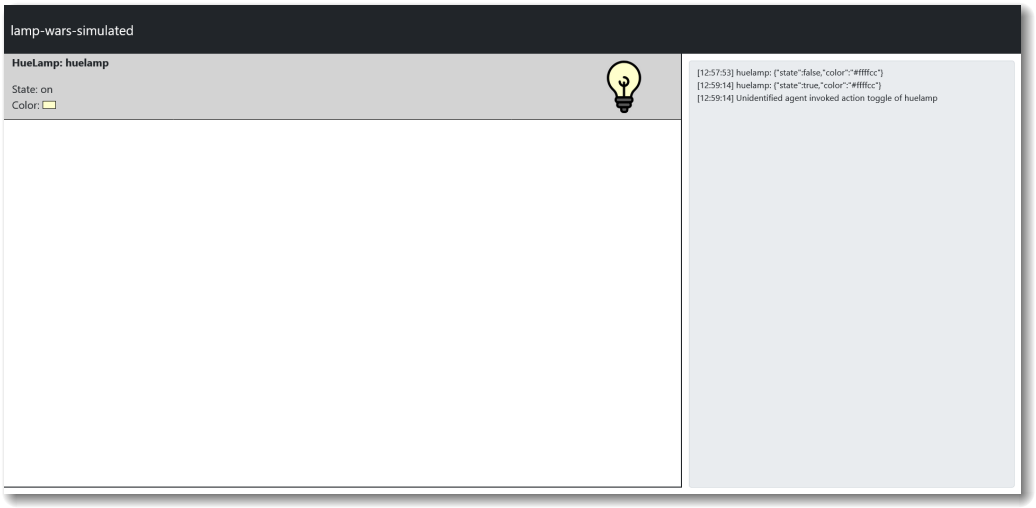


Figure 5.5: The front-end application showing a simulated smart lamp.

and implement that as well. This mechanism was used in the user study since users were asked to control mobile robots and they were programmed to have a basic routing algorithm at the proxy level since it was easier than program that at the machine level.

Finally, to make the simulation come full-circle, the notion of the environment is present in the simulated world. Things can get a reference to an environment object and use it to either read its state or act to change it. This is essential to implement a simulation that let things behave coherently within it.

To make things interesting, the environment is implemented just as any other thing, exposing a thing description that must declare the `@type` *Environment* thus it can be accessed by the external to give the user full control over it. The environment simulation object is instanced before the others and then passed as a reference to all the simulation objects that can use them in their logic. This opens the possibility for simulating scenarios with multiple things having a cause-effect relationship in their behaviour through interaction with the environment such as for example a heater and a temperature sensor.

The front-end application for the Simulation Environment was developed to be just a basic stub, it receives updates through a WebSockets with all the things' state and is able to show it (Figure 5.5). Custom components can be defined that interpret the state changes and show them on a graphic interface. Since just a basic interface was needed this was not developed intensively but there is potential for an interesting future project investigating how to give proper support for users to define their components effectively and support complex rendering of the simulation state.

Chapter 6

Evaluation

After developing a working draft of the whole system. In-house testing proved that the solution was robust enough to allow new users to try it out with real users in a controlled environment.

Since the system was always built having in mind its target purpose of enabling people without proper training in computer science to develop their solutions in automation scenarios, a preliminary evaluation study could highlight the potential of the tool and gain feedback on the current implementation to improve it for future works.

This led to the definition of a user study carried out during a week when people with little or zero programming experience came for the first time in contact with the tool and tried to accomplish simple automation tasks with a few real smart objects available in the University of St. Gallen's laboratory.

In this chapter the choices made for the definition of the study and its routine are explained, then a demographic analysis of the study participants is presented and in the end, relevant outcomes are shown.

6.1 Designing a User Study

The design activity of the user study focused on what was the best way to simulate a situation in which users could need the tool to solve some problems.

The main issue was that given the time available to complete the project it was nearly impossible to find experts and let them test the tool in a domain they know using even complex smart things.

The next best solution was to create a study targeting people with no relevant previous experience in programming and let them approach simple and understandable domains and things. This was the approach towards which the user study was designed.

6.1.1 Evaluation Variables

First of all, it was necessary to define which variables were relevant and thus needed to be tracked during the study itself.

The main goal of the study was to evaluate:

- the **usability** of the overall system;
- the **user-friendliness** of the agent paradigm presented in visual form;

To collect data that could help evaluate qualitatively these aspects, it was decided to ask the users to complete some specific tasks with the tool and measure how long it took for each user to complete them individually within one hour and a half as the maximum time slot.

An audio recording was also kept on since it was interesting to apply the *think-aloud protocol* commonly used for usability testing in this scenario to follow the user thought process and be able to understand better what users struggled with in order to improve it in future iterations of the tool. This was important both for the interface usability and especially for understanding whether people were comfortable with the agent paradigm or not.

To further evaluate more objectively the usability, the System Usability Scale (SUS)[8] was chosen for an after-study survey.

Since there was interest in understanding how easily people with no experience could grasp the concepts of agent-oriented programming, little to no training was given to users. A one-page explanation of the basics of what is an agent was given for them to read and then they were shown a three minutes long tutorial video presenting the interface and its features before starting the actual study.

6.1.2 Task Design

Users were given different tasks to complete that were timed individually. In total, five tasks were designed to be solved by users.

Since, as stated before, no assumption could be made about the knowledge of users about any domain, tasks were placed in very simple domains that users could easily relate to. To provide all the necessary contextual information relevant to understanding the domain, tasks were grouped into scenarios: each scenario presented the smart things available to use and the general situation that users needed to imagine to solve problems.

The smart things themselves were designed to be as simple as possible, with very basic actions and properties and all relevant to the task to not confuse the users with an abundance of operations not useful to complete the task goal. In order to do that, the implemented simulator was used to mask under

an "application-level" Thing Description the actual controls to be sent to the device acting as a logic controller and allowing to have the desired high-level behaviour.

Finally, task descriptions were given as descriptions of people's behaviour, as if the agent to be programmed by the user was a person that needed instructions to complete the task. This was done to have people less worried about the idea of programming and more focused on the behaviour they wanted to achieve since the text on the block language was designed to create a narrative flow describing the agent behaviour.

Below the finalized scenarios and their tasks are presented, highlighting how each task was chosen to test a specific interaction between the users and the visual language.

Scenario 1 - Lamp In the first scenario, designed to make users familiarize themselves with the interface and the basics of agent programming, users were asked to imagine having their agents living in a shared flat and controlling a smart lamp for their living room.

The lamp has two properties:

- **state** is a true/false value that indicates if the lamp is on
- **color** is a text value with a color code (e.g. #FF0000 is red)

The lamp has two actions:

- **toggle** switches the state of the lamp from `true`(on) to `false`(off) and vice versa.
- **setColor** changes the color of the lamp by putting as input the color code.

In **task one** users were asked to have an agent continuously check the lamp state and switch it on whenever it found it was off, then wait for a few seconds and repeat. This very basic example already introduced most of the basic concepts needed to develop a solution which are: agent-to-thing interactions, flow control (the agent switch on the light only if it's off since it can only toggle), and loops.

Task two was a modification of task one asking users to achieve the opposite behaviour of having the light always off. This task was designed to have users analyze their solution, understand the control flow and modify it to achieve a different result.

In **task three**, the last in this scenario, users were asked to have an agent looping through the colours of the rainbow. This task was interested in seeing

if users could understand how to use the "memory management" features of agent programming to create the optimal solution to a rather repetitive task. It was also the first task involving sending input to a smart thing.

Scenario 2 - Farming In the second scenario, designed to immerse users in a more specific domain with more complex things, users were asked to imagine having their agents managing a farm using a tractor. The tractors were simulated by mobile robots that could move in a three-by-three grid of fields and each field was identified with a number from 0 to 8.

The tractor has two properties:

- **position** returns the number of the field in which the tractor is;
- **direction** returns the direction the tractor is facing as a number from 0 to 3 since the tractor rotates only 90 degrees.

The tractor has three actions:

- **goHome** tells the tractor to go to its designated home spot outside of the field grid, it returns the time needed by the tractor to complete the operation;
- **moveToField** tells the tractor to move to a field specified as input giving its number, it returns the time needed by the tractor to complete the operation;
- **checkSoilHumidity** checks the soil humidity and returns a value, a colour sensor was used to check whether a piece of coloured paper was present on the field.

In **task one** users were asked to move the tractor in each field and check the soil condition. They were also told to make the agent remember the humidity of each field as it goes by. This task was focused on using the agent beliefs to optimize the solution and using basic programming logic to tell the agent to move each time in the field with the next number. Keeping the humidity value was also interesting to see if users understood how belief worked and that they needed to use some sort of data structure to keep the data and remember which field corresponds to the stored humidity value.

Task two was a modification of task one asking users to, instead of remembering the humidity value, check if it was under a threshold and notify another agent (already implemented). This task was interested in seeing if communication primitives were intuitive and effective as well as seeing if users could find a way to execute plans when beliefs were added.

6.2 Demographic Analysis

Participants for the study were recruited by word-of-mouth and thanks to the help of the Behavioural Lab[1] of the University of St. Gallen.

In total twenty people participated in the study. Before starting the routine they were asked a few questions to understand better their background and see if any correlation emerged with the results.

The main demographic features observed are presented in Table 6.1. As it can be seen the sample is quite representative for gender, age and level of school although since they were recruited through the University most of them were of course students. They did though come from different faculties so there was some variance in that as well.

The most important aspect was about previous programming experience: most of the users answered that they didn't have had any sort of relevant experience and seven out of twenty have never even seen a programming language before. Most of those who had some previous experience usually said that it was an introductory course using Python or R.

It was interesting to have people with very different backgrounds in that sense since it was compelling to see if the previous experience was beneficial for understanding or an obstacle given that the agent paradigm is profoundly different from procedural programming.

Users were also asked to answer a few questions to compute a score measuring their attitude towards using computers in general. The questions were inspired by a study[31] that was aimed at measuring attitude towards computers for teachers and was itself based on the *Computer Attitude Scale*[29]. To keep the survey short though, just a few items were selected.

The questions asked users to evaluate some statements about their experience with computers in their day-to-day life in a range from 1 (fully disagree) to 5 (fully agree). The average score resulted in 3.97 which indicate that users were at ease in interacting with the machine so despite not having experience in programming they were regular computer users which is exactly the target category that was identified when designing the system.

6.3 Study Routine

The full study routine lasted one and a half hours. Participants came one at a time in assigned time slots spanning one week.

The study was performed in a room in the laboratory of the University of St. Gallen, users had a laptop to use the Web IDE and the web service was

[1] https://behaviorallab.unisg.ch/en

Study Participants	
Total	20
Age	11 under 25
	8 between 25 and 50
	1 50 or older
Gender	11 female, 9 male
Level of School	8 high-school or lower
	4 bachelor degree
	8 master degree or above
Current Occupation	13 students
	5 working students
	2 workers
Programming Experience	8 Yes, 12 No
Computer Attitude Score	Mean=3.97
(out of 5)	σ=0.62

Table 6.1: Demographic analysis of the study participants

hosted on a different machine managed by a supervisor. Real smart things were used to have the user see the effect of their actions on the real world, namely a smart RGB lamp and two mobile robots (Figure 6.1).

First, participants were asked to fill in a background data form providing the demographic information and their general attitude towards computers, programming and smart things. The concept of a smart thing is introduced in the form to let the users ask about it if they have never come across the concept before.

As preparation, they were given a written description of the study routine explaining the idea of tasks and scenarios. They were also asked to try to *think aloud* in order to understand their reasoning process during the execution of tasks. A one-page long description of "what is an agent" explaining basic agent concepts by making similarities with people's behaviour was also given to read and consult even during the task execution.

They were then shown a three minutes long tutorial video showcasing the platform and its functionalities and controls. They were shown how to move blocks around and stick them together and they were given a rough explanation of a hello world agent that is generated as a stub whenever a new "file" is created like it's usually done in IDEs.

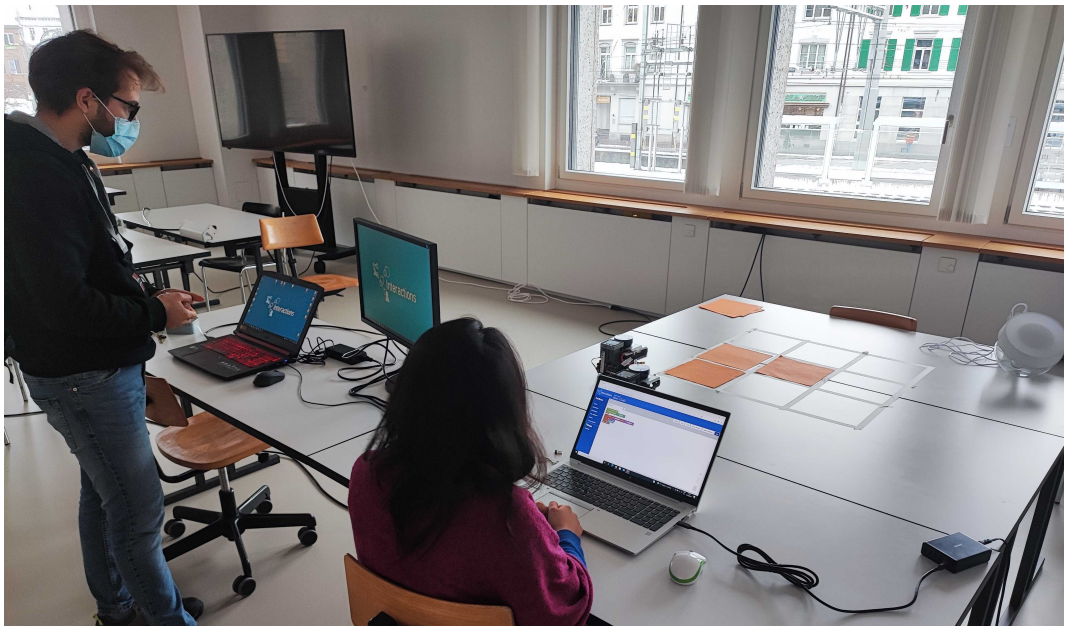


Figure 6.1: The user-study setup.

After the preparation was completed users were shown the scenario first, the description of the smart thing to be used and then one task at a time on a monitor managed by the supervisor. The audio registration and stopwatch were started for each task.

Users were free to ask questions to the supervisor, who could answer without guiding them to the solution. Whenever they felt ready to submit their solution the time recording was stopped and the solution run by the supervisor. If a partial solution was submitted the supervisor tried to point out what was missing by making the user observe the thing behaviour and asked to continue by restarting the stopwatch.

Sometimes users were moved to the next task even if the solution was not fully completed. This was done to avoid excessive frustration. All the users followed the same sequence of tasks since they were organized with progressing difficulty.

At the end of the study, users were asked to fill out another form evaluating each task difficulty level (on a scale from 1 to 5) and the usability of the system using the SUS.

6.4 Qualitative Outcomes

Although the study was not conducted with a strict approach, some interesting qualitative results can be derived by observing the measurements taken while users completed tasks. Results are summarized in Table 6.2 and discussed below.

Study Results			
Task	**Mean difficulty score**	**Mean time**	**Participants**
Task 1.1	4.36	25m 41s	20
Task 1.2	2.88	7m 32s	18
Task 1.3	3.15	10m 50s	20
Task 2.1	3.07	9m 31s	15
Task 2.2	3.50	6m 13s	5
SUS score	Mean = 73.28 σ=9.05		

Table 6.2: Results of the user study

6.4.1 Task Results Analysis

Most of the users managed to complete three tasks out of five total, mostly due to the fact that users needed a lot of time for the first task to enter in the correct mindset and understand blocks behaviour. This is reflected by both the difficulty score that users attributed to task one and its average time which was significantly higher than the following tasks. In the comments that users gave motivating their scoring the fact that the interface was new and they needed time to get used to it is often mentioned.

Users performed better in task two, demonstrating that they understood their previous solution well enough to be able to change it to modify the behaviour without starting from scratch. Some users did ask if they could edit the previous solution, some others, although preferring to start from a blank canvas, were able to remember the meaning of the blocks they used before and find them quickly.

With task three almost all the users missed the point of creating an efficient solution making use of agent beliefs to remember the sequence of colours and iterate on that. Most of them did implement a solution successfully repeating a lot of blocks but in little time proving autonomy in reaching a working solution with the small experience gained from previous examples. While doing that most of them felt like a “better solution” would have been available somehow,

the ones that did try to achieve it struggled and didn't complete the task because they got lost.

Task four (2.1) was similar to task three, most of the users were able to translate their previous solution, although not efficient, to the new scenario pretty quickly. Most of them failed in using beliefs efficiently and creating a data structure in the agent's mind.

Task five was approached by very few users due to time limitations. The few that did had no trouble identifying the communication primitives and understood the idea of exchanging messages but didn't manage to complete the full task.

6.4.2 General Considerations for Future Evaluations

One of the aspects that were underestimated is the time needed by users to get familiar with the block-based interface. Although they are considered intuitive, they are purposefully built in a way that lets people play and explore the possible combinations. For people that have never used anything similar that can take some time to get used to and discover how blocks work. A new iteration of the study may benefit from having a more in-depth tutorial dedicated to the interface and to block mechanics or giving users some free time to play around with blocks without any objective.

Users also struggled in understanding the idea of interacting through the Web of Things affordances, confusing properties and actions, but we assume that the target users for the system are at least somewhat experienced in that area. To give an example, a lot of users struggled in understanding that for the agent to *see* whether the lamp was on or off they needed to invoke the property affordance and check the returned value. Testing again with a different user group might request some training on these topics to evaluate better just the interaction of users with the agent paradigm.

The hardest agent programming concepts for users to grasp were loops and flow control, especially for people that had some experience with other programming languages that were looking for more familiar constructs (if-else or for loops). Although these constructs are available in Jason and implemented as an extension of AgentSpeak they were left out from this prototype because it was interesting to see if people with no previous training would have been brought to naturally reason in those terms. When asked what they were struggling with when trying to implement a loop, most users answered that they wished for a "repeat the plan" block, but failed to match it with assigning the same goal again to the agent.

For what concerns flow control, that was hard to grasp because it introduced the idea of failing plans. This required a deeper understanding of how

the agent reasoning process works and thus confused users that would have not expected the agent to fail and drop what it was doing but instead just skip some parts of the plan.

The most interesting observation is concerning the use of beliefs. Almost everyone didn't think of using beliefs to optimize the solution and guide the agent's behaviour. This arise the question of whether the abstraction is hard to understand or a better visual representation is needed to make users comprehend how to manage the agent memory. Further investigation should focus on this.

Of course, user feedback was incredibly useful to understand if the text on the blocks was understandable and conveyed the right behaviour. This will need further investigation and refinement but, as a first prototype, it was quite successful.

6.4.3 Final Evaluation

Overall the system was considered usable receiving an average score on the System Usability Scale of 73.3 out of 100. Although the SUS is generally used to compare different systems and individual scores are less meaningful, because of its extensive use an adjective rating scale was defined and the score is considered above *good*[3].

By listening to the user reasoning process it was clear that they were able to understand the general flow and with a little more time and guidance they might have been able to solve even more complex problems. This is further proved by the results of the different tasks that although rising in complexity do not rise in perceived difficulty and time needed to figure out a solution. This might indicate that the system is effective in making users learn about how agents behave and get confidence with the paradigm.

It is worth noting that the best participant was a student with no previous experience in computer science who managed to complete all the tasks in significantly less time than the average, proving that the system can appeal to people that know nothing about programming but approach the task with the right mindset.

Conclusions

Main Contribution

As stated multiple times the main contribution of this thesis work is the application of visual programming techniques to the agent-oriented paradigm in a way that fixes as the ultimate goal the full engineering of multi-agent-based solutions to problems in different WoT powered domains and realized by non-technical domain experts.

This project did not only conceptualize a visual agent language but also the vision for an accessible Integrated Development Environment mixing agent-oriented programming and the Web of Things in a seamless interface for both humans and software agents. It then also developed a prototypal implementation of such a system that was evaluated on a sample of users with promising results to validate the initial assumptions made when defining the system requirements.

In doing so, the development steps were first the analysis of the requirements coming from both the IntellIoT project and the research interest of the collaborating groups. Then the analysis of the available tools in the multi-agent system engineering community brought to the identification of Jason and the JaCaMo platform as the ideal candidates to implement the solution.

An analysis of the Jason language and its fundamentals followed, to identify a credible mapping that could convert the syntax to a more human-understandable set of concepts that were later implemented as blocks through the Blockly library to implement the visual language.

An interface supporting the usage of the block language was developed alongside the realization of a Runtime Environment, wrapping the execution of a JaCaMo multi-agent system under a REST API and building tools for agents to invoke Web of Things affordances and manipulate the returned JSON payloads.

Finally, a Simulation Environment was created to both simulate smart objects and mask raw APIs under an application-level interface for real smart things to help test out the solution, create demos to showcase the functionali-

ties and develop the scenarios needed to make users try the system.

The thesis work was then completed with an evaluation study, bringing a group of users with different backgrounds in contact with the tool for the first time and with little to no explanation of its functionalities and of the agent paradigm in general, to see whether the developed solution was friendly enough to pick up and use and gain feedback to improve it in future iterations.

Overall this project brought to the realization of a usable tool and more importantly to the exploration of new routes to integrate the Web of Things technologies with the agent-oriented software engineering in order to enable autonomous controlling systems for real-world devices in different domains. This of course opened up a lot of potentially interesting challenges to further investigate in future related works.

Open Challenges and Future Work

Working on this project touching different research fields allowed to understand better what it might mean to have them seamlessly integrated. This, though, was not the scope of the thesis work itself since this first exploration was mostly motivated to see whether the combination of these topics might bring interesting results.

From what we were able to see with the development of this project we believe that the hypothesis that the integration of agents and WoT is crucial to building intelligent IoT systems is true, and also doing that by means of a simple visual interface might accomplish the goal of keeping humans always in control of such complex systems.

Of course, a lot can be done to improve the existing solution, first and foremost the development of the visual language is an ongoing process, that requires many iterations to be able to pinpoint the basic concepts of agent programming in such a way to make it easily understandable by any user, finding the right metaphors that convey the agent behaviour and empower people to program them efficiently. A lot of work is still needed in that sense, experimenting with different visual paradigms and going towards the definition of a fully customized solution designed around agent programming can help to understand whether the agent paradigm is truly effective as it theoretically should in bridging the abstraction from machines to people.

Finding an efficient way of providing more experts users with the powerful tools that are present in the Jason language that were purposefully left out from this implementation such as annotations and higher ordering without overwhelming novices might be interesting as well. Finding an appropriate balance in that sense might be challenging, but needed if the platform is to

be used not only by non-technical users but by expert agent programmers as well.

From an agent engineering perspective, the IDE focused on the definition of agents. But as we've seen the agent dimension is just one of the three layers composing every multi-agent system. In order to have a powerful multi-agent system development tool, environment programming and organization programming must be taken into consideration for future expansion.

From a Web of Things perspective instead, event affordances were left out from the scope of the whole system, but they are needed if the tool wants to support the full expressive power of the Thing Description model. Also, the integration with the MAS environment is very basic for the time being.

This last feature is maybe one of the most interesting for future works since having Web of Things fully integrated into the agent environment could open up to even better solutions that improve the separation between the control logic and the actual controlled hardware.

Lastly, the proposed Simulation Environment could be polished and further developed in a fast and effective tool to create simulations to run agents in WoT based environments.

Hopefully, given the interest in the project from both the research groups, some of these open challenges can be undertaken in future research projects.

Acknowledgements

Coming to the end of my student years, a big thank you is due to all the people that believed in me and that shared this piece of life with me.

First I would like to thank everyone that I worked with in St. Gallen while developing this thesis, from professor Simon Mayer that welcomed me into his group and guided me in the development of my work always leaving me free to explore my ideas, to all the colleagues that I had the opportunity to meet in the office. You made me feel at home even if I was in a completely new place and it was wonderful to feel part of such an inspiring community made of people coming from all over the world and with so many different passions tied together by the love for Computer Science and innovation. Thanks in particular to Danai for being my go-to Jason expert and answering my daily questions and to Iori for helping me with all the technical infrastructure setup in the lab.

On the topic of St. Gallen, I want to also thank my flatmates Alexia, Amanda and Hyo for being super nice, letting me have some fun after my long working days and also agreeing to take part in my user-study.

To all my Italian friends goes my thank you for always being there for me both to my hometown group in Ancona, to all my friends in Cesena that started this long journey with me, and my university colleagues of the SpaceTeam with honourable mentions to Francesco, Delu and Simone for sharing quite literally every struggle and joy that brought to this destination.

I want to thank my family for the help that they give me every day and that can never be taken for granted, for the support both moral and financial that they ensured me to allow me to keep studying, learn what I love and get to have the living abroad experience that I always dreamed of.

I can not thank my girlfriend Giada enough for always pushing me towards achieving my goals, listening rambling about my work, giving me advice and just generally for being by my side. May this be just one of the many achievements we get to share.

I would also like to thank Angelo Croatti, who actually suggested sending me to St. Gallen in the first place, for introducing me to the world of PhD research, always answering my questions about it and the world of Mixed

Reality which bridged me into the interest in HCI in general (and VR gaming as well to be fair).

And, last but of course not least, I would truly like to thank professor Alessandro Ricci for supporting me during the development of this thesis and for being able through his teaching to share his passion for this "crazy agent world" which I've started to love myself and I hope to continue exploring in my days ahead.

The past years and the current time have been characterized by an unexpected series of events that cloud our vision of the future, may this accomplishment be a light to guide me always and remember me to never stop dreaming.

Thank you to everyone.

Bibliography

[1] Cleber Jorge Amaral, Jomi Fred Hübner, and Timotheus Kampik. Towards jacamo-rest: a resource-oriented abstraction for managing multi-agent systems. *arXiv preprint arXiv:2006.05619*, 2020.

[2] Kevin Ashton et al. That 'internet of things' thing. *RFID journal*, 22(7):97–114, 2009.

[3] Aaron Bangor, Philip Kortum, and James Miller. Determining what individual SUS scores mean: Adding an adjective rating scale. *Journal of usability studies*, 4(3):114–123, 2009.

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

[5] Rafael H Bordini and Jomi F Hübner. A Java-based interpreter for an extended version of AgentSpeak. *University of Durham, Universidade Regional de Blumenau*, 256, 2007.

[6] Rafael H Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons, 2007.

[7] Michael E Bratman, David J Israel, and Martha E Pollack. Plans and resource-bounded practical reasoning. *Computational intelligence*, 4(3):349–355, 1988.

[8] John Brooke. Sus: a "quick and dirty" usability scale. *Usability evaluation in industry*, 189(3), 1996.

[9] Margaret M Burnett and David W McIntyre. Visual programming. *COMPUTER-LOS ALAMITOS-*, 28:14–14, 1995.

[10] Andrei Ciortea, Olivier Boissier, and Alessandro Ricci. Engineering world-wide multi-agent systems with hypermedia. In *International Workshop on Engineering Multi-Agent Systems*, pages 285–301. Springer, 2018.

[11] Andrei Ciortea, Simon Mayer, and Florian Michahelles. Repurposing manufacturing lines on the fly with multi-agent systems for the web of things. In *Proceedings of the 17th international conference on autonomous agents and multiagent systems*, pages 813–822, 2018.

[12] Philip R Cohen and Hector J Levesque. Intention is choice with commitment. *Artificial intelligence*, 42(2-3):213–261, 1990.

[13] Michael P Georgeff and Amy L Lansky. Reactive reasoning and planning. In *AAAI*, volume 87, pages 677–682, 1987.

[14] Neil Gershenfeld, Raffi Krikorian, and Danny Cohen. The internet of things. *Scientific American*, 291(4):76–81, 2004.

[15] Dominique Guinard. *A web of things application architecture: Integrating the real-world into the web*. PhD thesis, ETH Zurich, 2011.

[16] Mahdi Hannoun, Olivier Boissier, Jaime S Sichman, and Claudette Sayettat. MOISE: An organizational model for multi-agent systems. In *Advances in Artificial Intelligence*, pages 156–165. Springer, 2000.

[17] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial intelligence*, 8(3):323–364, 1977.

[18] IntellIoT. Deliverable D2.1 Use case specification & Open Call definition, 2020. `https://intelliot.eu/wp-content/uploads/2021/08/Use-case-specification-and-Open-Call-definition.pdf?utm_source=deliverables-subpage&utm_medium=in-text-link`.

[19] Nicholas R Jennings. On agent-based software engineering. *Artificial intelligence*, 117(2):277–296, 2000.

[20] Amy J Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, et al. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)*, 43(3):1–44, 2011.

[21] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):1–15, 2010.

[22] Bonnie A Nardi. *A small matter of programming: perspectives on end user computing*. MIT press, 1993.

[23] Donald A Norman. *The psychology of everyday things*. Basic books, 1988.

[24] Hyacinth S Nwana. Software agents: An overview. *The knowledge engineering review*, 11(3):205–244, 1996.

[25] Shaileen Crawford Pokress and José Juan Dominguez Veiga. MIT App Inventor: Enabling personal mobile computing. *arXiv preprint arXiv:1310.2830*, 2013.

[26] Anand S Rao. AgentSpeak (L): BDI agents speak out in a logical computable language. In *European workshop on modelling autonomous agents in a multi-agent world*, pages 42–55. Springer, 1996.

[27] Partha Pratim Ray. A survey on visual programming languages in internet of things. *Scientific Programming*, 2017, 2017.

[28] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. Give agents their artifacts: the A&A approach for engineering working environments in MAS. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–3, 2007.

[29] Neil Selwyn. Students' attitudes toward computers: Validation of a computer attitude scale for 16–19 education. *Computers & Education*, 28(1):35–41, 1997.

[30] Yoav Shoham. Agent-oriented programming. *Artificial intelligence*, 60(1):51–92, 1993.

[31] Timothy Teo. Pre-service teachers' attitudes towards computer use: A singapore survey. *Australasian Journal of Educational Technology*, 24(4), 2008.

[32] David Weintrop. Block-based programming in computer science education. *Communications of the ACM*, 62(8):22–25, 2019.

[33] David Weintrop and Uri Wilensky. To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th international conference on interaction design and children*, pages 199–208, 2015.

[34] Mark Weiser. The computer for the 21st century. *ACM SIGMOBILE mobile computing and communications review*, 3(3):3–11, 1999.

[35] Kirsten N. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages & Computing*, 8(1):109–142, 1997.

[36] Erik Wilde. Putting things to REST. 2007.

[37] Michael Wooldridge. *An introduction to multiagent systems*. John wiley & sons, 2009.

[38] Michael Wooldridge and Nicholas R Jennings. Intelligent agents: Theory and practice. *The knowledge engineering review*, 10(2):115–152, 1995.

[39] WoT Working Group. Web of things (WoT) architecture, 2020. `https://www.w3.org/TR/2020/REC-wot-architecture-20200409/`.

[40] WoT Working Group. Web of things (WoT) Thing Description, 2020. `https://www.w3.org/TR/wot-thing-description/`.