Enhancing BDI Agents Using Fuzzy Logic for CPS and IoT Interoperability Using the JaCa Platform

Burak Karaduman 1,2,*, Baris Tekin Tezel 3 and Moharram Challenger 1,2

1 Department of Computer Science, University of Antwerp, 2020 Antwerp, Belgium; moharram.challenger@uantwerpen.be
2 AntSyMo/Cosys Core-Lab, Flanders Make Strategic Research Center, 3001 Leuven, Belgium
3 Department of Computer Science, Dokuz Eylül University, İzmir 35390, Turkey; baris.tezel@deu.edu.tr
* Correspondence: burak.karaduman@uantwerpen.be

Abstract: Cyber-physical systems (CPSs) are complex systems interacting with the physical world where instant external changes and uncertain events exist. The Internet of Things is a paradigm that can interoperate with a CPS to increase the CPS’s network and communication capabilities. In the literature, software agents, particularly belief–desire–intention (BDI) agents, are considered options to program these heterogeneous and complex systems in various domains. Moreover, fuzzy logic is a method for handling uncertainties. Therefore, the enhancement of BDI with fuzzy logic can also be employed to improve the abilities, such that autonomy, pro-activity, and reasoning, which are essentials for intelligent systems. These features can be applied in CPSs and IoT interoperable systems. This study extends the CPSs and IoT interoperable systems using fuzzy logic and intelligent agents as symmetric paradigms that equally leverage these domains as well as benefit the agent & artifact approach. In this regard, the main contribution of this study is the integration approach, used to combine the CPS and IoT augmented with fuzzy logic using BDI agents. The study begins with constructing the design primitives from scratch and shows how Jason BDI agents can control the distributed CPS. The study then performs the artifact approach by encapsulating a fuzzy inference system, utilizing time-based reasoning, and benefiting from symmetric fuzzy functions. Lastly, the study applies the self-adaptiveness method and flexibility plan selection, considering the run-time MAPE-K model to tackle run-time uncertainty.

Keywords: software agents; Jason framework; CArtAgO platform; fuzzy logic controller; fuzzy inference system; cyber-physical system; Internet of Things

1. Introduction

Cyber-physical systems are complex systems that require autonomy, pro-activity, and reasoning to handle environmental effects during their run-time operations. Given a certain point, they should be able to reach a certain goal by observing transitions between states to achieve the final state. However, uncertainties that may emerge from the physical side can prevent the output of any given input. In other words, because of the non-determinism of the real world, their functionality can be affected by any unpredictable event. Therefore, they have to reason about their environment, which requires context-awareness. In this regard, a CPS monitors and checks the external and internal events via sensors or network ports in each cycle by building reasoning on those inputs. As a CPS has to sustain its operation for a long time, it may require establishing communication with other instances by forming a topology. This makes the CPS ‘distributed’ and a communication-intensive system where the Internet of Things paradigm can overlap its requirements [1]. A common platform and paradigm should be utilized to achieve the interplay between two paradigms. As the integral paradigm for implementing such a complex and distributed CPS and IoT interoperable system, software agents [2,3] can be deployed as cyber entities. In this
way, software agents can raise the programming abstraction as well as provide reactivity, pro-activity, and autonomy to the target systems.

A software agent can be thought of as a cyber entity that has a degree of autonomy and can behave proactively by taking initiative in its actions. A software agent should be able to achieve a given goal without requiring any human intervention. A simple example is an agent deployed onto a robot to clean the house instead of the residents. While the agent attempts to satisfy this goal, it influences the environment and is influenced by the environment. During this interaction cycle, the internal structure of the agent changes continuously. For this reason, it should act based on its plan to adapt and react to these stimuli. One well-known agent-oriented programming architecture and approach is the belief–desire–intention mechanism [4,5].

BDI-based systems [6,7] can be applied as integral paradigms for CPSs and IoT interoperable systems. Moreover, one of the advantages of BDI software agents is that they can incorporate theoretical approaches [5,8,9] and applicability with systems in various application domains [10–13]. On the one hand, BDI architecture allows for reasoning on events and creating plans for reaching goals, on the other hand, fuzzy logic can be used to enhance the capabilities of reasoning to deal with run-time uncertainties.

In this study, the fuzzification of environmental perceptions (to analyze events based on data and time for plan selections and adaptation decisions) was studied. Moreover, the Mamdani-style fuzzy inference system, which applies fuzzification of the input variables, rule evaluation, aggregation of the rule outputs, and defuzzification to create a fuzzy logic controller, was designed and encapsulated into CArtAgO artifacts [14]. In addition, CArtAgO artifacts were used to establish a UDP connection with an IoT system to gather sensor data. A timer artifact was created to provide time-based reasoning. To implement software BDI agents, the Jason agent-oriented programming environment was used [15]. The output of the defuzzification process was used to control a motor to arrange the speed of a fan to cool down its perimeter. The software agents observed the states of the artifacts by focusing on them; when a state change occurred, the software agents behaved accordingly. Lastly, the whole implementation was achieved considering the MAPE-K loop as a run-time model [16–18].

This work proposes a novel method to integrate the fuzzy set theory with the multi-agent system in the scope of smart and innovative solutions, including a symmetrical approach that uses multi-agent systems as an integral paradigm to leverage CPS and IoT approaches based on symmetric fuzzy functions. Furthermore, this study provides design steps to integrate these multiple paradigms based on previous studies [13,19]. The study was extended via the JaCa platform to provide an ecosystem that runs the aforementioned paradigms and approaches. To show the integration requirements and the design steps, time-oriented measurements were taken. Experimental results show that our integration platform, namely JaCa, is a suitable environment that paves the way for integrating the CPS, IoT, MAS, and fuzzy logic theory. To show the application of the proposed approach, a step-by-step updated case study was given. Furthermore, in the literature, we have not found an application that uses the Jason BDI agents and the CArtAgO artifact design environment to provide a symmetrical approach that aims to integrate fuzzy logic to enhance the CPS and IoT system.

The novel contributions of this paper can be summarized as follows: (i) It utilizes Jason software BDI agents as integral paradigms to achieve interoperation of the CPS and IoT system. (ii) It proposes the use of CArtAgO artifacts to implement a fuzzy logic controller to enhance the reasoning of BDI agents. (iii) It applies the MAPE-K loop as a design reference and run-time model to trigger an adaptation plan using fuzzy logic.

The composition of the document is as follows: Section 2 provides background information on the aforementioned paradigms and the fuzzy inference system applied in this work. Section 3 covers the related work of this study and shows the research gap. Section 4 presents the proposed architecture. Section 5 describes the integration of fuzzy logic and
the software BDI agents. Section 6 discusses the study and offers conclusions and possible future works.

2. Background

In this section, background information about multi-agent systems and BDI agents, the fuzzy inference system, embedded systems, and CPS are given to provide insight into the rest of the document.

2.1. MAS and Jason BDI Agents

Jason [15] is a prolog-like agent programming language that was established based on AgentSpeak [20]. It also has an extended interpreter version of the Java environment. Jason applies a well-known procedural reasoning system (PRS) architecture [21], which explicitly embodies the BDI model [22]. In the BDI, the software agents continuously observe their surroundings and react instantly to the changes. These reactions are taken according to the mental states of the agents.

In the BDI, an agent has three cognitive components: belief, desire, and intention. Belief is the information that belongs to the agent itself, other agents, and agents’ surroundings. Desire represents all goals that can be potentially achieved states. Any desire is a potential trigger for the agent’s actions. Lastly, intention is defined as any state of activities that one decides to realize.

Belief is the starting point of the agent information. A plan is an action to achieve the defined goals and sub-goals. In Jason, plans are constructed by triggering an event, context, and body. Triggering an event expresses which plan is suitable for which target event. Context states whether the corresponding plan is appropriate to be activated or not. Lastly, the body consists of a set of actions and sub-goals. In each reasoning cycle, preconditions of the plans are checked using the data stored in the belief base. While the context is changed, the agent applies that plan if a condition is triggered.

2.2. CArtAgO Artifacts

In the AI domain, an agent is considered a software entity situated in an environment, capable of realizing autonomous actions to achieve its design goals. Autonomy is the prominent feature of an agent. Moreover, agents can be reactive by responding to environmental changes and can behave proactively by exhibiting goal-oriented behavior. In a distributed MAS topology, the software agents share the physical environment in which they are deployed. Agents & artifacts [23–26] consider the environment where the agents are situated as a first-class abstraction. In other words, the environment is modeled as a dynamic set of entities. An artifact is a computational object that provides some features for the software agents and modularity for the development. These features are:

- **Observable properties**: The state variables that show the current status of an artifact that the agents can perceive.
- **Observable events**: The signals that can be indicated to the agents when certain events occur.
- **Operations**: The actions belong to the artifacts. These operations can be triggered by the agents that can change the values and status of observable properties. These changes can trigger the events of the other agents as well.

The artifacts can model the environmental entities for the software agents during run-time. Agents benefit from these artifacts while pursuing their goals. Our study utilized these artifacts to create timers, the fuzzy logic controller, and a communication interface by providing modularity and concurrency handling. Implementation details about agents & artifacts will be mentioned in Section 5.
2.3. Internet of Things

The Internet of Things (IoT) is about networking physical objects, such as devices, instruments, vehicles, buildings, and other items embedded with electronics, software, sensors, and actuators. This enables these objects to collect and exchange data over a network. The IoT allows objects to be sensed and controlled remotely across existing network infrastructure, creating opportunities for more direct integration of the physical world into computer-based systems. The IoT describes a system where the internet is connected to the physical world via various sensors. The IoT can work and interact without human intervention. The development of IoT involves many issues, such as infrastructure, communication, interfaces, protocols, and standards. Our study uses the IoT paradigm with CPS and agents, considering the interoperability principle [11,25,27].

2.4. MAPE-K Loop

Modern software-intensive cyber-physical systems operate in dynamic environments where they have to deal with volatile environmental effects. System components can encounter unpredictable obstacles and situations. Because of these uncertain conditions, they may become temporarily or permanently unavailable. Ultimately, the system may deviate from its usual behavior and switch to an abnormal state. In this case, self-adaptation is applied as a practical approach to such problems. A well-recognized engineering approach to utilize adaptation involves using a feedback control loop, namely MAPE-K. As Figure 1 illustrates, it consists of four computation steps, namely monitor–analyze–plan–execute over a knowledge base [28].

Figure 1. Overview of the MAPE-K loop.

- **Monitor**: The raw data are collected using distributed and standardized sensor nodes of the system.
- **Analysis**: Raw data are processed to achieve data-to-information transformation. This can be achieved using reasoning mechanisms and logical controls. However, uncertainty can diverge the analysis phase and beliefs of the system. Therefore, a pre-processing phase, such as fuzzification, can be applied.
- **Planning**: The pre-processed contextual information obtained from the analysis phase determines the plan and adaptation decision. Considering the BDI architecture, the software agents decide which plan should be applied to reach the goals.
• **Execution:** According to the decisions taken in the planning phase, the execution phase realizes the decided actions. In this phase, agents decide to actuate the physical components, such as motors, lights, and switches—their target components.

• **Knowledge:** Knowledge maintains data of the managed system, beliefs of the agents, steady-state conditions of the environment, adaptation goals, and other relevant states that are shared by the software agents.

### 2.5. Fuzzy Inference System

The fuzzy theory was first introduced by Zadeh [29] to handle the problems described with imprecise and uncertain information. The fuzzy theory has attracted a lot of interest; it is used in decision-making and control and automation engineering (especially in intelligent systems) [30]. Fuzzy logic can mimic the continuous nature of human decision processes by improving methods based on binary logic. They are applicable for dynamic systems to achieve smooth calculations instead of traditional logic. A system might be affected during the run-time operation because of external and instant changes. Therefore, the system should be able to adapt itself to handle the situation. In contrast to conventional methods, the fuzzy theory is a platform-independent mathematical model that can be applied in any domain. Subsequently, fuzzy-based approaches, such as FLC, are quite applicable for dynamic systems, such as cyber-physical systems, mechanical systems, and robots, to deal with uncertainty while achieving smooth behavior.

A fuzzy set is a pair that can be defined with the aid of its membership function $\mu_A: U \rightarrow [0, 1]$ using to map each element of the fuzzy set to $[0, 1]$ and it can be shown as a set of pairs expressed in Definition 1:

**Definition 1.** $A = \{x, \mu_A(x)\} | x \in X$ where $x = (-\infty, \infty)$.

In 1973, Zadeh [31] published his second most influential paper, outlining a new approach to the analysis of complex systems, in which he proposes capturing human knowledge with fuzzy rules. A fuzzy rule usually takes the form $R: \text{if } x \text{ is } A, \text{ then } y \text{ is } B$ where $A$ and $B$ are linguistic values defined by fuzzy sets on discourse universes $X$ and $Y$, respectively; $\mu_A$ represents the membership degree. Considering the foundations of fuzzy logic [29,31], the fuzzy sets characterized by their membership functions can be designed to form a fuzzy logic controller. Multiple fuzzy sets can then be formed to create a type-1 fuzzy system, consisting of the fuzzifier step, the rules definition, the inference mechanism, and the defuzzifier [32].

As illustrated in Figure 2, the fuzzifier step converts precise numbers to fuzzy sets in defined ranges. The inference step then processes the fuzzy rules based on the activation of the inputs. Before the defuzzifier step, an aggregation function selects the dominant rules. Lastly, in the defuzzifier step, the fuzzy sets are converted to precise numbers. Figure 3 illustrates the inference process of a Mamdani type-1 fuzzy inference system [33].

---

**Figure 2.** Overview of type-1 fuzzy systems.
Figure 3. Mamdani type-1 fuzzy inference process.

When the inputs are fed into the fuzzifier step, membership values that represent those input values are obtained. In case the premise of a rule has more than one expression, these expressions are merged via an operator, namely t-norm, to have the activation power of that rule. Figure 3 shows the two expressions that are combined using the and operator. Here, the and operator is represented as the minimum operation, which limits the fuzzy sets by cutting the shapes. In other words, by triggering the and operation, the minimum membership value is taken and the consequent set's boundary is determined.

The operation outputs, which are the results of each rule, are combined in the aggregation step. The aggregation operator can be the maximum operation, product, sum, probabilistic, etc. In our study, we used the maximum operator as the aggregation function. Still, the resulting cut shape (fuzzy set) should be converted to have the number. In our study, we used one of the centroid methods, namely the center of gravity, which produces output according to the shape's area. The implementation details are mentioned in Section 5.

2.6. Fuzzy Logic-Based BDI Agents

Fuzzy logic-based BDI agents aim to integrate the fuzzy logic theory into the software BDI agents to enhance the decision-making, action, reasoning, and plan selection capabilities under run-time uncertainties [13,18,19]. The fuzzy logic approaches are decomposed and encapsulated into the artifacts. In this way, modularization and reusability of the fuzzy computational components can also be achieved. The BDI agents use these fuzzy artifacts to enhance their capabilities, such as decision making, uncertainty handling, input data, and action execution smoothening. The advanced fuzzy sets allow dealing with more uncertainty. In this study, ordinary type-1 fuzzy sets were used to ignite the primary initialization of their integration and the realization of the proposed approach. In further studies, the plan was to boost the approach using advanced fuzzy sets and reinforcement learning, such as the fuzzy Q-learner [34,35]. Fuzzy logic and the BDI integration method have similar application approaches, such as [36–38].

2.7. Embedded Systems

Embedded technology allows programming a microprocessor and computing hardware using embedded software. In this way, an application can be developed for performing any dedicated task. Embedded hardware binds the cyber world with the physical world using I/O ports. Actuators and sensors can be plugged into I/O ports. A sensor mea-
sures and converts the physical quantity in the environment to a digital value. Measured quantities are stored in the memory. An actuator transforms a digital command into a physical phenomenon and creates an effect in the environment, such as motion, sound, light, etc. The perceptions can be gathered via sensors, and actions can be realized through an actuator for an agent-controlled system.

2.8. CPS

CPSs consist of three aspects—cyber, physical, and network. The cyber part is created using embedded computing technologies and merges with the physical side. Cyber-physical systems are next-generation embedded systems that have been advanced to achieve physical tasks based on motion and rotation. These systems are deployed into the physical world. The network part makes the CPS distributed and able to communicate with other CPS instances or different kinds of systems using the internet. However, the interaction required with the physical world makes the CPS unpredictable because of the real world’s dynamic events. Therefore, a CPS needs to reason these events and adapt its behavior accordingly. In this way, it can sustain its operation despite these uncertain phenomena. One way to achieve it is by making the CPS smart with software BDI agents and applying fuzzy logic techniques. Sometimes, the CPS may require to react to these events as soon as possible. Therefore, fuzzy inference can enhance decision-making and smoothen physical actions.

3. Related Work

In the literature, the application of fuzzy logic has been addressed by various perspectives, including vision surveys, projects, and research directions. This study’s related work explored the scope of multi-agent systems, fuzzy logic, and self-adaptive systems.

Generally, CPS and IoT challenges widely lack acceptable intelligent approaches and multi-paradigm methods and consider the complexity of these systems [39,40]. In [41], the authors mention joint features of the industrial Internet of Things (IIoT) and CPS. They underline a few methods about the applicability of IoT-enabled [42] solutions with cyber-physical production systems. In their study, CPS is defined as a networked system that consists of many sensor nodes. To implement their proposed architecture, an IoT-like architecture [43] approach can be used, such as node, edge, and cloud levels to modularize and organize the software agents’ capabilities [44].

Leitao et al. [45] discuss the software agents’ capabilities, CPS challenges, and requirements. They focused on the applicability of the MAS to CPS and the IoT system. Their study showed that the MAS is generally very good at creating collaboration and integrity when distributed, providing adaptable decisions when unpredictability exists during run-time. In general, it is good at reasoning (concerning events) and providing sustainability.

In [46], the authors surveyed uncertainty (in a self-adaptive systems perspective). The participants agreed on implementing self-adaptation mechanisms to cope with unanticipated changes during run-time. They also suggested that uncertainty could be represented using run-time models, such as the MAPE-K loop.

Bolturk and Kahraman [47] reviewed the relationship between robots and fuzzy sets. They focused on the advantages of fuzzy logic as it is a flexible way to model human behavior. Moreover, the authors of [48] mentioned fuzzy sets and their extensions. They emphasized that fuzzy set extensions aim to provide more capability to deal with uncertainty.

Valdez et al. [49] presented a comparative study of conventional and type-2 fuzzy logic controllers for velocity regulation. They compared PI, PID, and fuzzy logic controllers using the EV3 LEGO Mindstorms kit and simulated their controllers in MATLAB.

Cuevas et al. [50] proposed using type-2 fuzzy logic to control a mobile robot’s behavior (to deal with uncertainty). They considered that mobile robots should be able to tolerate a high level of imprecision in their operating environments. They planned to navigate their mobile robots in an unknown environment.
Xing et al. [51] introduced an agent-oriented embedded control system for an autonomous-guided vehicle. They benefited from the multi-agent system (implementing organizational features between system components).

Ciortea et al. [52] benefited from the JaCaMo platform by running web mash-ups, to create a distributed multi-agent system for controlling industrial manufacturing robots. They mostly focused on the artifact (CArtAgO) level to control the network events via the Jason BDI agents. They used automated planning to synthesize basic plans for actuating the movements of the robot arms (for disabling and assembling wooden stools). Despite their state-of-the-art study, their work was quite communication-intensive and their focal point was solely on running web-based artifacts for controlling the robotic systems in a loosely-coupled manner. In our study, we focused on the BDI agent and artifact levels to create fuzzy computational artifacts (to enhance the software agents for the CPS). Moreover, we created timer and communication artifacts to achieve time-based reasoning and CPS/IoT interoperability, respectively. It was contemplated that robot arms are physical artifacts that should be tacked from the CPS perspective as well. In this way, the physical uncertainties can be handled and smoothened using fuzzy logic.

Minotti et al. [53] investigated the application of agent-oriented programming for web applications based on the JaCa-web framework. They geared the BDI agents with the artifact-based web environment and services based on service-oriented architecture for distributed systems.

Croatti and Ricci. [24] provided the ported and open-source version of JaCa for Android mobile systems based on JaCa-Android architecture [54]. In this way, the development of agent-based applications on mobile devices is also supported by IoT devices, specifically wearable technologies. Their adaptations aim to assist the users as personal assistants[55] based on the agents & artifacts approach. A single BDI agent accesses the services and applications of the target mobile phone. It presents the TraumaTracker case study and shows the usability of the JaCa approach in the healthcare domain [11,25].

Palanca et al. [26] introduced IoT artifacts for the SPADE Python-based agent development environment [56]. Their artifact model aims to maintain the specific aspects of IoT devices by integrating characteristics of the programming model into the SPADE platform, especially for embedded devices. Their artifact approach conforms to CArtAgO artifacts [57]. They also describe how their artifact model can be extended in embedded C for the ESP32 microcontroller.

Bienz et al. [58] presented a similar study with [52]. They provided a search engine for the Web of Things that allows software agents to perform search queries autonomously to enhance their goal-oriented behaviors. They focused on the hypermedia MAS [59] and Web of Things to deploy software BDI agents for network-based goals. In our study, we focused on the interoperability of the Internet of Things and the cyber-physical systems, while modeling some features as artifacts, as also proposed in [52,58,59]. Moreover, they applied fuzzy logic to enhance software BDI agents and used fuzzy logic to analyze/apply adaptation. We also applied the run-time model MAPE-K loop to empower our design perspective.

4. Proposed Architecture

This section presents the proposed architecture that merges the MAPE-K loop [60], BDI agents, and the fuzzy inference system. Figure 4 illustrates the proposed architecture. In the MAPE-K loop, there are two parts—the managing system and managed system. The managing system monitors the state of the managed system to control its operation and applies adaptation when required. In our design, we considered the BDI agents and artifacts as first-class abstractions. To provide adaptation, the system needs context-awareness. In the Jason framework, this feature is achieved implicitly by the procedural reasoning system. The BDI agents observed the environment. This maps with the monitor phase of the runtime model. The BDI agents then update their belief base. The communication interface, which can be an artifact, is also checked in case of any data input from the network part.
In our design, the BDI agents reason over the environmental changes, time, and network events. The fuzzy logic controller (FLC) uses artifacts to infer the gathered data to execute the physical entity. However, before the realization of the FLC execution, the beliefs of the agents should be gathered. Once the input parameters are fuzzified, the membership degrees can be stored in the belief base of the agents. To run the inference engine, the fuzzy rules should be given to obtain a result. The defuzzifier then converts the fuzzy result to a crisp value. This value is used by the executor to arrange the control power on a physical entity. However, as the artifacts are observable properties when their states also alter, an analysis based on their states can be made to decide whether there is a requirement for the adaptation. If the analyzer conditions are met, then the adaptation plan is scheduled and executed by the executor. The concrete application of the proposed architecture will be mentioned in detail in Section 5.

5. Case Study: Distributed Fan Controller Enriched with Fuzzy Inference for the Multi-Agent System

In this section, the process of integrating fuzzy logic and the MAS is tailored. To achieve the CPS and IoT interoperation, a few steps and experiments were followed and conducted. We applied a from-scratch implementation to show the integrability of these paradigms. From our point of view, the design and implementation of fuzzy logic integration with the MAS as an integral paradigm to leverage CPS and IoT approaches should be explicitly shown as purified from black-box approaches, because any external tool creates dependency and prevents modularity and modifiability. However, tool technologies, such as MATLAB and jFuzzyLogic [61], can be employed as services via CArtAgO artifacts. Any possible further improvements will be dependent on tool providers; however, black-box tools can be beneficial to validate the construction phases.

As mentioned, the BDI software agents commit to their goals by executing plans. However, at the same time, they can reason on events and react accordingly. The implemented plans of the BDI agents process these plans based on the triggering event. These events can be external via physical interactions or internal where the agents themselves have actions or belief changes caused by other agents. As Listing 1 shows, a Jason plan has the following form:

Listing 1. The form of a Jason plan.

```
1 triggering_event: application_context ← plan_body.
```
The triggering events should be satisfied to schedule that plan. In Section 5.1, the first version of the fuzzy logic and the software BDI agents are given.

5.1. Jason Only Implementation

An abstract case study (as an earlier study and the first step) was implemented using a single software agent [13]. In that work, the software agent controlled a CPU fan based on the fuzzy BDI approach. The fuzzy sets were implemented using the Jason Java environment. The effectiveness of the fuzzy logic and BDI integration was shown based on a scenario where the temperature rose. The non-fuzzy BDI agent followed the traditional Boolean logic plans represented by Listing 2. When the software agent was initialized, it directly applied the !keepCool plan. Jason’s reasoning mechanism checked each plan that met the triggering event conditions based on belief $T$. It then realized the changeFanSpeed action and continued to reason on the same plan.

Listing 2. Boolean logic based BDI Agent’s Plans.

```java
1 !keepCool.
2 /* Plans */
3 +!keepCool: temp(T) & T<20 <$> changeFanSpeed(0);!keepCool.
4 +!keepCool: temp(T) & T>=20 & T<35 <$> changeFanSpeed(0.25);!keepCool.
5 +!keepCool: temp(T) & T>=35 & T<50 <$> changeFanSpeed(0.35);!keepCool.
```

In Listing 3, fuzzy logic-based software BDI agent plans are shown. In line 2, the software agent initiated the keepCool plan. In each reasoning cycle, the membership degrees are calculated, and the plan with the highest membership degree is selected using the defined line 1. This time changeFanSpeed action also realizes the actuation power considering the membership degree. To evaluate the effectiveness of the fuzzy BDI method, we ran experiments and statistical approaches. Interested readers can find the details of the study in [13,19].

Listing 3. Fuzzy logic-based BDI Agent’s Plans.

```java
1 /* RULE */ isIt (T):− temp(T,D1) & not(temp( ,D2) & D2>D1).
2 !keepCool.
3 +!keepCool: isIt (cold) <$> temp( cold,D1);
4 changeFanSpeed(0.4,D1);!keepCool.
5 +!keepCool: isIt (warm) <$> temp( warm,D1);
6 changeFanSpeed(0.6,D1);!keepCool.
7 +!keepCool: isIt (hot) <$> temp( hot,D1);changeFanSpeed(1.0,D1);!keepCool.
```

It was then decided to create a concrete study to deploy the software agents onto an embedded board to control an actual motor, a basic CPS example, as it makes a rotational movement. In this scenario, we wanted to control a CPS that gathered the data from a temperature sensor. A wireless embedded board collected the sensor data as the IoT system. Section 5.2 mentions deploying the software BDI agents onto embedded boards to control a CPS that gathers data from an IoT system.

5.2. Physical Deployment of the Jason BDI Agents

The physical case study applies the above fuzzy BDI code, which was given in Listing 3 and the corresponding membership function. Brief information about the physical components will be shared to provide better insight into the physical setup.

5.2.1. Embedded Hardware

To achieve CPS and IoT interoperability, the study was extended using the ESP32 IoT development board, the LM35 temperature sensor, and Raspberry Pi 3 + BrickPi boards.
LM35 was connected to the ESP32’s analog pin to sample the environment’s temperature. ESP32 sends the collected data to the Raspberry Pi 3 board using UDP communication. The Jason agent then stores these data into its belief base via the Java environment and actuates the fan through a motor. Figure 5 represents the connection architecture to implement the physical case study and Figure 6 illustrates the concrete implementation.

The concrete implementation consisted of an LM-35 temperature sensor. The ESP32 sampled the temperature data and delivered them to Raspberry Pi 3. In Raspberry Pi 3, the software agent was deployed. The software agent reasoned (based on the membership degree) and arranged the speed of the motor.

The ESP32-WROOM-32 microcontroller is a low-end, low-cost, dual-core microcontroller from the embedded hardware family. It integrated Wi-Fi and Bluetooth modules that targeted various applications ranging from low-power sensor networks to the most demanding tasks.

LM-35 temperature sensor can measure from a −55-degree centigrade to a 150-degree centigrade. The input voltage can vary from +4 to 30 volts. It consumes around 60 microamperes. For each one-degree increase, the voltage raises by +10 volt mills by the sensor’s vout pin, e.g., if the sensor outputs 100-volt mills at the vout pin, then the temperature (in centigrade) will be 10 degrees centigrade. Figure 7 represents the LM-35 temperature sensor.

Raspberry Pi 3. It is high-end embedded hardware, as shown in Figure 8. Raspberry Pi 3 is a credit card-sized low-powered computer board with Ethernet and Wi-Fi connections. Moreover, it has an HDMI video output, an audio output, and an SD card slot to run a Linux operating system. Raspberry Pi is beneficial hardware used for high-end tasks. It has many adaptation and extension boards for specific domains and goals.
The BrickPi Hardware Interface is a hardware interface for Raspberry Pi that allows controlling special sensors and actuators. The BrickPi is attached to the top of the Raspberry Pi. It allows controlling the servo motors that can rotate at high-speed. Figure 9 depicts the BrickPi adaptation hardware interface.

As mentioned, we created an IoT system to establish CPS and IoT interoperability. While the CPS has tightly-coupled relations with the agents, the IoT system was designed as loosely coupled because we preferred using ESP32 as a low-end device. Here, a low-end device refers to embedded hardware that samples the data and sends it to the agents and artifacts container device (Raspberry Pi 3). In Section 5.2.2, embedded software that was used to program these devices will be provided.

5.2.2. Embedded Software

The ESP32 software, shown by Listing 4, was implemented using Arduino IDE. Lines 2 and 3 included Wi-Fi and Wi-Fi-based UDP functionalities. Lined 4 and 5 were used to define the network SSID and password while lines 6 and 7 stored the target device’s IP address and destination port. A UDP instance was created at line 9. Between lines 10 and 17, the setup function was defined to create necessary configurations for Wi-Fi, UDP initialization, and connection status. The loop function ran the code lines between 20 and 24 continuously. For each cycle, the temperature value was read from pin 36 and sent via UDP using the beginPacket function. The packet ends at line 24.

Listing 4. ESP32 Software for sampling temperature data and sending it via UDP.

```
#include <WiFi.h>  
#include <WiFiUdp.h> 

/* WiFi network name and password */
const char * ssid = "SSID";
const char * pwd = "password";
const char * udpAddress = "192.168.0.164";
const int udpPort = 4444;

// create UDP instance
WiFiUDP udp;

void setup(){
  Serial.begin(115200);
  //Connect to the WiFi network
  WiFi.begin(ssid, pwd);
  Serial.println(""");
  // Wait for connection
```
```java
while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.println("...");
} // This initializes udp and transfer buffer
udp.begin(udpPort);

void loop(){
    int sensorReading = analogRead(36);
    udp.beginPacket(udpAddress, udpPort);
    udp.printf("%u \n", sensorReading);
    udp.endPacket();
}
```

Listing 5 represents the UDP receiver software in the Jason environment. As Jason uses Java as an environment, a Java function was implemented to receive UDP packets. In line 1, the listening port was defined and a buffer was created in line 2. `getUDPMessage` was created to receive incoming packets. When a packet was received, it was stored in the belief base and the `StringBuilder` function was called to transform the data into human-readable text. The Jason agent called this function for each agent cycle, updated its belief base, and then selected the suitable plan according to the context of the data. To achieve this, the agent software mentioned in Listing 3 was modified, as given by Listing 6.

Listing 5. UDP Receiver software in the Jason environment.

```java
ds = new DatagramSocket(4444);
receive = new byte[1024];
public void getUDPMessage() {
    UDPrev = new DatagramPacket(receive, receive.length); ds.receive(UDPrev);
    updateFBDIPercepts(Double.parseDouble(data(receive));
    public, StringBuilder data(byte[] a) {};
```

Listing 6. UDP Receiver software in the Jason environment.

```java
isit (T):- temp(T,D1) & not(temp(_,D2) & D2>D1).
! init .
+! init: true <- .print("starts"); .wait(4000);! sample.
+! sample: true <- .wait(1); sampleSensorData;! keepCool.
+! keepCool: isit (cold) <- ?temp(cold,D1);fanAction(250,D1);! sample.
+! keepCool: isit (warm) <- ?temp(warm,D1);fanAction(350,D1);! sample.
+! keepCool: isit (hot) <- ?temp(hot,D1);fanAction(500,D1);! sample.
```

At line 2, the agent was initialized using `!init` goal. Line 3 instructed the goal by selecting a suitable plan. At first, the agent waited for 4 seconds to be sure that the hardware configurations were prepared. Then, according to the incoming data, fuzzy-based plans were selected using the "isit(T)" rule in line 1. However, when the data packets were sent, a lag was observed between the sent data and the received data. Therefore, time measurements were taken to detect the bottleneck.

5.2.3. Time Measurements

We sampled the time parameters of each phase to which the data were transferred to. Experiments were achieved in a LAN network using a private modem and wireless connections. The time synchronization was established using an NTP server in ESP32-Arduino software using a Java library in the agent software and a physical–digital clock. The time variable was sent to the agent software and was subtracted by the current time. The NTP server in the ESP32-Arduino server is shown in Listing 7.
The agent instructs the UDP receiver function for each reasoning cycle to gather the temperature data, which are sent by ESP32. However, we discovered that the agent’s reasoning cycle was slower and caused desynchronization between the real-time temperature change and actuation. Therefore, we tested each phase to validate that our assumption was true. Moreover, we tested our fuzzy function with pure Java and pure Python implementations. To measure the time differences, the incoming time variable was subtracted by the local time of the target device. In this case, the target device was the Raspberry Pi that ran the agent. To achieve this, the `LocalDateTime.now().getSecond()` function was used.

Considering the values in Table 1, we decided to use a single plan (see Listing 8), which only realized receiving UDP packets that had enumerated data. We discovered that there were packet losses during the datagram packet transmission. Considering a non-enumerated packet transmission, it was not possible to detect the cause of the time delay and data desynchronization. Therefore, we experimented with the bottleneck using two different devices as clients—ESP32 and PC. To detect the packet losses, we applied different message counts. The PC that ran the experiments had 16 GB of RAM; Intel(R) Core(TM) i7-10850H, 2.70GHz, 2.71 GHz CPU.

### Listing 8. Simple plan to receive UDP messages.

```java
+!sample: true <- sampleSensorData; !sample.
```

Considering Tables 1 and 2, a thread mechanism should be created to automatically update the agent’s knowledge base without calling any sensing action. This thread should be created inside the Jason environment. In this way, the software agent could synchronize with the temperature data stream. However, creating extra threads caused some problems as the software agents are the thread owners, and their mechanisms are private. Therefore, we adapted our approach using the CArtAgO framework, which provided many advantages and thread-safe operations that will be mentioned in Section 5.3.

### Table 1. Time delay and activities.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Time Delay</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP32/Arduino</td>
<td>≈1 ms</td>
<td>Sample and send</td>
</tr>
<tr>
<td>Jason(Sense) only</td>
<td>≈0 ms</td>
<td>Receive sense data</td>
</tr>
<tr>
<td>Jason(Sense)-to-Jason(Reasoning)</td>
<td>≈8–10 s</td>
<td>After sense to plan selection</td>
</tr>
<tr>
<td>Jason(Plan)-to-Jason(Actuation)</td>
<td>≈0 ms</td>
<td>Plan intention and action</td>
</tr>
<tr>
<td>ESP32(Sender)-to-Java(Receiver)</td>
<td>≈0 ms</td>
<td>UDP communication</td>
</tr>
<tr>
<td>Java only</td>
<td>≈0 ms</td>
<td>Match data to fuzzy functions</td>
</tr>
<tr>
<td>ESP32-to-python</td>
<td>≈0 ms</td>
<td>UDP communication</td>
</tr>
<tr>
<td>Python only</td>
<td>≈0 ms</td>
<td>Match data to fuzzy functions</td>
</tr>
</tbody>
</table>
Table 2. Experiments based on device and data packets.

<table>
<thead>
<tr>
<th>Device</th>
<th># of Packets</th>
<th>Time Delay</th>
<th># of Lost Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP32</td>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>ESP32</td>
<td>50</td>
<td>1 s</td>
<td>5</td>
</tr>
<tr>
<td>ESP32</td>
<td>100</td>
<td>3 s</td>
<td>5</td>
</tr>
<tr>
<td>ESP32</td>
<td>150</td>
<td>3 s</td>
<td>8</td>
</tr>
<tr>
<td>ESP32</td>
<td>250</td>
<td>4 s</td>
<td>9</td>
</tr>
<tr>
<td>ESP32</td>
<td>300</td>
<td>5 s</td>
<td>19</td>
</tr>
<tr>
<td>ESP32</td>
<td>500</td>
<td>5 s</td>
<td>221</td>
</tr>
<tr>
<td>ESP32</td>
<td>1000</td>
<td>5 s</td>
<td>721</td>
</tr>
<tr>
<td>PC</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PC</td>
<td>50</td>
<td>1 s</td>
<td>0</td>
</tr>
<tr>
<td>PC</td>
<td>100</td>
<td>2 s</td>
<td>0</td>
</tr>
<tr>
<td>PC</td>
<td>150</td>
<td>2 s</td>
<td>0</td>
</tr>
<tr>
<td>PC</td>
<td>250</td>
<td>4 s</td>
<td>0</td>
</tr>
<tr>
<td>PC</td>
<td>300</td>
<td>5 s</td>
<td>15</td>
</tr>
<tr>
<td>PC</td>
<td>500</td>
<td>5 s</td>
<td>215</td>
</tr>
<tr>
<td>PC</td>
<td>1000</td>
<td>5 s</td>
<td>715</td>
</tr>
</tbody>
</table>

5.3. Enhancing BDI Agents Using Fuzzy Logic and JaCa

As mentioned in Section 2.2, the CArtAgO framework models the software agents’ environment using artifacts. They can be considered the computational components that produce beliefs and events for the agents. The artifacts also provide modularity and encapsulation. In this way, an artifact becomes observable by more than one agent, and the agents can also create more than one instance of that artifact. The value of an artifact is updated using the `updateValue` method. CArtAgO then implicitly updates the belief base of the corresponding agent(s). In this way, the artifacts can be used in a thread-safe manner. As a fuzzy logic controller has many computational stages, each stage can be interpreted as an artifact. By applying the separation of concerns principle, independent refinement of the computational resources for controlling and computing is achieved. In other words, a fuzzy logic controller, a timer, or a communication interface can advance by adding new features without blending the code into the agent programming level.

In our JaCa implementation, an initial plan was implemented to have the agents create their artifacts. Listing 9 shows the mock-up sensor artifact creation.

Listing 9. The artifact creation plan.

```java
1   makeArtifact("mocksensor1","example_JaCaMo.MockSensor",[],MockSensor1);
2   mockdata[artifact_id(MockSensor1)];
3   focus(MockSensor1);
```

In our fuzzy BDI-based JaCa implementation, we created multi-agents by assuming that they were distributed inside a room. In line 1, the agents created the mock-up sensor artifact, naming it the `MockSensor1`. The agent could instantiate more than one mock-up sensor artifact. It calls the “mockdata” operation to generate a floating value between 6 and 35, randomly. These values mimic the usual room temperature. The agent focuses on this artifact to keep track of its state changes. As creating uncertain events may not be so realistic in a physical environment, a mock-up sensor artifact was created.

The lines between 1 and 2 in Listing 10 show how the mock-up sensor generates temperature data. Once the temperature value is generated, it becomes fuzzified. As mentioned in Section 2.5, the first phase is the fuzzification of the crisp number. As
three membership degrees are given, this artifact should have three observable properties, namely d1, d2, and d3. The lines between 7 and 24 show the implementation of fuzzy triangular sets (for the sake of brevity, an excerpt of it is shown). The boundary overlaps were implemented as shown between lines 16 and 18.

Listing 10. Mock-up sensor artifact.

```java
SecureRandom r = new SecureRandom();
double random = 6.0 + r.nextDouble() * (35.0 - 6.0);
double d1,d2,d3;
defineObsProperty("d1", 0.0);
defineObsProperty("d2", 0.0);
defineObsProperty("d3", 0.0);
if (n_cpuTemp<6.5) {
    d1=n_cpuTemp/6.5;
    d2=0.0;
    d3=0.0;
    d1f.updateValue(d1);
    d2f.updateValue(d2);
    d3f.updateValue(d3);
} else if (n_cpuTemp>=6.5 && n_cpuTemp<18.0 ) {
    d1=(18.0-n_cpuTemp)/11.5;
    if (n_cpuTemp>=10.0) {
        d2=(n_cpuTemp−10.0)/9.5;
    }
    else    
        d2=0.0;
    d3=0.0;
    d1f.updateValue(d1);
    d2f.updateValue(d2);
    d3f.updateValue(d3);
} else {
    d2=0.0;
    d3=0.0;
    d1f.updateValue(d1);
    d2f.updateValue(d2);
    d3f.updateValue(d3);
}
```

A triangular function that has a lower limit $a$, upper limit $b$, and any value $m$ between these limits where $a < m < b$, can be defined as the fuzzy set using the functions shown by Equation (1).

$$ \mu_{\text{trian}}(x) = \begin{cases} 
0, & x \leq a \\
\frac{x-a}{m-a}, & a < x \leq m \\
\frac{b-x}{b-m}, & m < x \leq b \\
0, & x \geq b 
\end{cases} \quad (1) $$

In our implementation, we used triangular fuzzy sets to form the fuzzy logic controller. To provide better insights for the rest of this subsection, the fuzzy rules and fuzzy functions were given beforehand. Figure 10 shows the triangular fuzzy functions, which were used to implement the fuzzy logic controller. Listing 11 shows the fuzzy rules of the fuzzy logic controller and Listing 12 lists the fuzzy function boundaries.

![Figure 10. Fuzzy functions.](image)
Listing 11. Fuzzy rules of the fuzzy logic controller.

1 RULE 1: IF sensor IS cold AND time IS short THEN speed IS low;
2 RULE 2: IF sensor IS warm AND time IS long THEN speed IS average;
3 RULE 3: IF sensor IS hot AND time IS long THEN speed IS average;
4 RULE 4: IF sensor IS hot AND time IS extreme THEN speed IS fast;
5 RULE 5: IF sensor IS hot AND time IS short THEN speed IS fast;
6 RULE 6: IF sensor IS warm AND time IS short THEN speed IS average;
7 RULE 7: IF sensor IS warm AND time IS extreme THEN speed IS average;
8 RULE 8: IF sensor IS cold AND time IS short THEN speed IS low;
9 RULE 9: IF sensor IS cold AND time IS extreme THEN speed IS average;

Listing 12. Fuzzy function boundaries.

1 // Sensor
2 TERM cold:= trian 0 6.5 18 ;
3 TERM warm:= trian 10 19.5 26 ;
4 TERM hot:= trian 21 32.5 39 ;
5 // Time
6 TERM short:= trian 0 10 25 ;
7 TERM long:= trian 15 30 45 ;
8 TERM extreme:= trian 35 50 60 ;
9 // Speed
10 TERM low:= trian 0 166.5 370 ;
11 TERM average:= trian 250 499.5 750 ;
12 TERM fast:= trian 666 832.5 999 ;

As mentioned, time-based reasoning was also considered. Therefore, a timer artifact was created. To create the timer artifact, the Java timer library was used. As shown in Listing 12, three membership degrees were given. Therefore, three observable properties were created—t1, t2, and t3.

The same implementation code that is shown in Listing 10 (with corresponding boundary values for time inputs) was used to create the timer artifact shown by Listing 13. However, this artifact had extra operations, such as stopping the timer and signal calling for the expiration of the time. The save function, defined by lines 8 to 12, is used for creating a session and committing the changes. Because the timer artifact creates an extra thread in addition to the agent’s main thread, an external session must be created to achieve a thread-safe operation. As the timer updates itself each second, it should commit these changes after one second without waiting for the reasoning cycle or any scheduled event. Therefore, the commit function is called. If the timer expires (1-min duration) then it sends a signal to the agent by creating an event, namely timerExceeded.

Listing 13. Timer artifact.

1 final Timer timer = new Timer();
2 defineObsProperty("t1", 0.0) ;
3 defineObsProperty("t2", 0.0) ;
4 defineObsProperty("t3", 0.0) ;
5 timer.scheduleAtFixedRate(task, 0, 1000) ;
6 save() ;
7 ... 
8 void save() { 
9 this.beginExternalSession();
10 commit();
11 this.endExternalSession(false);
12 p.updateValue(p.intValue()+1);
13 ... 
14 void timer_stop() { 
15 this.beginExternalSession();
16   timer.cancel();
17   timer.purge();
18   this.endExternalSession(false);
19   ...
20 private void call_signal() {
21       this.beginExternalSession();
22       signal("timerExceeded");
23       this.endExternalSession(false);
24   }

As mentioned in Section 2.5, a Mamdani-style fuzzy logic controller was implemented using CarTAgO artifacts. The AND operation was realized using the Math.min Java function. Listing 14 shows the inference phase of the AND operation. The rules given in Listing 11 were implemented, applying the Min operation on sensor and time membership degrees. The lines between 1 and 12 show the cold-time, warm-time, and hot-time parameters and their target projections to the speed fuzzy functions. To realize the rule aggregation to find the dominant rules, the Max function was used and stored in the array, namely selected.


<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>// Cold – Time</td>
</tr>
<tr>
<td>2</td>
<td>double min1 = Math.min(d1_t, t1_t); // =&gt; S1 – Low</td>
</tr>
<tr>
<td>3</td>
<td>double min2 = Math.min(d1_t, t2_t); // =&gt; S1 – Low</td>
</tr>
<tr>
<td>4</td>
<td>double min3 = Math.min(d1_t, t3_t); // =&gt; S2 – Avg</td>
</tr>
<tr>
<td>5</td>
<td>// Warm – Time</td>
</tr>
<tr>
<td>6</td>
<td>double min4 = Math.min(d2_t, t1_t); // =&gt; S2 – Avg</td>
</tr>
<tr>
<td>7</td>
<td>double min5 = Math.min(d2_t, t2_t); // =&gt; S2 – Avg</td>
</tr>
<tr>
<td>8</td>
<td>double min6 = Math.min(d2_t, t3_t); // =&gt; S2 – Avg</td>
</tr>
<tr>
<td>9</td>
<td>// Hot – Time</td>
</tr>
<tr>
<td>10</td>
<td>double min8 = Math.min(d3_t, t2_t); // =&gt; S2 – Avg</td>
</tr>
<tr>
<td>11</td>
<td>double min7 = Math.min(d3_t, t1_t); // =&gt; S3 – Fast</td>
</tr>
<tr>
<td>12</td>
<td>double min9 = Math.min(d3_t, t3_t); // =&gt; S3 – Avg</td>
</tr>
</tbody>
</table>
| 13 | ...
| 14 | Double[] S1 = {min1,min2};
| 15 | Double[] S2 = {min4, min5, min6, min8};
| 16 | Double[] S3 = {min7, min9};
| 17 | double[] selected = new double[3];
| 18 | selected[0] = Collections.max(Arrays.asList(S1));
| 19 | selected[1] = Collections.max(Arrays.asList(S2));
| 20 | selected[2] = Collections.max(Arrays.asList(S3));

In the next phase, defuzzification should be done to receive the crisp output according to the partial projected area of the speed fuzzy functions.Listing 15 shows the defuzzification implementation and required functions.

Listing 15. Defuzzification Phase.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>calculateDeFuz(selected[0], selected[1], selected[2]);</td>
</tr>
<tr>
<td>2</td>
<td>void calculateDeFuz(double bound1, double bound2, double bound3) {</td>
</tr>
<tr>
<td>3</td>
<td>int high_b=decide_low_limit(bound1, bound2, bound3);</td>
</tr>
<tr>
<td>4</td>
<td>int low_b=decide_high_limit(bound1, bound2, bound3);</td>
</tr>
<tr>
<td>5</td>
<td>for (speed = low_b; speed &lt; high_b; speed+=1) {</td>
</tr>
<tr>
<td>6</td>
<td>if (speed &lt; 166.5) {</td>
</tr>
<tr>
<td>7</td>
<td>d1 = speed / 166.5;</td>
</tr>
<tr>
<td>8</td>
<td>d2 = 0.0;</td>
</tr>
<tr>
<td>9</td>
<td>d3=0.0;</td>
</tr>
<tr>
<td>10</td>
<td>else if (speed &gt;= 166.5 &amp;&amp; speed &lt; 370.0) {</td>
</tr>
<tr>
<td>11</td>
<td>}</td>
</tr>
</tbody>
</table>
d1 = (370.0 − speed) / 203.5;
if (speed >= 250.0) {
    d2 = (speed − 250.0) / 249.5;
} else {
    d2 = 0.0;
}
else if (speed >= 370.0 && speed < 499.5) {
    d1 = 0;
    d2 = (speed − 250.0) / 249.5;
    d3 = 0;
}
...  

int decide_high_limit(double bound1, double bound2, double bound3)
{
    if (bound1 != 0.0){
        low_bound[0] = 0;
    }
    if (bound2 != 0.0){
        low_bound[1] = 250;
    }
    if (bound3 != 0.0){
        low_bound[2] = 666;
    }
    return Collections.min(Arrays.asList(low_bound));
}

double val = findMaxAndCut(d1, d2, d3, bound_a, bound_b, bound_c);
total2 += val; total1 += (val * speed); System.out.println("DEFUZZ= " + total1 / total2);

To achieve the defuzzification and automate this phase, the dominant rules stored in the selected array are given as parameters to the calculateDeFuz function. Then, low and high boundaries are found. In this way, the delta steps can scan the projected area on the speed variables’ fuzzy functions. Then, the “for loop” at line 6, approaches with delta steps starting from the low boundary to the high boundary by incrementing the speed variable, triggered if conditions calculate the membership degrees stored in the variables, namely d1, d2, and d3. These variables are then sent to the findMaxAndCut function; if there is a membership degree that exceeds the limits of the membership degrees, it is set to that limit. The function then returns the maximum membership degree, which also scans the lower membership degrees. As mentioned, we used the center of gravity as the defuzzification method. Line 31 shows the calculation of the projected area on the speed’s fuzzy functions according to the center of gravity method. To validate the defuzzification step, the results from the jFuzzyLogic [61] were compared with our implementation. As can be seen from Figure 11, the difference is quite minor; it should also be considered that the API that sets the velocity accepts only integer numbers.

As mentioned in Section 5.2.3, there was an 8–10 second lag between the sent data and the received data. To tackle that problem, the changes described in Listing 16 were made. At first, the code shown in Listing 5 was adapted as an artifact. As shown in line 1, the agent then created that artifact and a separate intention using the !! notation. As lines 2 to 7 show, the getUDPMessage function was enhanced with CArtAgO’s ExternalSession function. In this way, a thread-safe and asynchronous flow was achieved. Lastly, as shown in line 7, after the compilation of the getUDPMessage, to receive the incoming packages, the plan was recalled to create a continuous loop in a separate intention (using a separate thread). As shown in Figure 12, the process was also tested using the Packet Sender application. After these steps, the software agent was deployed onto embedded hardware. A sample video could be viewed at https://youtu.be/mjLZpoMGxrA (accessed on 15 June 2022).
Figure 11. The comparison of the result with the jFuzzyLogic.


```java
1 makeArtifact("udp0","example_JaCaMo.getUDPMessage",[],IdUDP);!udpFollow;
2 getUDPMessage(){
3   this.beginExternalSession();
4   ...
5   this.endExternalSession(false);
6 }
7 +udpFollow <- getUDPMessage;!udpFollow.
```

Figure 12. Asynchronous communication with the IoT system.
Distributed and Adaptive Scenario

As mentioned, the MAS provides distributed topology for the CPS and IoT systems. In a distributed topology, it is expected that the instances in the network should be able to communicate with each other to transfer information. As multiple fans inside a room attempt to cool down the room temperature because of an uncertain event, one of the node sensors may become broken or unavailable for a few seconds. In this case, the software agent should be able to adapt. Therefore, the time criterion was considered and fuzzified. In case the time criterion has a membership degree in the extreme state, the agent switches to the adaptation state by running the adapt plan. The agent broadcasts the need for sensor input membership degrees to the whole network. However, because of environmental heterogeneity, the agent should evaluate the data provided by the other agents based on time and distance. For example, if one of the other agent sensors is also unavailable for a few seconds, it could be long, but not extreme. Then, the membership degrees of the sensor input should be lowered and sent to the agent that broadcasted for the membership degrees. Therefore, a relation between distance and the time criteria should be established to compensate for the membership degrees, to be sent to the agent that requires an update. The hedge modifier, especially the concentration operation, can be applied to modify the agents’ beliefs about their membership degrees. As Equation (2) shows, the hedge concentration can be defined as:

\[ A^k = [\mu_A(x)]^k, \text{ where } \{ k \in \mathbb{R}^+ : k > 1 \} \]  

Regarding fuzzy set \( A \) and membership function \( \mu_A(x) \), defined by Definition 1 in Section 2.5, \( A \) represents the concentrated form of any given membership degree \( \mu_A(x) \) base value to the power of the positive \( k \) value. Recall the Definition 1, \( \mu_A(x) \) ranges \([0, 1]\) and \( k \) is any positive real number in this limit. Therefore, the agents’ beliefs can be sharpened using this operation, considering \( k \) going to infinity and \( A \) approaching zero. Moreover, as the physical rules are applied in a distributed CPS/IoT topology, the \( k \) value is bound with respect to time and distance, resulting in the definition of Equation (3).

If the neighbor agents have any membership degree to the extreme of time input, they return \(-1\) for each membership parameter. If there is no membership degree to the extreme variable, then the neighbor agents send the three membership degrees of the sensor input using Equations (3) and (2). It is also assumed that the nodes cannot establish a connection with any node more than 10 meters away. Considering this scenario and its constraints, Equation (3) was defined in terms of distance and time.

Therefore, distance \( d \) has the limits of \( d \in [d_{\text{min}}, d_{\text{max}}] \) where \( d_{\text{min}} = \varepsilon \) and \( d_{\text{max}} = 10 \). As mentioned in Section 5.3, \( t2 \) represents the membership degree of the long variable limited by \([0, 1]\). As the \( k \) should be more than 1, the \( k \) value should still 1 in case \( d \) and \( t2 \) values are minor. In this way, the \( k \) value represents the power of trust for the agent(s) requesting sensor data from the other agents. Naturally, it is assumed that trusting the sensor data to the closest agent is convenient, but data timeliness is also a determining factor in this case.

\[ k = ((1 + d/10) + t2), \text{ where } \{ k \in \mathbb{R}^+ : k > 1 \} \]  

If the time is exceeded and no solution is found, the timer artifact sends a signal to the agent and calls for the technician. In Section 6, the study is discussed, the paper is concluded, and future works are presented.

6. Discussion and Conclusions

In this section, the study is discussed, the conclusions are drawn, and future work is presented.
6.1. Discussion

This study uses fuzzy logic-based BDI agents to achieve interoperability between CPS and IoT paradigms. The multi-agent system paradigm was used as an integral approach. While agent-based programming helps communication and organization between the multiple complex CPSs, fuzzy BDI techniques can be a way to tackle uncertainty that can emerge from both internal interactions and external events. The fuzzy BDI approach is also suitable for advancing the decision-making capabilities of self-adaptive systems where unpredictable events have a high percentage of occurring. Fuzzy logic was proposed as an enhancement to these systems. As another enhancement, monitoring and sensor sampling errors can be reduced using multi-fidelity methods, such as fusion and filtering. It is believed that the fundamental theory of fuzzy logic and BDI architecture should be combined based on systematic procedures and evaluated using multiple case studies. However, before starting application-based methods, theoretical fundamentals should create a basis for future studies. The study benefited from Jason and CArtAgO platforms in the scope of the agents & artifacts approach. The distributed scenario was limited because the software agents required an organizational perspective that could be achieved using the Moise platform. The distributed topology was also kept small to show how the fuzzy logic foundations could be applied for the belief reduction of an agent. As physical constraints and rules should be considered before making a decision, we defined a formula based on distance and time.

During the integration phase, a few challenges were encountered. As there is no single systematic approach to designing a fuzzy logic-based system, domain knowledge is required to determine the fuzzy functions. The creation of the rule set is also time-consuming. As the number of functions increases, the number of rules also increases and it becomes hard to maintain. The trial and error approach was used to tune the system for more than a few rounds to achieve the required results. This created a burdensome verification phase and consumed time. Therefore, a domain expert is required to diminish the design endeavor.

The ordinary type-1 fuzzy logic limited the study. The fuzzy logic controller was implemented using the Mamdani system. If this system needs to be adapted to another domain, it should be redesigned according to the target domain’s requirements with that domain’s expert. The applicability of the BDI reasoning is also a criterion. If the domain only needs to use reactive behavior, then the necessity of the BDI pro-activity is lessened, or the BDI architecture might not be suitable. Moreover, the fuzzy rules and sets have to be re-arranged or recreated according to the new domain’s needs.

From the application point of view, fuzzy logic has been used in various domains. It extends the traditional Boolean logic and mimics human reasoning. It can be used in distributed and mobile systems. In transportation, the mobile robots inside a warehouse may require autonomy, pro-activity, and social skills. These features can be gained via BDI agents. Then, fuzzy logic can smoothen their actions, such as incremental speed-ups or decrement/deacceleration, in case of uncertainties. Moreover, in case of collisions of multiple mobile systems, multiple distance sensor inputs can be fuzzified to obtain a reasonable escape angle and speed. Furthermore, in factory automation, the speed of the pressing machine can be smoothened and arranged according to the product features while the BDI agents control the process phases collaboratively [62]. As a significant advantage of using fuzzy logic to tackle short-term uncertainties, it can be combined with reinforcement learning techniques to create better uncertainty mitigation by addressing the mid-term and long-term unanticipated events. Furthermore, the aforementioned application areas can be enlarged using advanced fuzzy sets. However, their integration into the BDI agents requires extra study.

6.2. Conclusions and Future Work

CPSs are complex systems that interact with the physical world. On the other hand, the IoT is a communication-intensive and data-oriented paradigm. As these systems can be
required to merge, multi-agent systems are proposed as integral approaches to combine these two systems, empowering their capabilities using fuzzy logic to tackle the run-time uncertainties in the short term.

This study aimed to provide novel contributions to hybridizing the IoT and CPS paradigms by proposing the *agents & artifacts* approach as first-class entities for implementing these highly complex systems. Firstly, the study shows the research gap on fuzzy logic and its integration with the software BDI agents. The detailed case study showed how to form these systems from scratch. Secondly, the MAPE-K loop was considered the reference run-time model and it merged with the fuzzy logic controller. Lastly, the artifacts were used to form the components of the FLC. In addition, the fuzzy membership degree was also used to analyze the system’s current status to decide the necessity of triggering an adaptation plan.

In the future, it is planned to use the fuzzy Q-learner [60] to mitigate the long-term uncertainties and Moise platforms, to create an organizational perspective of the agents. Moreover, integration with a simulation environment, such as WeBots [63], will also be considered to ease the physical deployment for testing and applying trial/error methods. The fuzzy logic’s platform-independent feature can be applied in a simulation environment, and uncertain events can be easily created for experimental purposes. As fuzzy logic also has various fuzzy sets, such as type-2, intuitionistic, and Pythagorean, they can also be integrated to gain better results.

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

**Funding:** University of Antwerp: Project no. 45838; and Flanders Make Strategic Research Center: AsistedDfA ICON project.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Not applicable.

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

**References**


32. Cuevas, F.; Castillo, O.; Cortés-Antonio, P. Generalized Type-2 Fuzzy Parameter Adaptation in the Marine Predator Algorithm for Fuzzy Controller Parameterization in Mobile Robots. Symmetry 2022, 14, 859. [CrossRef]


