Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Multi-Agent Systems on Ultra Low Power Platforms

Master Thesis

Jannik William
williamj@student.ethz.ch

Computer Security Group
Department of Information Technology and Electrical Engineering
ETH Zürich

Supervisors:
Naomi Stricker
Dr. Andres Gomez
Prof. Dr. Kaveh Razavi

November 15, 2022

Acknowledgements

I'd like to thank Andres Gomez and Naomi Stricker for their extensive support throughout the thesis. The weekly and spontaneous meetings were always very interesting and helping.

The biweekly meetings with Jomi Hübner and others of the Universidade Federal de Santa Catarina gave me some very helpful insights from the multi-agent community perspective.

I also want to thank Prof. Dr. Kaveh Razavi, the Computer Security Group, the Computer Engineering Group and the chair of Interaction- and Communication-based Systems at University of St. Gallen for providing me with the opportunity to work on this interesting topic.

Abstract

As the interest in IoT increases, lots of embedded devices get distributed. Despite some of them having complex low-level functions like signal processing or machine learning, they typically lack higher-level intelligence. With multi-agent systems in AgentSpeak, there exists a programming language for high-level behaviour, but only for systems with no practical resource constraints. This thesis will look at an existing framework for translating a single-agent AgentSpeak program to C++, evaluate the possibilities for the extension to multi-agent systems and implement them on a modern ultra-low-power sensor platform. The usefulness of the embedded multi-agent framework is demonstrated in a small light control application. Experiments show very promising results regarding energy consumption and communication overhead.

Contents

Chapter 1

Introduction

The interest in Internet of Things (IoT) is increasing year after year. This is fueled through advancements in ultra low-power computing (e.g. recent advancements in the PULP project [1]), efficient wireless networks (e.g. [2]) and energy harvesting (e.g. [3–5]). Wireless Sensor Network (WSN) are an important part in this domain. They consist of many sensor nodes, which measure the environment and then send the data to a central node to process it and reason about actions [6]. There is a trend to make the sensors itself more intelligent with complex filters and machine learning models. But the system overall usually lacks higher level intelligence and autonomy. As the processing capabilities and the efficiency of ultra low-power embedded systems increases continuously, the question of whether it is viable to shift reasoning straight to the nodes to make sensor networks smarter and more independent.

The community of Multi-Agent Systems (MAS) [7] has important concepts and models to let different entities act autonomously and collaborate through communication. One such model is belief-desire-intention (BDI) [8], where the different agents are modelled in a way, that replicates the human decision-making. Agents can act reactive, where they act on events and proactively, where they actively try to work towards a goal with self-initiative. These behavioural patterns creates a desirable separation of concerns in complex systems, that facilitates modularity, reconfigurability and even the testing of individual software components. The area of use however is limited to high power computers, as these frameworks usually resort to high-level software platforms (e.g. Java for JaCaMo [7] or JADE [9]) and require full networking capabilities.

This thesis will try to merge the expertise from the MAS community with that of the low-power community and extend an existing single-agent framework in C++ with inter-agent collaboration capabilities for low-power sensors. This approach allows experts for embedded systems to implement low-level functions for the individual sensors, while experts from the MAS community can define the high-level behaviour of the whole system. With such a strategy, sensor networks can become a lot more self contained and use shared knowledge of the whole network to act intelligently (e.g. offloading tasks to more powerful agents),

resulting in a completely new way of deploying connected sensors.

In Chapter 2, this thesis is put in relation to the current scientific topics and advantages of this solution are compared to other approaches. Chapter 3 describes the theoretical foundation about MAS and introduces the low-power sensor platform, which is used to implement the agents. The system architecture of the single-agent BDI framework is introduced in Chapter 4 with the extensions needed for the multi-agent framework described in Chapter 5. Chapter 6 presents the implementation of the agent framework on the DPP3e platform. The single agent and multi-agent framework is evaluated and analyzed in a demo application in Chapter 7. In Chapter 8, the promising results are summarized and future research topics are briefly presented.

Chapter 2

Related Works

There are lots of work on MAS and low-power embedded systems, but very little on low-power MAS. As of this, there are important challenges to be solved for running communication aware BDI agents on resource constraint devices. Three different key related topics can be identified: traditional MAS, low-power computing and low-power communication. Especially the link from the MAS community to the low-power community poses a challenge, as there does not exist much work in linking both of them.

The scientific society of traditional MAS have lots of experience in bringing autonomy and intelligence in distributed systems. MAS are composed of self-organizing entities called agents. Agents exhibit reactive and proactive behaviour, allowing them to react on environmental changes and taking initiative to strive towards a certain objective. Their social ability, meaning the capability to communicate and exchange knowledge, allows them to jointly solve complex tasks [8]. There also exists higher level patterns, namely organizations, where agents can be grouped to have some sort of regulation and coordination, allowing to bring a structure in a complex environment consisting of heterogeneous agents [7]. The advantages of agents and MAS can be seen in a lot of works, e.g. [10–12]. One major drawback of such MAS systems is their need for high power computers, as a lot of the frameworks, such as JaCaMo or JADE, require Java and full networking capabilities.

In the domain of low-power computing, there has been lots of progress made in making computing more efficient and enable increasingly complex artificial models to be employed on embedded processors. E.g. [13] presents a system-on-chip (SoC) with multiple cores for low-power near-sensor analytics algorithms (NSAAs) and deep neural network (DNN) inference on IoT devices. Also on the side of software, there is constant progress being made to make the devices themselves more intelligent. [14] presents a framework to enable online learning for IoT devices. We therefore have the benefit of having increasingly powerful low-power devices with complex data processing at our hands, but there still lacks scientific works, that explore higher level autonomy of these devices. They usually just sense, infer and report to a central entity to decide on actions.

In low-power communication, we can distinguish between asynchronous and concurrent communication protocols. Asynchronous protocols, such as Bluetooth Low Energy (BLE) [15], is characterized by on-demand transmission of data. This allows very little communication delay, as data can be transmitted at any time. A drawback of this approach is, that it is not possible to broadcast a message to every node in an energy efficient manner. There exists a mesh operating mode in BLE [16], but this mode needs high power, always-on relay nodes to work. For low-power asynchronous flooding network, there exists works, that use very efficient wake-up radios to initiate communication (e.g. [17]). Concurrent protocols work differently. They rely on network-wide time synchronization and time slots. As this approach allows nodes to just wake up for the necessary time, it is much more efficient. E.g. Low-Power Wireless Bus (LWB) [18] utilizes this strategy. This protocol has the advantage of increasing the range through a multi-hop flooding technique and thus can very efficiently broadcast packets throughout the network.

There exists some work, that tried to link the low-power domain with agents. In e.g. [19], there is a MAS implemented in Embedded C or in [20] a mobile agent middleware for self-adaptive WSN in an assembly-like language. The solutions in these works however is application specific and completely ignores the BDI model. The approach in [21] tries to solve similar problems, as proposed in this work, but only considered a single agent without taking low-power requirements into account.

Compared to traditional MAS, the suggested general-purpose BDI agent programming framework runs on resource constraint devices, while retaining the core reasoning characteristics of the BDI model, like the reasoning cycle. In the domain of low-power embedded systems, this agent programming framework brings a higher level of autonomy and intelligence to wireless sensor networks. One of the enabling factors, which allows MAS on embedded devices, is efficient all-to-all wireless networks.

Chapter 3

Preliminaries

In this chapter, some important prerequisites are presented. First, traditional Multi-Agent Systems (MAS) with its core concepts are presented. In a second part, the platform, where the agents are implemented is introduced. It is a state-of-the-art sensor platform with multiple processors to separate tasks.

3.1 Traditional Multi-Agent Systems

3.1.1 Agents

To understand MAS, it is necessary to first understand the term *agents*. According to [22], agents are entities (e.g. pieces of software), that can perceive the environment and act on it. But how do they relate to other types of software?

We can classify programs into two different categories: functional and reactive. The functional program paradigm tries to solve problems in a declarative manner ("everything is a function"). The sphere of influence is very narrow and clearly defined (e.g. a data compression tool takes in data and outputs the compressed data).

If we look at more complex programs, such as the operating system of a smartphone, the functional approach cannot be applied, since there is no clear input and output. The behaviour depends on various internal states (e.g. running applications) and events (e.g. user input requesting the deletion of a file). Such systems, with their main characteristic of maintain interaction with their environment, are called reactive systems.

Agents belong to reactive systems [8]. What distinguishes them from other reactive systems is their autonomy. This is achieved, when an agent can be given a goal (or desired state), and it pursues actively, while it decides on its own, how it achieves it. The main characteristics of such agents are their autonomy, proactiveness, reactivity and social ability [23].

Autonomy exists in various degrees. From coffee machines, that only do what they are told to do up to humans, which are fully autonomous. Agents are somewhere in between. Goals can be delegated to them and they figure out themselves, how they should achieve them. As it would be too complicated to have them figure out everything entirely on their own, a set of plans is provided for the agent. To now achieve a goal, an agent combines different plans to an overall plan.

Agents are required to be **proactive**, meaning that they actively try to work towards a goal. On the contrary, we have passive programs, that only do something, when they are called.

Another characteristic of an agent is their **reactivity**. When the environment changes, an agent should react on that and possibly choose a different way in reaching the goal.

Finally, the agent should be able to coorperate and coordinate with other agents. To be regarded as **social ability**, the communication should not only be the exchange of simple measurement information and commands. It should be a knowledge-based information exchange (exchange beliefs, goals and plans).

3.1.2 Belief-Desire-Intention

A common design model for agents is the belief-desire-intention (BDI) approach, which is based on human behaviour. The internal state of an agent is stored in these three data structures.

A **belief** is the knowledge (does not have to be truthful), that an agent has about the environment. This can e.g. be the current room illuminance.

Desires are options, that an agent might take. It is not committed to those and they can be contradictory.

If an agent commits to one of its options, it becomes an **intention**. Now the agent works actively towards its commitment.

To make decisions and take actions, there needs to be some kind of mechanism in the agent. This is called **practical reasoning**. In the BDI model, the reasoning works as follows [8]:

1. update agents beliefs based on the environment
2. decide on an intention
3. compile a plan to achieve the intention
4. execute plan

3.1.3 AgentSpeak

AgentSpeak [24] is a programming language, which can be used to represent an agent in the BDI model. An implementation with extended features is Jason, which is used in this work. An AgentSpeak program starts with specifying the initial beliefs an agent has of the environment and initial goals, the agent pursues. After the initial state of the agent, its plans are stated. They consist of an event, that triggers the plan (e.g. a belief or goal addition). A plan can have a context, meaning a condition, which has to be met (evaluates to true). Finally, a plan has a context, where events can be generated and/or actions invoked. There are two types of actions, normal actions and internal actions. Normal actions have the characterization, that they change the environment. Internal actions on the contrary do any type of processing in the agent's mind. Examples are e.g. `.wait(5000)`, which suspends an intention for 5 s. Another important example is `.broadcast(tell, belief)`, which broadcasts a belief amongst all agents. This internal action is key to multi-agent cooperation. In Listing 1, there is an example of a simple AgentSpeak program.

```
1  // Initial beliefs
2  belief .
3
4  // Initial goals
5  !goal .
6
7  // Plans
8  +belief: condition <- action .
9
10 +!goal <- .internal_action .
```

Listing 1: There is an initial belief `belief` and initial goal `!goal` for the agent. They create the events `+belief` and `+!goal`. For both of the events, there are plans, which the first one has a condition (another belief, which has to equal to `true`, that the plan is executed) and the action `action` in the body. The second plan is triggered by the goal addition `+!goal` and invokes the action `.internal_action`.

3.1.4 Illocutionary Force

The information exchange between agents is knowledge based. So the communication has to somewhat represent different types of interactions. Lets e.g. look at the phrase "the light is on". It tells a listener, that the light is on. Basically, it is an exchange of a belief about the state of the light. If we however look at the phrase "turn on the light", commands the listener to turn on the light. In other words, the listener is told to achieve a certain state, namely that the light

is on. These acts of speech can be systematically listed. The most important used in this work are:

- `tell`: Inform the other party about something, that is true (belief addition)
- `untell`: Inform the other party about something, that is not true anymore (belief deletion)
- `achieve`: Command the other party to reach a certain state (goal addition)
- `unachieve`: Command the other party to not reach a certain state anymore (goal deletion)

3.1.5 Proactivness vs. Reactiveness

The terms proactive and reactive are already introduced in Section 3.1.1. As they are important concepts for the intelligence of agents, it should be clarified, how they are represented in an AgentSpeak program. Both behaviour patterns are already indirectly introduced in the example of Section 3.1.3. Proactiveness (goal-directed behaviour) is directly represented by the the plan, which is triggered by the goal addition `+!goal`. The reactive nature is exhibited by the event change induced plan (it reacts to the environment).

3.1.6 Minimum Requirements for a Platform Running Low-Power Agents

To take advantage of single agent and multi-agent systems, we need a low-power platform with enough computing power and memory to reason and save plans. The platform also needs to have some kind of low-power communication means. A suitable platform is presented in the next section.

3.2 Platform

3.2.1 DPP3e

The DPP3e [25] (Figure 3.1) is a modern harvesting based sensor platform based on the Dual Processor Platform (DPP) [26]. The Dual Processor Platform features a separation of core application and communication. This allows for independent development of the different domains. Between the two domains sits a processor interconnect, which is used for asynchronous inter-processor communication. In Figure 3.2, the data flow is visualized.

The application domain features an Ambiq Apollo 3 Blue Plus [27] with 2 MB flash size, 768 KB of SRAM and a clock speed of 48 MHz. This provides enough

space for the BDI framework with complex logic. The Arm Cortex-M4 based processor is tailored for low-power application, as it features an active power consumption of 6 µA/MHZ (950 µW at 48 MHz and 3.3 V) and a 1µA deep sleep mode. This microcontroller also supports Bluetooth Low Energy (BLE), which supports low-power short range asyncronous communication.

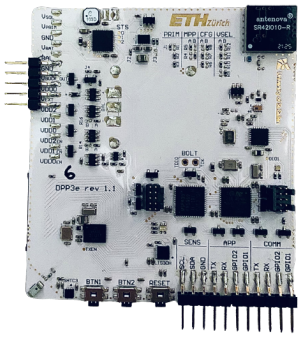

Figure 3.1: The DPP3e harvesting based sensor platform allows the development of intelligent sensor networks

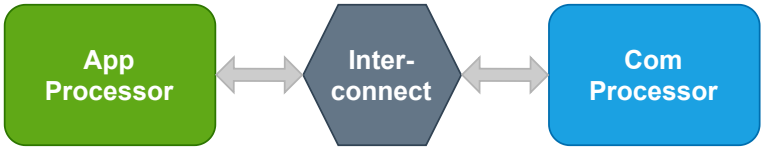

Figure 3.2: BOLT (Processor Interconnect) acts as a asynchronous data queue between the application and communication processor

3.2.2 Low-Power Wireless Bus

To let agents communicate, a robust low-power, bidirectional means of communication is needed. Low-Power Wireless Bus (LWB) [18] supports all-to-all multi-hop, low-power wireless network based on synchronized network-wide floods (see Figure 3.3). This enables energy efficient communication between all the agents in the network. The broadcast happens in communication rounds with a fixed period, where each participating node is able to send their data in the corresponding node slot. In each slot, the message is broadcasted throughout the network (all-to-all communication).

The broadcast is based on flooding and time synchronization mechanism of Glossy [28]. The flooding happens in multiple hops, where each node in range is receiving the data and retransmits it in the next hop. In Figure 3.4, an example of a message flood is visualized.

In the communication domain of the DPP3e, an STM32L433 microcontroller

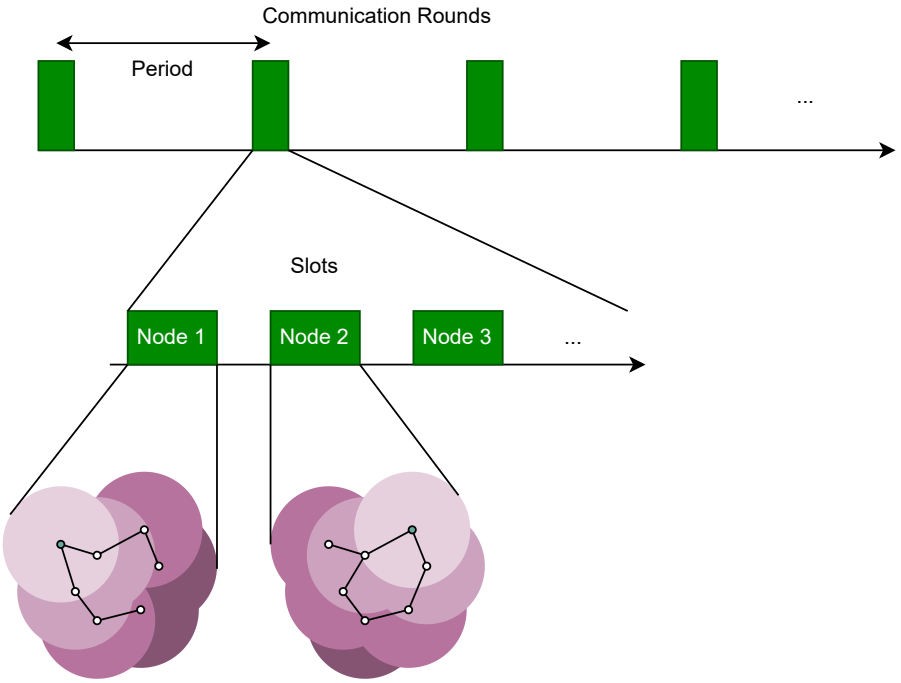

Figure 3.3: The Low-Power Wireless Bus (LWB) works in communication rounds, where every node has the opportunity to transmit data. For that, every node receives a slot in a single round, which is accompanied by a network flood.

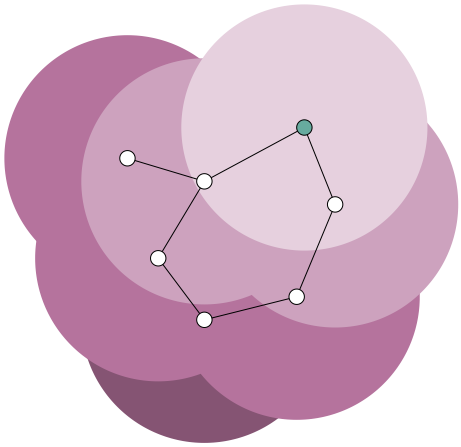

Figure 3.4: Glossy is a network flooding protocol, where every node receives data from a previous node and transmits it further to other nodes, which did not yet receive the data. This allows to transmit data over wide distances without requiring high transmission power.

in conjunction with the Semtech SX1262 LoRa transceiver with a chip antenna is used. This allows the platform to perform long-range communication. The bandwidth is limited compared BLE, since the LWB protocol has a low duty-cycle (long communication periods) to save energy. In each slot, the maximum amount of data, that can be sent, is around 120 Bytes (including necessary header). With a communication period of 15 s, a node is able to send around 8 B/s.

3.2.3 Processor Interconnect

The application and communication domain exchange data over a processor interconnect called Bolt [29]. It is a asynchronous messaging protocol based on queues. This interconnect allows the application and communication domain to perform its tasks independently from one another (memory and clock independent).

The DPP3e platform with the Ambiq Apollo 3 Blue Plus are excellent platforms to implement low-power BDI agents, since they are energy efficient and have ample memory and support multiple communication protocols.

Chapter 4

Low-Power Single Agent System

Dos Santos [30] introduced a Jason-based [8] lightweight single agent framework for embedded devices, which implements the principles mentioned in Section 3.1.1 without the social ability. Jason is an interpreter for an extended version of AgentSpeak (Section 3.1.3). My first goal of this thesis is to integrate this embedded BDI framework on a Ultra-Low Power (ULP) embedded platform with tight processing and memory constraints. The concepts presented in this chapter will be published in the form of a workshop paper with the title *Increasing the Intelligence of low-power Sensors with Autonomous Agents* at the *Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things* at *SenSys '22*.

The framework is devided into two main components. One of them is a translation engine, which translates an AgentSpeak program into highly efficient C++ code. The other component is the BDI runtime library tied together with the hardware-dependent code, which is compiled in an executable binary (Figure 4.1). The high-level development process is to program an agent in AgentSpeak and implement hardware-dependent code (perception, action functions, real-time OS, etc.) in C/C++.

4.1 Translation Engine

The main task of the translation engine is to convert an AgentSpeak program into C++. It connects the belief base to perception update functions and actions to their corresponding function according to a naming convention. The AgentSpeak program is translated into a special C++ structure representing the initial states and plans of a single agent. To improve the memory and runtime performance, the names of beliefs and goals are converted to an integer value. Action names are also lost, as they are tied to the corresponding function addresses. As the integer

is 8 bits in size, the number of distinct values is limited to 256. In contrast to the full Jason implementation of AgentSpeak, only propositions, instead of predicates (allows the use of variables) are allowed. This however does not limit us in developing reasonable complex agents, if we consider the limited computational power of embedded devices.

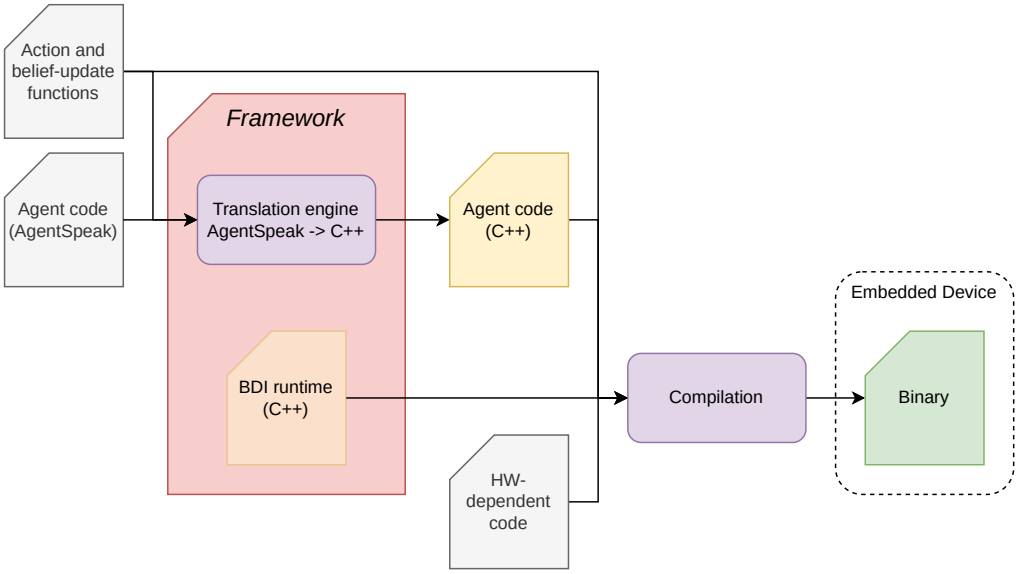

Figure 4.1: The AgentSpeak program together with the action and belief-update functions are processed by the translation engine, which outputs the C++ equivalent of the agent code. Together with the BDI runtime and the hardware-dependent code, it is compiled to an executable binary.

4.2 BDI Runtime

The BDI runtime contains the reasoning algorithm of the agent. As a simplified version of the Jason reasoning cycle, there is no inter-agent communication implemented. Figure 4.2 visualizes the detailed reasoning process of the embedded BDI runtime and Algorithm 1 the reasoning loop as executed by the microcontroller. It consists of following steps:

1. *Update the belief base and generate events from new perceptions*
   In this step, sensors percept the environment in the belief update functions. The measured values are converted to boolean beliefs (e.g. a threshold comparison of the room illuminance). An event is generated upon a belief change and added to a queue.

2. *Event selection*
   A single event is selected and processed.

3. *Retrieval of relevant plans*
   All relevant plans to deal with the event are selected.

4. *Determination of applicable plans*
   The relevant plans are filtered based on their context.

5. *Selection of an applicable plan*
   This step selects the first applicable plan.

6. *Selection of an intention for execution*
   An intention from the intention stack is selected. It has to be active and must not be suspended (waiting for an event.

7. *Execution of one step of an intention*
   This step executes a single instruction of an intention. Supported are the addition and deletion of a belief, adding a subgoal, adding new goals and actions.

As one still might want to use communication (e.g. to control actuators), one can leverage the action functions to enable non-agent communication.

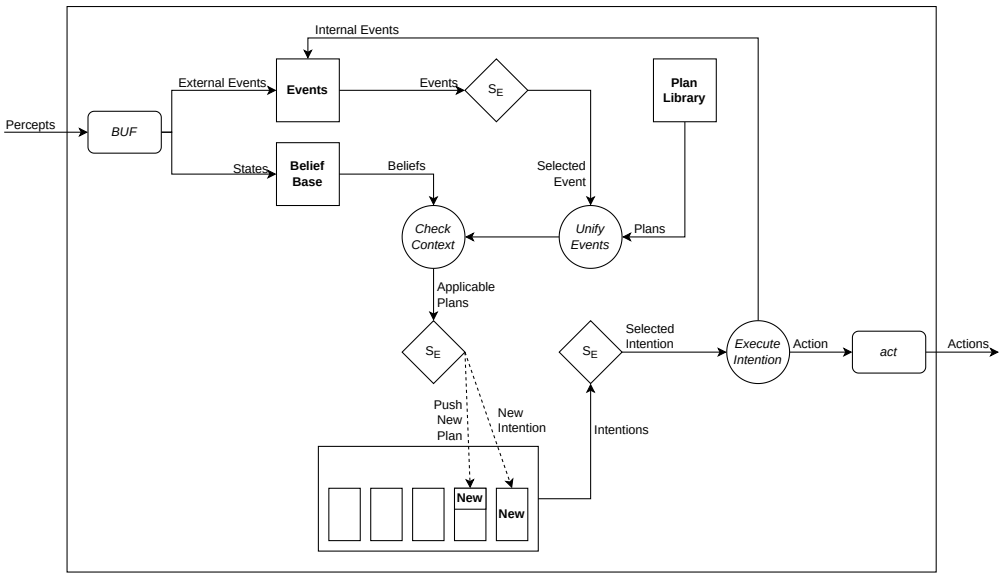

Figure 4.2: The practical reasoning cycle of the embedded BDI runtime is based on a simplified version of the full Jason reasoning cycle. It was developed with low-power embedded devices in mind and hence it only offers the basic features. This figure is taken from [30].

Algorithm 1: Reasoning loop as executed by the microcontroller

```
Add initial beliefs to belief base;
Add initial intentions to intention stacks;
while true do
    Perceive the environment;
    Update belief base based on changes in environment;
    Generate events based on changes;
    if Event queue is not empty then
        Select event to be processed;
        Select relevant plans;
        Filter plans based on context;
        Select first plan;
        Add instructions of the plan into intentions;
    end
    if Intention stacks are not empty then
        Select single instruction of an intention;
        Execute this instruction;
    end
end
```

4.2.1 Reasoning Frequency in Low-Power Environments

It is important to decide, at which frequency the reasoning cycle (shown in Figure 4.3) is executed on the microcontroller, as an agent should be responsive, but the device should consume only as much energy as necessary. Since the frequency of running a cycle depends on the application, no general statement can be made (there is an example in Section 7.2.1). In Section 5.3, another, more sophisticated approach is discussed.

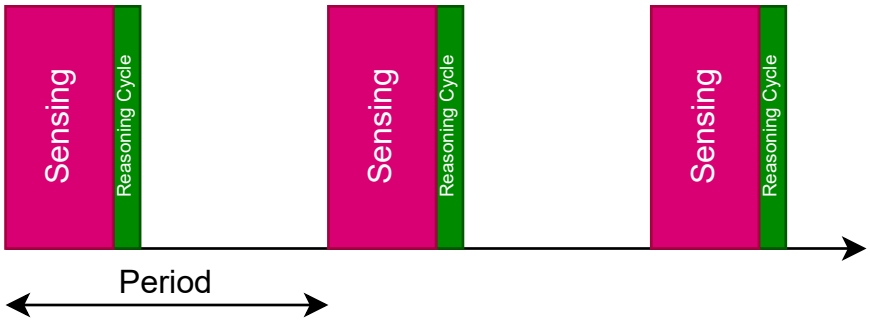

Figure 4.3: The period of the reasoning cycle needs to be small, as in one cycle, there is only one event processed and one instruction of an intention executed. As of this, there needs to be multiple cycles executed to come to a decision.

Chapter 5

Cooperative Multi-Agent System

The main goal of this thesis is the extension of the embedded BDI framework to support multi-agent cooperation. I will discuss in this chapter, what modifications need to be done in the translation and runtime parts.

5.1 New Features on the Agent Level

The simplest addition to the simplified AgentSpeak language to make the framework multi-agent capable is the `.broadcast` internal action. This method allows to broadcast a message amongst all other agents. With the illocutionary forces (Section 3.1.4) of `tell` and `untell`, there is a basic agent communication protocol to exchange beliefs, which is sufficient for basic interactions.

Another, more subtle detail, is the required support for arguments for the `.broadcast`-action. This needs adaption in the translation engine, which needs to parse the arguments and convert it to different data types (in Jason, they are usually of the type `Predicate`, but this is too broad for resource-constrained devices).

5.1.1 Reasoning Cycle

The reasoning cycle needs additional steps (see colored steps in Figure 5.1) to process incoming messages from other agents. These messages are available in a mailbox, which provides an interface to the hardware-dependent code to put in messages (incoming arrow in Figure 5.1). At the start of a reasoning iteration, the mailbox is checked for new mail within *checkMail*. In a subsequent step (S_M), a single message is selected (in Jason, this step can be modified, but the embedded BDI framework implements the default behaviour of selecting the first message). A social acceptance step (*SocAcc*) is performed on the message. This step should

filter out unwanted messages. As this step is not important for simple and small applications and only gains importance in big MAS with an organizational layer, it marks every received message as socially acceptable. Finally, the belief base is revised in *BRF* based on the received message. The dashed arrow from the belief update function (*BUF*) to the belief revision function indicates, that updated beliefs are taken into account, when generating events. The sending of messages is not shown in the figure, since it is part of the *act*-step. As it will be shown in Section 6.2, the sending of a message is not performed in the high level reasoning cycle, but will be part of the action functions (see Figure 4.1, Action and belief-update functions).

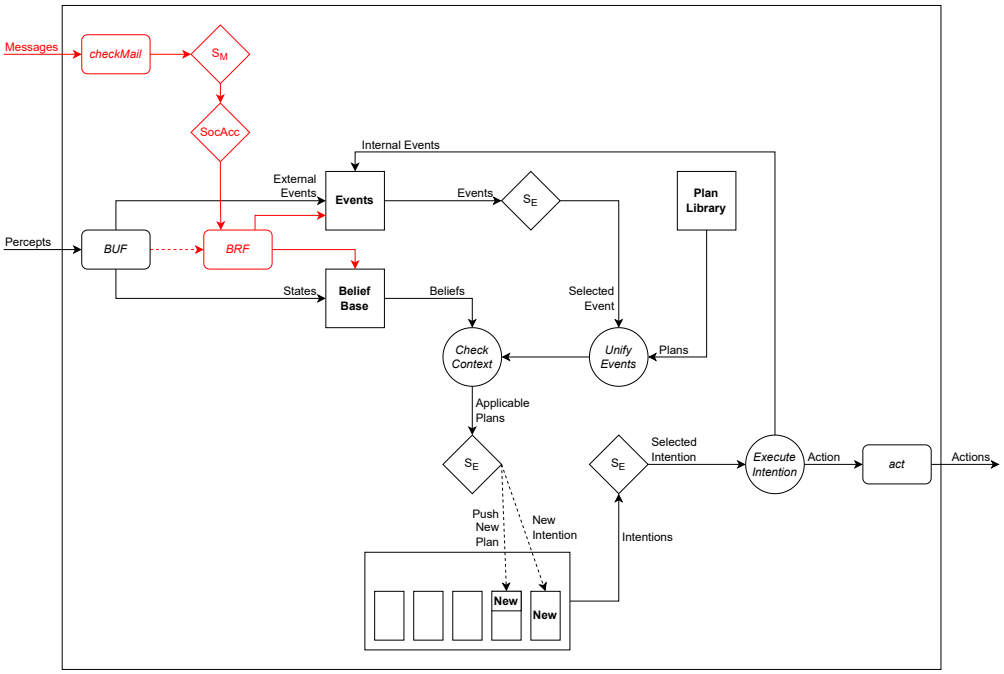

Figure 5.1: This shows the minimal additions made to the BDI reasoning cycle highlighted in red.

5.2 Derive a Common Language Between Agents

The translation engine of the single agent framework translates propositions in non-reproducible way. It enumerates the proposition based on their first appearance in the AgentSpeak program. This works fine with single agents, as these use the propositions only for internal operations. If however multiple agents want to communicate with each other, these proposition values need to be passed around and could lead to a misunderstanding. In Figure 5.2, an example demonstrates this issue.

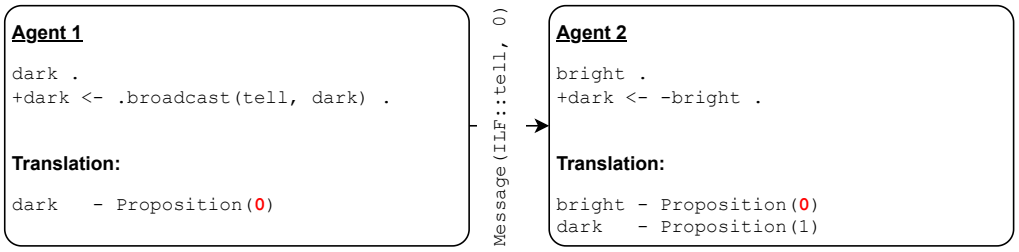

Figure 5.2: Agent 1 wants to tell the belief `dark` to Agent 2. As Agent 2 translated this belief differently, it falsely translates it to the belief `bright`.

The most promising solution is the use of hashed proposition names. As hashes produce reproducible output given a fixed input, same proposition names get translated equally in every agent program (Figure 5.3). Another advantage is, that this solution does not depend on a global state or require all agents in an environment to be compiled at the same time. However special considerations on hash size and function has to be made to prevent hash collisions (when different inputs produce the same output). Collisions within the same agent can however be detected at compile time. As demonstrated in Figure 5.4, the previously mentioned issue is resolved.

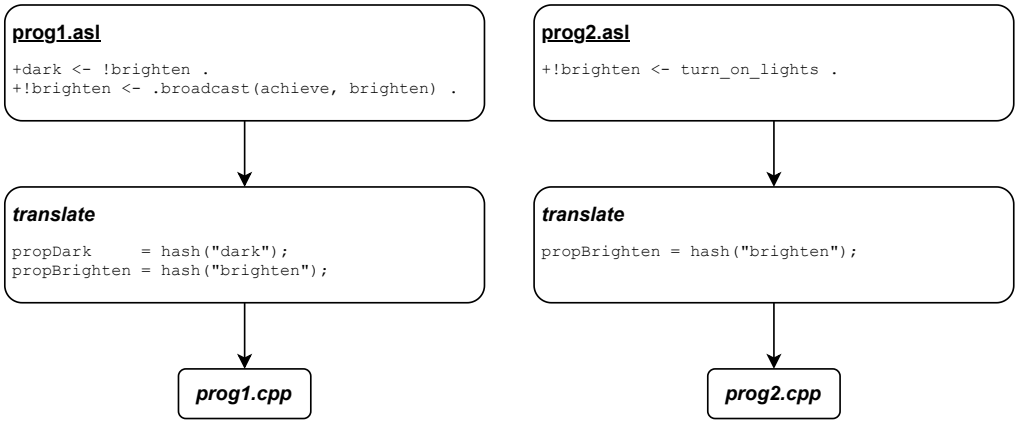

Figure 5.3: In the translation engine, instead of enumerating propositions in the order of appearance, they get hashed. This results in propositions having the same name in different programs are represented by the same numerical value, allowing inter-agent communication.

5.3 Extension to a Full Decision Cycle

In traditional MAS, the reasoning cycle can be executed with a high frequency, as no tight resource constraints exists. Executing at a low frequency, as described

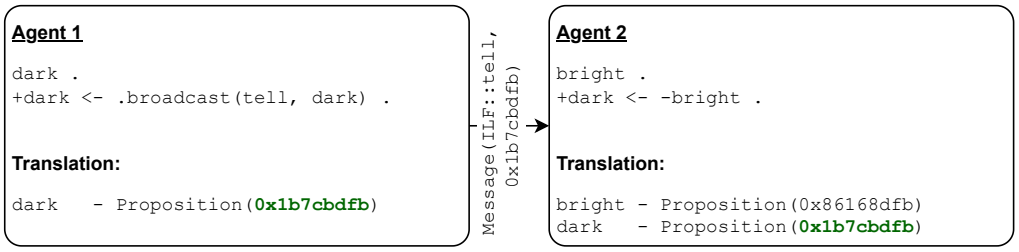

Figure 5.4: The same example introduced in Figure 5.2 is now working.

in Section 4.2.1, creates unnecessary computational delays, because the information to decide on an action is already available most of the times. It would be beneficial to execute the reasoning loop until it is decided on an action and then suspend execution for a certain time (see Figure 5.5). This lets the agent be responsive, once there is information available to make a decision. But it is also saving on energy, because there are not that many empty reasoning cycles happening. The problem is, how do we decide, if an agent came to a decision and is allowed to sleep. The simplest idea one could have, is to stop, if there are no events pending, all intentions and all messages are processed. There are however some programming patterns, (e.g. [8, Chapter 8]), where we have plans in the form of Listing 2. In most of the cases, we want to execute this plan only once in a decision cycle only to make sure, we keep a state in a changing environment. In this example, the intention base is never empty the same time the event base is empty. This results in the breaking condition never becoming true. A solution for this problem is to only check the event base for events other than events from achievement goals. There needs to be more research made on this topic, as it is only tested with the application developed in Chapter 7 and is not a universal solution. For now, the developer has to make sure, that the breaking condition complies with his requirements.

```
1  +!goal <- action1; action2; !!goal .
```

Listing 2: This adds the event `+!goal` (does not wait for the event to be processed, since the event is added from an achievement goal) after executing the plan. Thus the event base and intention base are never empty at the same time.

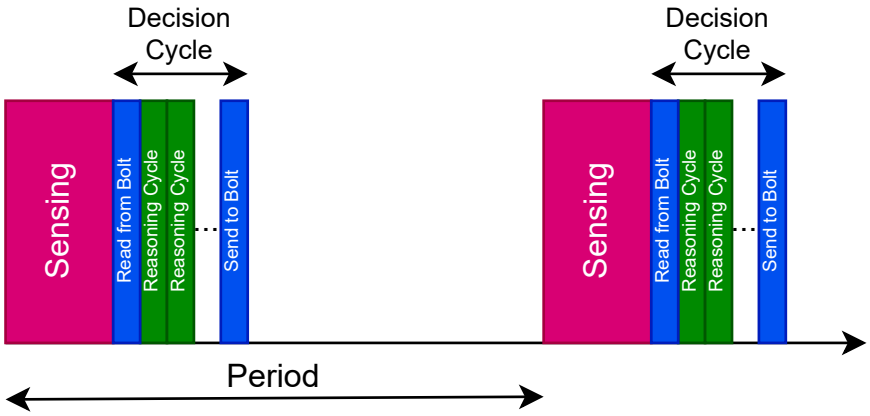

Figure 5.5: The decision cycle executes the reasoning cycle, until the breaking condition triggers a break in the loop.

Chapter 6

Implementation

This chapter describes the implementation of the single agent and multi-agent application on the DPP3e platform. In Section 6.2, the extensions of the embedded BDI framework is discussed with a focus on the implementation details.

6.1 Single Agent Implementation

To introduce the embedded BDI framework to the Ambiq Apollo platform, firstly an out-of-the-box single agent application was developed on a Sparkfun Edge Board. The high level application structure is, to have an agent running on the application processor of the DPP3e, which communicates actions over BLE advertising packets to a server, which controls the actuators. This means of communication is efficient for this application, as there is only a one-way communication, which makes expensive listening redundant. The DPP3e can have multiple sensors connected, which are read as part of the belief update functions. Since the Ambiq Apollo 3 Blue Plus supports BLE communication, Bolt and the communication processor is not used.

The software is based on the open source Real-Time Operating System (RTOS) FreeRTOS [31], which is integrated in the AmbiqSuite SDK [32] (Software Development Kit (SDK) for Ambiq Apollo microcontrollers). It is divided into three different tasks. The setup of the system is done in the `Setup` task. This also starts the application tasks. The `SensorTask` starts a timer, which periodically calls a function, which percepts the environment with a Vishay VEML7700 lux sensor [33] and an AMS AS7262 spectral light sensor [34] (with the sensor driver implemented within the scope of this thesis) and does a single reasoning cycle for the agent. Communication is handled by the `RadioTask`. This task starts a handler for the BLE stack. A packet can be sent through invocation of the function `bleAppSendData`. A Raspberry Pi receives the BLE commands and controls the actuators.

The periodic function is called every 3 s. As already discussed in Section 4.2.1,

intentions need multiple steps to be processed. This leads to a computational delay. This effect is mitigated through the fact, that once it is decided on an action, it is executed almost immediately, as BLE has almost no delay because of its asynchronicity.

6.2 Multi-Agent Implementation

We can now use the full potential of the DPP3e's separation of application and communication domain. The embedded BDI runtime has to only care about reading and writing messages to Bolt (Section 3.2.3). In detail, in every reasoning cycle, the agent checks the indication line of Bolt, if there are new messages in the queue. If there are, it reads them and stores it in the mailbox for further processing by the agent. The communication domain handles the communication between other nodes and does message concatenation (in one full decision cycle, multiple messages could be generated by an agent which are then concatenated to a single message packet to transmit over LWB).

In the multi-agent implementation, the full decision cycle (Section 5.3) is implemented. This allows us to push the wake-up cycles further apart to save more energy in the application domain, as computational delay issues are eliminated.

6.2.1 Full Decision Cycle

Because of the full decision cycle, the wake-up periods could be increased to 15 s. To realize the full decision cycle, the reasoning cycle (`agent->run()` method) is embedded into a loop with following break condition:

- only events of the type `EventOperator::GOAL_ACHIEVE` are in the event queue and
- the intention base is empty and
- the mailbox is empty

Additionally, the number of execution is limited to 256. This should prevent, that unfortunate plans keep the decision cycle active for an unlimited amount of time. This number can be modified to meet the application's requirement.

6.2.2 Communication

A routine, which reads messages from Bolt and puts it into a mailbox, is added to the periodical task. The sending part is implemented in the function `internal_action_broadcast`, where the message is serialized and sent to

```
typedef struct {
  uint8_t               node_id:5;
  uint8_t               cmd:3;
  uint32_t              period;
  uint64_t              timestamp;
} bolt_header_t;

typedef struct {
  bolt_header_t         header;
  union {
    struct {
      uint32_t          proposition;
      uint8_t           illoc_force;
    } bdi_msg;
    struct {
      uint32_t          action;
    } bdi_action;
  };
} apollo_message_t;
```

Listing 3: The header of Bolt-message indicates the sending node, which type of message it is and some timing information. The type can be a timesync-, agent- or an action message (13 bytes). The content of the message depends on the type. An agent message contains a proposition and an illocutionary force (5 bytes), and an action message contains an integer representing an action (4 bytes).

Bolt. Whenever an action is performed, a message gets send to Bolt too, but as a different type (this is no agent message). In Listing 3, the message structure is defined.

The software of the communication domain is divided into three main tasks: the communication task, a pre- and post-communication task. The communication task handles the LWB communication, the pre task is responsible for the packet generation and the post task to process the received messages. The pre task reads agent messages from Bolt and composes it to a single LWB-message (see Listing 4). This allows the node to send multiple agent- and action-messages in one packet. The post task does the opposite. It receives messages from other nodes and splits it up into individual messages. Agent messages are sent to Bolt, where an agent can read it. If the node is configured as the gateway, it forwards the action to be taken care of by an external host program.

```
typedef struct {
  uint8_t               cmd;
  union {
    struct {
      uint8_t           illoc_force;
      uint32_t          proposition;
    } bdi_msg;
    struct {
      uint32_t          action;
    } bdi_action;
  };
} dpp_mas_bdi_t;

typedef struct {
  uint8_t             len;
  dpp_mas_bdi_t       msg[DPP_MAS_BDI_ARRAY_MAX_LEN];
} dpp_mas_bdi_array_t;
```

Listing 4: An LWB-message contains an array of Bolt-messages, which is either of the type agent or action. The size of the LWB-message is determined by the number of individual messages (`1 byte + 6 bytes * numberOfMessages`).

6.2.3 Embedded BDI Framework

Translation Engine

The parsing part of the translation engine is extended to recognize internal actions (new type: `BodyInstruction.BodyType.INTERNAL_ACTION`). Since arguments are necessary for `.broadcast`-actions. This is done by first storing each argument in a list of strings for each body instruction. When writing the C++ file (in `HeaderCreator.java`), it is checked, if the internal action equals to `.broadcast`. If it is the case, the arguments are converted to the specific C++ types (illocutionary forces as `IllocForce` and the proposition as integer in the class `Proposition`). The arguments themselves are stored in a special object of the type `ActionArgument`, which is a wrapper for any argument type.

The proposition translation is modified to implement a CRC32 hash instead of enumerating the propositions. This is done in the function `hash_proposition` inside the `HeaderCreator`-class. It is also checked, whether there are collisions. This does not solve for inter-agent hash collisions, but can catch them, if they happen inside an agent.

In the original framework, the propositional value is represented by an 8-bit integer. This is too small for a hashing function, as the collision probability is around 0.39 % for 2 hashes. Therefore the integer size is increased to 32 bits. This has only a collision probability of $2.32 \cdot 10^{-8}$ %.

In the configuration file `agent.config`, there is now the possibility to set the mailbox size and the minimum belief base size. The specification of these sizes is necessary to prevent buffer overflows on the small memory. In the single agent framework, the belief base did not need to be specified, since all beliefs were known in advance. That changed in the multi-agent version, since an agent can always receive an unknown belief from another agent. It is called minimum belief size, as it has at least the size to store all known propositions.

To check, if the corresponding C++-functions for all actions are present, the functions-file is read and searched for the expected function names. However if a function is commented out, the translation engine still recognized it as present. A small processing step was added, when the file gets loaded, where the comments get removed. This is done by pattern matching with the regular expression `([ \t]*\/\/.*|\/\*(.|\n)*?\*\/)`. It is not a full fledged solution, since it does not ignore strings like `"/*"`, but if the developer has an eye on it, it does work.

BDI Runtime

The red marked additions in Figure 5.1 you can find in two classes. The mailbox is implemented in the `Mailbox` class. It has functions for pushing and popping messages to and from the queue, can check the mailbox for pending messages and check a message for its social acceptance. These functions are called in the `Agent` class as part of the reasoning cycle (apart from pushing messages to the queue, this is done externally). This leads to the updated `Agent::run()` method in Listing 5.

The messages contained in the mailbox are implemented in the class `Message`. It is instantiated with a proposition and a illocutionary force. It contains the `Message::process_message()` function, which acts as the belief revision function. It currently revises the belief base based on messages with the illocutionary force `IllocForce::TELL` and `IllocForce::UNTELL`. There, it checks the state of a belief and generates necessary events in case of a change. Additionally, there are serialize- and deserialize-functions to convert messages to a byte stream and back to interface with any communication device. This was used to test the framework over UART with manually created messages. In the DPP3e however, the serialization happens in the communication domain on a different microcontroller, where multiple messages can be aggregated (see Section 6.2.2).

The action functions now support arguments (only the ones specifically implemented in the translation engine, which currently is only `.broadcast`). Even if the arguments are only supported for selected functions, the general template for the pointer to action functions changed from `bool (*take_action)(void)` to `bool (*take_action)(VectorQueue<ActionArgument> * args))`. To instantiate an object, one can either pass a proposition or illocutionary force, but

```
1  void Agent::run()
2  {
3    // Update beliefs
4    beliefs->update(events);
5
6    // Receive and select messages
7    if(mailbox->check_mail())
8    {
9      Message * msg = mailbox->select_message();
10     if(mailbox->socc_acc(msg))
11     {
12       msg->process_message(beliefs, events);
13     }
14   }
15
16   // Checks if there are events to be processed
17   if (!events->is_empty())
18   {
19     event_to_process = events->get_event();
20     plan_to_act = plans->revise(&event_to_process, beliefs);
21     if (plan_to_act) {
22       intentions->add_intention(plan_to_act, &event_to_process);
23     }
24   }
25
26   // Runs intention in case there are any
27   if (!intentions->is_empty())
28   {
29     intentions->run_intention_base(beliefs, events, plans);
30   }
31 }
```

Listing 5: The updated C++ code for the reasoning cycle. Lines 6 to 14 are new. The belief revision function is encapsulated in the function call of Line 12.

other types can be easily implemented by adding an additional constructor in the `ActionArgument` class and an additional type in `ActionArgumentType`. Arguments are stored in the objects of `BodyInstruction`, which represent one instruction of a plan body.

In the `EventBase` class, an additional function to check, if there are only events of a specific type in the event base is added. In the breaking condition of the full decision cycle (see Section 5.3), it is used to determine, if there are only events of the type `EventOperator::GOAL_ACHIEVE` in the event base.

As the number of allowed proposition values exceeds 256, it is also required to increase the size uf the variable, that holds a queue size. They are changed from the type of `uint8_t` to `uint32_t`. These variables are used to initialize the `VectorQueue` class template (based on the builtin class `vector`). As a 32 bit

processor architecture has usually a 4 byte memory alignment, it should not have any memory impact.

6.2.4 Flexibility of Extending the Framework

The code of the framework is structured in a way, that allows easy extensions of other more complex actions. E.g. the more sophisticated `.send`-action (where a message is dedicated to a specific agent) needs its own definition in the `write_header()` function in the `HeaderCreator`-class with the specifications of the arguments. The recipient could be defined as own argument type with an additional field in the struct defined in Listing 3. In the C++ file, where the actions and belief update functions are defined, a function called `bool internal_action_send()` can be added where a message is sent to Bolt (similar to the function `bool internal_action_broadcast()`.

To support more illocutionary forces, the handling of the message needs to be implemented in the function `Message::process_message()`. There it is just a matter of adding cases to the `switch`-statement. For more complex illocutionary forces, one needs to extend the message class with additional member variables (e.g. `tellHow` needs a plan in the message).

Chapter 7

Experimental Evaluation

In this section, I will evaluate the single agent and multi-agent framework by designing a demo application. This application is then implemented both as single agent and multi-agent version. Both are measured and evaluated. In the last section, I will compare and contrast both implementations.

7.1 Case Study: Controlling Room Illuminance with Intelligent Agents

This demo application should show that the framework, whether it is single agent or multi-agent, is able to efficiently and autonomously control systems composed of sensors and actors in a proactive manner. It should however not be too complex, such that the internal mechanics of the agents is fully comprehensible, to prove proper functioning of the system.

A simple system to show above design goals is the control of the illuminance in a room autonomously. As the room is a dynamic environment with changing conditions, the agent should proactively preserve the illuminance as eco-friendly as possible (e.g. preferring outdoor light over artificial light). It can do that by triggering following actions:

- turn on or off overhead light
- raise or lower blinds to let sunlight into the room

The agents can perceive the room environment by following channels:

- indoor illuminance in lux
- outdoor illuminance in lux

The second goal is to adapt the illuminance to the users preference. This is done through an additional sensor input (like a switch).

7.1.1 Goal 1: Preserve the Room Illuminance Efficiently

A proactive goal pattern ensures that the agent remains committed to the goal of preserving the room illuminance, whether it should be dark or bright. This pattern keeps checking the environmental state with the instruction of `!!preserve_light` and `!!preserve_dark`. The context of the plans determines whether the room should be brightened up, darkened or the room is sufficiently illuminated (e.g. if it is `bright_inside` and the room should be bright, the agent does nothing). To make the system eco-friendly, it prefers outdoor light instead of artificial light. But outdoor light should only be used, if it is sufficiently high in intensity. This is done through the beliefs `standard_mode_available` and `eco_mode_available`. The standard mode is available, if the lights are turned off and the eco mode, if outdoor illuminance is high enough (i.e. it is sunny). When the agent wants to brighten the room, it considers, which means are available and selects the most eco-friendly solution. This is determined by the order of the plans, as the first applicable is selected.

7.1.2 Goal 2: Update Environment Based on User Preference

The user preference is considered in the context of the recursive plans. If the agent acquires the new belief `user_turn_on` the agent pursues the new goal of bringing the illuminance level of the room to a sufficiently high level. The agent is then commited to this goal, until `user_turn_on` is not believed anymore. On acquiring the new belief `user_turn_off` the agent commits itself to bring down the illuminance, until it is below the threshold, that `dark_inside` is believed.

7.2 Single-Agent Application

This section will evaluate the single agent framework using the demo application introduced above. Firstly, the experimental set-up is explained along with the exact agent application. Then the results with a trace of beliefs, goals, actions and illuminance trace are discussed. The following results will be published within the scope of the workshop paper *Increasing the Intelligence of low-power Sensors with Autonomous Agents* at the *Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things* at *SenSys '22*.

7.2.1 Experimental Set-Up

The single agent application is implemented on the Edge platform from Sparkfun Electronics. It features the Ambiq Apollo3 Blue with 1 MB of flash, 384 KB of SRAM and 48 MHz clock frequency (due to a production delay, the DPP3e could not be used). A BLE radio is also integrated into the Apollo3 Blue MCU for

short one-way communication. There are two light sensors connected over I^2C. An Ams AS7262 [34] to perceive the indoor illuminance and a Vishay VEML7700 [33] to do the same for the outdoor illuminance. In a lab at the University of St. Gallen, a room, which allows digital control of lights and blinds, the experiment is performed. The outdoor sensor is mounted on the outside wall next to the blinds, while the indoor sensor is facing upwards, such that it receives the indoor light most efficiently. The user preference is selected by an accelerometer sensor (LIS2DH12) by inverting the Z-axis. The Sparkfun Edge board is connected over UART to a Raspberry Pi, which receives raw data to log. The BLE module of the Raspberry Pi is used to receive the BLE packets (action commands to turn on/off the lights and to raise/lower the blinds) sent from the agent. The set-up is shown in Figure 7.1.

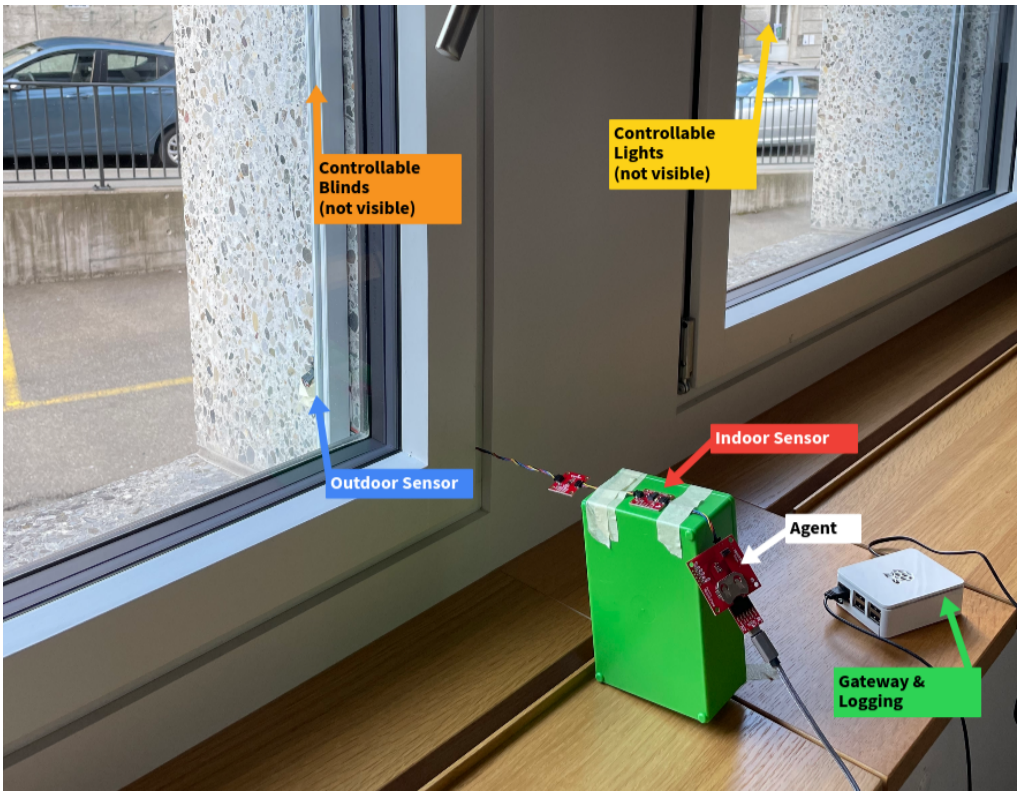

Figure 7.1: The indoor sensor is mounted on a box and is facing upwards. The outdoor one is mounted to the wall and is facing southwestward. The microcontroller contains the agent software and is connected via serial to a Raspberry Pi to log. The actions are sent over BLE to the Raspberry Pi, which can controll lights and blinds.

The agent program can be found in Listing 6. It implements the functionality of the demo application introduced in the first section of this chapter. The `{bright,dark}_inside` and `{sunny,cloudy,night}_outside` beliefs are

updated through the two light sensors, while the user_turn_on/off beliefs are updated based on whether the Z-axis of the accelerometer is positive or negative.

The experiment is performed in the evening, while outdoor illuminance is decreasing. This circumstance allows us to expose the ability of the agent to deal with a changing environment. The light threshold were set to following levels:

- sunny/cloudy threshold: 400 lux
- cloudy/night threshold: 250 lux
- indoor threshold: 30 lux

The reasoning cycle period was set to 3 s. That means, every 3 s, the sensors are read and one reasoning cycle of the agent is performed.

```
1  +user_turn_on <- !!preserve_light .
2  +!preserve_light: bright_inside & user_turn_on <- !!preserve_light .
3  +!preserve_light: dark_inside & user_turn_on <- !brighten ; !!preserve_light .
4  -!preserve_light: user_turn_on <- !!preserve_light .
5  +sunny_outside: user_turn_on <- +eco_mode_available .
6  +cloudy_outside: user_turn_on <- -eco_mode_available ; +standard_mode_available .
7  +night_outside: user_turn_on <- -eco_mode_available ; +standard_mode_available .
8  +!brighten: eco_mode_available <- turn_off_lights ; raise_blinds ; -eco_mode_available .
9  +!brighten: standard_mode_available <- turn_on_lights ; -standard_mode_available .
10
11 +user_turn_off <- !!preserve_dark .
12 +!preserve_dark: night_outside & user_turn_off <- lower_blinds ; !!preserve_dark .
13 +!preserve_dark: bright_inside & user_turn_off  <- turn_off_lights ; lower_blinds ; !!preserve_dark .
14 -!preserve_dark: user_turn_off <- !!preserve_dark .
```

Listing 6: This is the agent program to control the illuminance in a room based on user preference. The agent can proactively preserve the preferred illuminance level.

7.2.2 Results

The experiment lasted for 1208 s. The illuminance trace with the goals and actions from the agent are shown in Figure 7.2. The outdoor illuminance decreased from 431 lux at the beginning of the trace down to 262 lux. The indoor illuminance started at 7.49 lux and decreased to 4.6 lux, while having a high of 213 lux.

The initial state of the room was, that the blinds were down and the lights off. At the beginning, the goal of the agent was, to keep the room dark. Once it was committed to the goal of brightening the room, it triggered the action to raise the blinds (at the 75 s time mark). The delay from having this goal to perform the

action was 25 seconds, meaning 8 reasoning cycles were necessary to come to the decision to raise the blinds (the extra time is due to the communication delay). After 678 s, the indoor illuminance level fell below the threshold, therefore the indoor illuminance condition is considered dark. 9 s later (equals 4 decision cycles), the action got triggered to turn on the lights. Before the experiment ended, the agent was again given the goal to darken the room. This lead to turning the lights off and lower the blinds. This took 51 s in total.

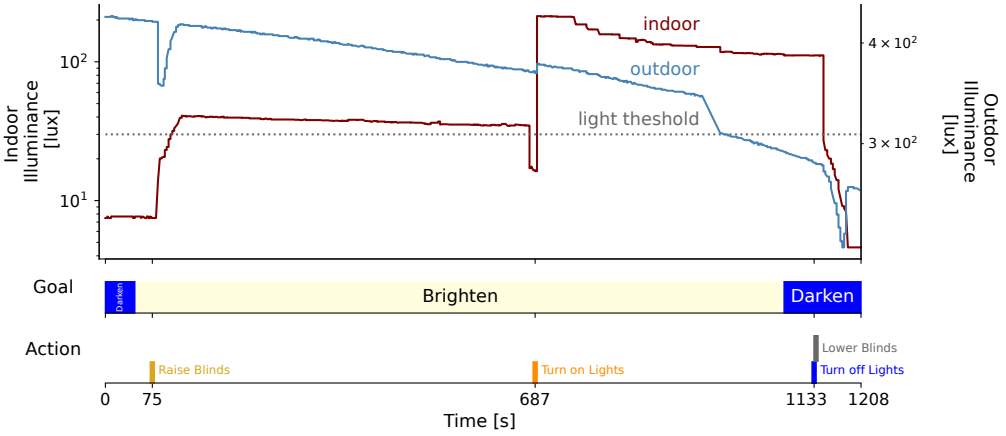

Figure 7.2: The experiment exposed the agents reactive and proactive behaviour by changing the user preference (goal) and a dynamic environment (changing illuminance levels).

Low-Level Metrics

In this subsection, the low-level performance of the agent on the hardware and C++ level is looked at. The agent code consisted of 13 lines of AgentSpeak code (with one plan on each line. There were a combined 22 instructions and 15 beliefs in the contexts. Translating these, resulted in 292 lines of C++ code (removed comments and empty lines). The composition of the binary is summarized in Table 7.1 and Table 7.2.

Memory region	Used Size	Region Size	%age Used
flash:	358'572 B	960 KiB	36.48%
sram:	393'208 B	384 KiB	100.00%

Table 7.1: The agent occupied a bit more than a third of the flash memory and used up all sram.

To measure the energy consumption of various tasks, the RocketLogger [35] was used. This device measured current, voltage and digital pins of the plat-

.text	.data	.bss	total
355'116 B	3'456 B	389'752 B	748'324 B

Table 7.2: In this table, the size of the different memory sections, the agent uses is stated.

form. The system was powered with 3.3 V with two different channels of the RocketLogger (one for the sensors and one for the microcontroller). The results in Table 7.3 are divided into sensing, reasoning and communication. It shows the average energy including its standard deviation for one execution and the average execution time including the standard deviation per task. Lastly, the number of samples is specified.

Task	E_{mean}	E_{std_dev}	t_{mean}	t_{std_dev}	n
Sensing	15.1 mJ	4.4 µJ	569.4 ms	63 µs	114
Reasoning	1.2 µJ	<0.1 µJ	334.3 µs	2.9 µs	114
Communication	47.8 µJ	<0.1 µJ	15.2 ms	31 µs	147

Table 7.3: Mean energy and execution time including standard deviation of the different tasks, the agent performed.

7.3 Multi-Agent Application

In the following section, the multi-agent framework will be evaluated with the same application implemented with two agents. The procedure in performing the experiment is the same, as in the single agent experiment.

7.3.1 Experimental Set-Up

Two DPP3e's are used to form a 2 agent system. The specs of this platform are introduced in Section 3.2.1. The BLE radios in the application domain are not used anymore for the communication. Instead the LWB protocol in the communication domain of the DPP3e is used. The demo application is divided into two agents. One located indoor and one outdoor, each equipped with a light sensor. There is another DPP board, which listens to the LWB protocol. This board is connected over UART to the Raspberry Pi, forwarding all packets. Together, they act as gateway to forward commands to the lights and blinds. The application domain of the outdoor agent is also connected over UART to the Raspberry Pi (it prints the outdoor light conditions) and together with the LWB packets get logged to a text file for further analysis. The application

domain of the indoor agent is connected to a computer, which logs the indoor light conditions. In Figure 7.3, the set-up for the multi-agent experiment is shown.

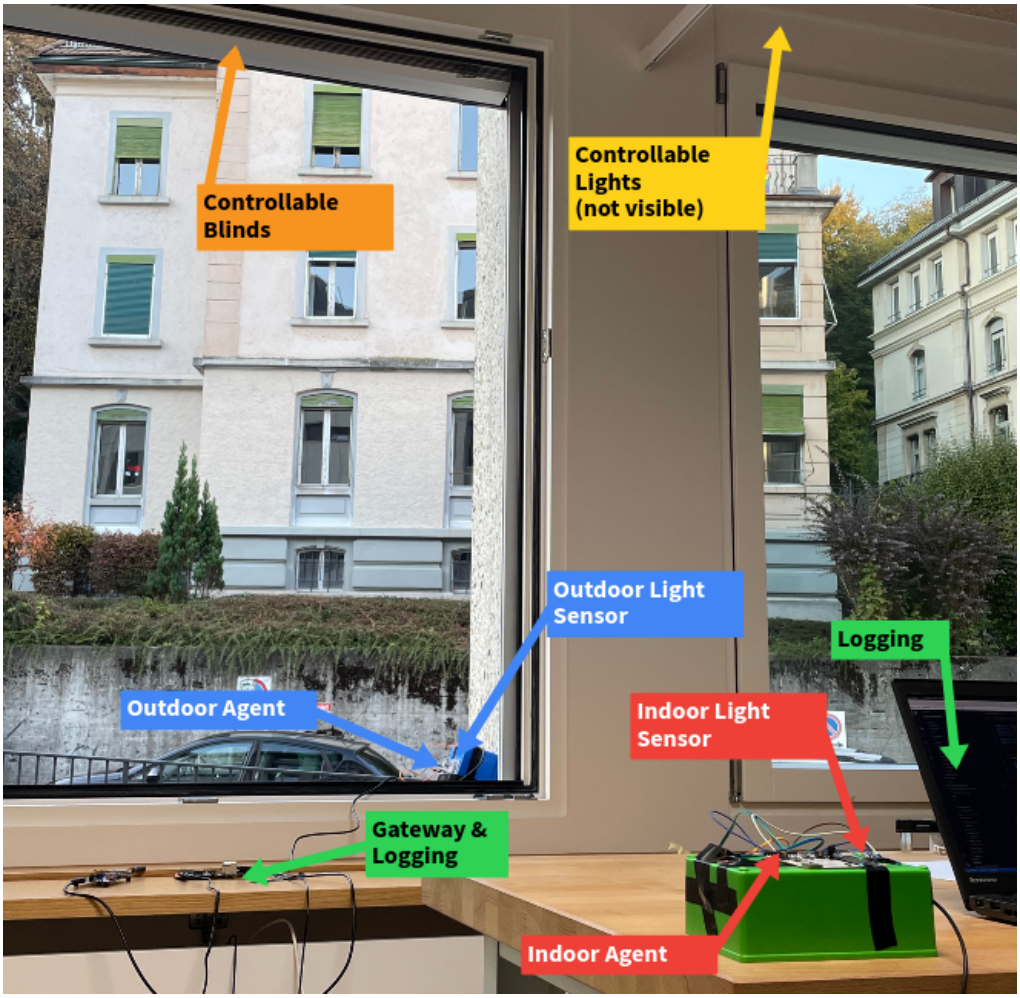

Figure 7.3: The outdoor sensor was placed at the same spot as in the single agent experiment. It is directly connected with a DPP3e running the outdoor agent. The DPP3e is connected to a logging device. The indoor agent with the sensor is mounted on a table facing upwards. It is also connected to a logging device. The gateway to receive action commands and act on the lights/blinds consists of a LWB radio connected via serial to a Raspberry Pi.

The two agent programs can be found in Listing 7 and Listing 8. The indoor agent has the same program as the agent in the previous experiment, but has no update functions for the `{sunny,cloudy,night}_outside` beliefs. They are communicated through broadcast actions by the outdoor agent. The user preference is selected by a General Purpose Input/Output (GPIO) pin of the

microcontroller running the indoor agent.

This experiment is conducted two times in the evening with decreasing outdoor illuminance. For the first execution of the experiment, we set the following light thresholds:

- sunny/cloudy threshold: 4000 lux
- cloudy/night threshold: 500 lux
- indoor threshold: 18.89 lux

For the second execution, we used these thresholds:

- sunny/cloudy threshold: 2100 lux
- cloudy/night threshold: 1800 lux
- indoor threshold: 5 lux

The decision cycle period was set to 15 s, which aligned with the communication period. These were however not synchronized (worst-case, it could be, that communication happens right before the decision cycle, leading to almost 15 s delay, until an action is sent).

```
1  +user_turn_on <- !!preserve_light .
2  +!preserve_light: bright_inside & user_turn_on <- !!preserve_light .
3  +!preserve_light: dark_inside & user_turn_on <- !brighten ; !!preserve_light .
4  -!preserve_light: user_turn_on <- !!preserve_light .
5  +sunny_outside: user_turn_on <- +eco_mode_available .
6  +cloudy_outside: user_turn_on <- -eco_mode_available ; +standard_mode_available .
7  +night_outside: user_turn_on <- -eco_mode_available ; +standard_mode_available .
8  +!brighten: eco_mode_available <- turn_off_lights ; raise_blinds;  -eco_mode_available .
9  +!brighten: standard_mode_available <- turn_on_lights ; -standard_mode_available .
10
11 +user_turn_off <- !!preserve_dark .
12 +!preserve_dark: night_outside & user_turn_off <- lower_blinds ; !!preserve_dark .
13 +!preserve_dark: bright_inside & user_turn_off  <- turn_off_lights; lower_blinds ; !!preserve_dark .
14 -!preserve_dark: user_turn_off <- !!preserve_dark .
```

Listing 7: The indoor agent program is exactly the same, as in the single agent set-up. The difference is, that the `{sunny,cloudy,night}_outside` beliefs are updated by another agent in the wireless network.

7.3.2 Results

First Execution

The experiment (Figure 7.4) lasted for 765 s. The initial conditions were, that the lights were off and the blinds down and the user preference was to brighten

```
!communicate.

+!communicate: sunny_outside
        <- .broadcast(tell, sunny_outside) ;
        .broadcast(untell, cloudy_outside) ;
        .broadcast(untell, night_outside) ;
        !!communicate .
+!communicate: cloudy_outside
        <- .broadcast(untell, sunny_outside) ;
        .broadcast(tell, cloudy_outside) ;
        .broadcast(untell, night_outside) ;
        !!communicate .
+!communicate: night_outside
        <- .broadcast(untell, sunny_outside) ;
        .broadcast(untell, cloudy_outside) ;
        .broadcast(tell, night_outside) ;
        !!communicate .
```

Listing 8: The outdoor agent senses the illuminance levels and based on the thresholds, broadcasts the corresponding beliefs. As the outdoor agent may start before the indoor agent, it is necessary to broadcast the beliefs regularly through maintenance goals.

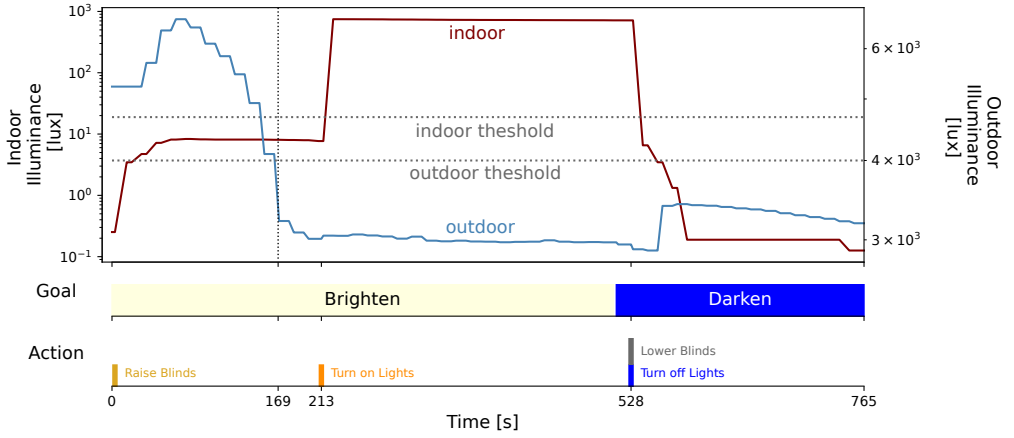

Figure 7.4: The first experiment exposed the latency from acquiring an outdoor belief until a corresponding action was issued.

the room. This initially triggered the action of raising the blinds. The indoor illuminance is actually below the threshold, but the sunny condition prevents the agent to turn on the light. After 169 s, the outdoor conditions indicate, that it is cloudy. Since the outdoor agent has a near worst-case alignment of the decision cycle and communication cycle, it needs 14 s, until the beliefs are sent over the LWB. The indoor agent requires two full decision cycles to come to the

conclusion to turn on the lights, meaning, the breaking condition got triggered to our disadvantage. This results in a total delay of 44 s from acquiring the belief, until the action gets triggered. There were two cycles required, because after acquiring the belief `cloudy_outside`, `+!brighten` was already processed, before `standard_mode_available` got added to the belief base (so there was no context matching the belief base for a plan for `+!brighten`). Therefore, the action gets triggered only in the next decision cycle. At 513 s, the user preference changed to darken. As `+user_turn_off` executes `!!preserve_dark`, the breaking condition gets triggered before a plan for `+!preserve_dark` can be executed. This results in a delay of one decision cycle, until the actions get triggered, resulting in a 15 s delay.

Second Execution

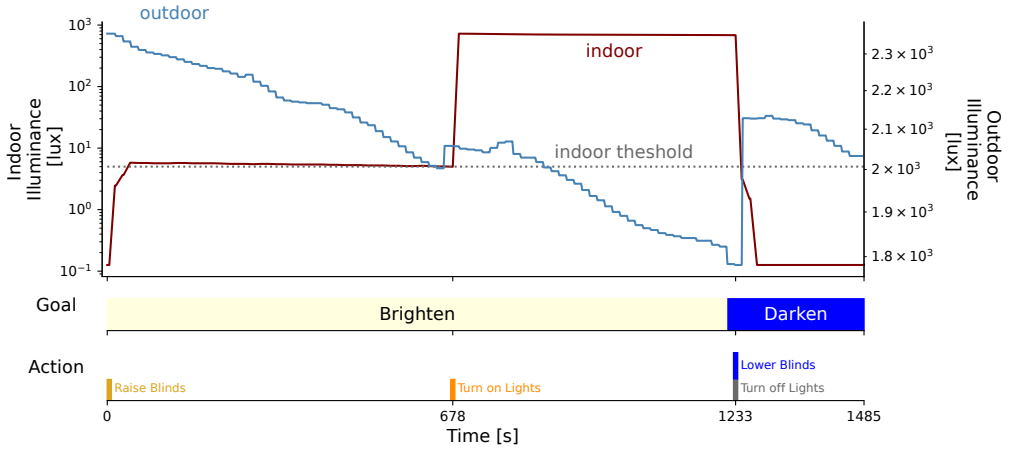

Figure 7.5: In the second experiment, the proactive and reactive behaviour of the agent was revealed. The system reacted correctly based on outdoor and indoor conditions.

The second execution of the experiment (Figure 7.5) lasted for 1485 s. The initial conditions were the same as above (lowered blinds and turned-off lights). As the outside conditions suggested, that it is sunny, the blinds gets raised, as the user preference is to brighten the room. This raised the indoor illuminance right above the threshold leading to the agent believe in `bright_inside`. At 675 s, the indoor illuminance fell below the threshold. 3 s later, the agent performed the action to turn on the lights. This small delay was from the offset of the sensing- and reasoning-task to allow different measurement times (changing illuminances need multiple iterations to adapt gain and integration time for the AS7262 sensor). At 1218 s, the user preference changed to darken the room, exposing the same agent behaviour than in the first execution of the experiment.

Low-Level Metrics

This subsection looks at the low-level metrics of the indoor agent. As the agent program is the same as in the single agent application, the translation yielded the same C++ code size. In Table 7.4 and Table 7.5, the composition of the binary is summarized. To compare it with the single agent, the ROMEM section correspond to the flash section and the combination of the latter three ones correspond to the sram section.

Memory region	Used Size	Region Size	%age Used
ROMEM:	209'988 B	2'000 KB	10.25%
RWMEM:	89'980 B	524'287 B	17.16%
TCM:	0 B	64 KB	0.00%
STACKMEM:	12'288 B	192 KB	6.25%

Table 7.4: The indoor agent occupies a smaller portion of the flash and ram, because it does not include the BLE stack compared to the single agent. This means, there is plenty of space for more complex agents. For the outdoor agent, the numbers look mostly the same, as both of them have a small AgentSpeak program.

.text	.data	.bss	total
209'220 B	768 B	101'500 B	311'488 B

Table 7.5: This table shows the subdivision of the different memory sections of the indoor agent.

The measurement of the energy consumption was performed by an Otii Arc [36] and are summarized in Table 7.6. It is to note, that the reasoning part corresponds to a full decision cycle including message fetching over Bolt. The energy and execution time values thus are significantly higher. The task that consumed the most energy, is sensing. This is also significantly higher, than in the single agent experiment, since longer integration times are chosen. Another significant energy consumer is the communication, which now requires almost three orders of magnitude more energy, although they cannot be directly compared , as here, the communication is bidirectional.

7.4 Analysis

The experiments showed, that it is possible to implement a low-power BDI agent on an embedded device, which can perform a basic home automation task. The

Task	E_{mean}	E_{std_dev}	t_{mean}	t_{std_dev}	n
Sensing	39.7 mJ	160.7 µJ	1113.4 ms	4.5 ms	190
Reasoning	54.8 µJ	17.5 µJ	1.609 ms	490.2 µs	115
Communication	25.9 mJ	4.0 mJ	419.9 ms	64.6 ms	115

Table 7.6: The agents in the MAS use a lot more energy for reasoning and communication. This is due to reasoning now consisting of a full decision cycle (may contain more than one reasoning cycle) and the communication over Bolt (fetching new messages and in case of a transmission, putting messages onto Bolt). Communication also requires a lot more energy because of the more sophisticated wireless infrastructure.

agents show reactive and proactive behaviour, where it reacts on user input and proactively adapts to the environment. Compared to the traditional tasks of sensing and communication, reasoning takes up very little time and energy, so the cost of adding intelligence and autonomy to a simple system is very minimal. The two setups also showed the differences of the single and multi-agent system. The single agent had practically no communication delay, whereas there was a worst-case delay of 15 s in the multi-agent system. There is however the advantage of the meshed communication in LWB, which BLE does not allow efficiently (BLE is point-to-point, which is not directly compatible with MAS). The computational delay could however be greatly improved, because of the introduction of the *full decision cycle* (see Section 5.3). This allows to do a "burst" of reasoning cycles directly one after the other and thus directly decide on an action.

Chapter 8

Conclusion

Current low-power systems gain more and more computational power and can perform increasingly complex tasks. But currently, they lack higher level intelligence and the ability to act autonomously. These behaviours are found in traditional MAS, where different agents can collaboratively work towards a goal. In this work, I demonstrated, that we can integrate important features of autonomous agents from the MAS community into low-power systems. This opens up new fields, because we can have a lot more autonomy in these highly constraint devices, which can be deployed everywhere, as AgentSpeak is far better suited to specify autonomous behaviour than the C programming language. I have shown in a specific application scenario, that a single agent system with a simplified wireless communication infrastructure, having a low latency distributing actions to actuator, that BDI agents work with good performance on low-power embedded systems. The framework is extended to allow inter-agent communication on a dedicated low-power wireless communication system, allowing to build more sophisticated applications with complicated interactions. The communication delay is however increased, but for systems, that need fast decision making, there exists other low-power, asynchronous communication means.

8.1 Future Work

It is clear, that the multi-agent demo application can at best be regarded as a *multiple agent system*, because the true advantage of multi-agent systems is not exposed (jointly achieve goals, competition etc.). The framework however is implemented in such a way that allows easy feature additions. The addition of further illocutionary forces allows then, to implement more complex multi-agent applications.

The breaking-condition developed for the MAS framework needs further refinement for proper functionality in any embedded agent, as there still exists some circumstances, where it gets triggered too soon and thus delays decision making into the next decision cycle.

To benefit from both the powerful traditional MAS and the low-power embedded MAS, it is desirable to develop a communication interface, that allows agents from both world understand each other.

Bibliography

[1] Angelo Garofalo, Gianmarco Ottavi, Francesco Conti, Geethan Karunaratne, Irem Boybat, Luca Benini, and Davide Rossi. A heterogeneous in-memory computing cluster for flexible end-to-end inference of real-world deep neural networks. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2022.

[2] Ali Nikoukar, Saleem Raza, Angelina Poole, Mesut Güneş, and Behnam Dezfouli. Low-power wireless for the internet of things: Standards and applications. *IEEE Access*, 6:67893–67926, 2018.

[3] Long Jin, Steven L Zhang, Sixing Xu, Hengyu Guo, Weiqing Yang, and Zhong Lin Wang. Free-fixed rotational triboelectric nanogenerator for self-powered real-time wheel monitoring. *Advanced Materials Technologies*, 6 (3):2000918, 2021.

[4] Nurettin Sezer and Muammer Koç. A comprehensive review on the state-of-the-art of piezoelectric energy harvesting. *Nano Energy*, 80:105567, 2021.

[5] Tao Yang, Shengxi Zhou, Shitong Fang, Weiyang Qin, and Daniel J Inman. Nonlinear vibration energy harvesting and vibration suppression technologies: Designs, analysis, and applications. *Applied Physics Reviews*, 8(3): 031317, 2021.

[6] Silvia Liberata Ullo and Ganesh Ram Sinha. Advances in smart environment monitoring systems using iot and sensors. *Sensors*, 20(11):3113, 2020.

[7] Olivier Boissier, Rafael H. Bordini, Jomi Hübner, and Alessandro Ricci. *Multi-agent oriented programming: programming multi-agent systems using JaCaMo*. MIT Press, 2020.

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

[9] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing multi-agent systems with JADE*. John Wiley & Sons, 2007.

[10] Reydson Schuenck Barros, Victor Hugo Heringer, Carlos Eduardo Pantoja, Nilson Mori Lazarin, and Leonardo Machado de Moraes. An agent-oriented ground vehicle's automation using jason framework. In *ICAART (2)*, pages 261–266, 2014.

[11] Nikolaos Papakostas, Anthony Newell, and Abraham George. An agent-based decision support platform for additive manufacturing applications. *Applied Sciences*, 10(14):4953, 2020.

[12] Omar Bahri, Asmaa Mourhir, and Elpiniki I Papageorgiou. Integrating fuzzy cognitive maps and multi-agent systems for sustainable agriculture. *Euro-Mediterranean Journal for Environmental Integration*, 5(1):1–10, 2020.

[13] Davide Rossi, Francesco Conti, Manuel Eggiman, Alfio Di Mauro, Giuseppe Tagliavini, Stefan Mach, Marco Guermandi, Antonio Pullini, Igor Loi, Jie Chen, et al. Vega: A ten-core soc for iot endnodes with dnn acceleration and cognitive wake-up from mram-based state-retentive sleep mode. *IEEE Journal of Solid-State Circuits*, 57(1):127–139, 2021.

[14] Haoyu Ren, Darko Anicic, and Thomas A Runkler. Tinyol: Tinyml with online-learning on microcontrollers. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2021.

[15] Carles Gomez, Joaquim Oller, and Josep Paradells. Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors*, 12(9):11734–11753, 2012.

[16] Seyed Mahdi Darroudi, Raül Caldera-Sànchez, and Carles Gomez. Bluetooth mesh energy consumption: A model. *Sensors*, 19(5):1238, 2019.

[17] Felix Sutton, Bernhard Buchli, Jan Beutel, and Lothar Thiele. Zippy: On-demand network flooding. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 45–58, 2015.

[18] Federico Ferrari, Marco Zimmerling, Luca Mottola, and Lothar Thiele. Low-power wireless bus. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, pages 1–14, 2012.

[19] SRR Dhiwaakar Purusothaman, Ramesh Rajesh, Karan K Bajaj, and Vineeth Vijayaraghavan. Implementation of arduino-based multi-agent system for rural indian microgrids. In *2013 IEEE Innovative Smart Grid Technologies-Asia (ISGT Asia)*, pages 1–5. IEEE, 2013.

[20] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Agilla: A mobile agent middleware for self-adaptive wireless sensor networks. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(3):1–26, 2009.

[21] Samuel Bucheli, Daniel Kroening, Ruben Martins, and Ashutosh Natraj. From AgentSpeak to C for safety considerations in unmanned aerial vehicles. In *Conference Towards Autonomous Robotic Systems*, pages 69–81. Springer, 2015.

[22] S Russel and P Norvig. *Chapter 2*, page 34–63. Prentice Hall, 3rd edition, 1999.

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

[24] 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.

[25] Luca Rufer, Naomi Stricker, Reto Da Forno, Lothar Thiele, and Andres Gomez. Demo abstract: DPP3e: A harvesting-based dual processor platform for advanced indoor environmental sensing. In *2022 21st ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 495–496. IEEE, 2022.

[26] Jan Beutel, Roman Trüb, Reto Da Forno, Markus Wegmann, Tonio Gsell, Romain Jacob, Michael Keller, Felix Sutton, and Lothar Thiele. The dual processor platform architecture: Demo abstract. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*, pages 335–336, 2019.

[27] *Apollo3 Blue Plus MCU Datasheet*. Ambiq Micro, Inc., Austin, US-TX, February 2022. Doc. Revision: 1.1.2.

[28] Federico Ferrari, Marco Zimmerling, Lothar Thiele, and Olga Saukh. Efficient network flooding and time synchronization with glossy. In *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 73–84. IEEE, 2011.

[29] Felix Sutton, Marco Zimmerling, Reto Da Forno, Roman Lim, Tonio Gsell, Georgia Giannopoulou, Federico Ferrari, Jan Beutel, and Lothar Thiele. Bolt: A stateful processor interconnect. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 267–280, 2015.

[30] Matuzalém Muller dos Santos. Programação orientada a agentes BDI em sistemas embarcados. Master's thesis, Universidade Federal de Santa Catarina, 2022.

[31] Market leading RTOS (real time operating system) for embedded systems with internet of things extensions, Oct 2022. URL `https://www.freertos.org/`. Accessed: 2022-10-31.

[32] Apollo3 Blue, Oct 2022. URL `https://ambiq.com/apollo3-blue/`. Accessed: 2022-10-31.

[33] *High Accuracy Ambient Light Sensor With I²C Interface*. Vishay Intertechnology, Inc., Malvern, US-PA, April 2022. Rev. 1.6, 28-Apr-2022.

[34] *AS7262 Datasheet*. Ams-Osram AG, Austria, March 2017. v1-01.

[35] Lukas Sigrist, Andres Gomez, Roman Lim, Stefan Lippuner, Matthias Leubin, and Lothar Thiele. Rocketlogger: Mobile power logger for prototyping iot devices: Demo abstract. In *Proceedings SenSys Conference*. ACM, 2016.

[36] *Otii Arc - Product Specification*. Qoitech AB, Sweden, 2022.