Anytime BDI

A time-bounded agent architecture

Márcio Fernando Stabile Junior

Thesis presented to the
Institute of Mathematics and Statistics
of the University of São Paulo
in partial fulfillment
of the requirements
for the degree of
Doctor of Science

Program: Ciência da Computação
Advisor: Prof. Dr. Jaime Simão Sichman

This work was supported by CNPq, Brazil, Grant 140448/2016-0.

São Paulo
December, 2022

Anytime BDI

A time-bounded agent architecture

Márcio Fernando Stabile Junior

This version of the thesis includes the
corrections and modifications suggested
by the Examining Committee during the
defense of the original version of the work,
which took place on December 2, 2022.

A copy of the original version is available
at the Institute of Mathematics and
Statistics of the University of São Paulo.

Examining Committee:

Prof. Dr. Jaime Simão Sichman – IME-USP
Prof. Dr. Rafael Heitor Bordini – PUC
Prof. Dr. Jomi Fred Hübner – UFSC
Prof. Dr. Gustavo Enrique de Almeida P.Alves Batista – UNSW
Profª. Drª. Anarosa Alves Franco Brandão – EP-USP

Ficha catalográfica elaborada com dados inseridos pelo(a) autor(a)
Biblioteca Carlos Benjamin de Lyra
Instituto de Matemática e Estatística
Universidade de São Paulo

Stabile Junior, Márcio Fernando
Anytime BDI: a time-bounded agent architecture / Márcio Fernando Stabile Junior; orientador, Jaime Simão Sichman. - São Paulo, 2022.
134 p.: il.

Tese (Doutorado) - Programa de Pós-Graduação em Ciência da Computação / Instituto de Matemática e Estatística / Universidade de São Paulo.
Bibliografia
Versão corrigida

1. AGENTES INTELIGENTES. I. Simão Sichman, Jaime. II. Título.

Bibliotecárias do Serviço de Informação e Biblioteca
Carlos Benjamin de Lyra do IME-USP, responsáveis pela
estrutura de catalogação da publicação de acordo com a AACR2:
Maria Lúcia Ribeiro CRB-8/2766; Stela do Nascimento Madruga CRB 8/7534.

Acknowledgments

To Professor Jaime Simão Sichman for his guidance and trust and for all the time he dedicated to this work.

To CNPq for enabling and funding this research.

To all the teachers who put their trust in me, believing that I could go further.

To all my family members who encouraged me to keep learning more.

To my wife Barbara, who always believed in me, even when I didn't.

Lastly and most importantly, to God for making all of this possible.

Resumo

Márcio Fernando Stabile Junior. **Anytime BDI:** ***Uma arquitetura de agentes limitada no tempo***. Tese (Doutorado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Quando se integram agentes BDI a ambientes onde o tempo de resposta do agente interfere na qualidade de suas ações, fica aparente o problema da ausência de controle sobre o tempo de processamento do agente. Não havendo alguma forma de realizar esse controle, não há garantias de que o agente irá conseguir deliberar sobre as informações percebidas e executar uma ação no ambiente dentro de um limite de tempo esperado. Com o objetivo de prover esse tipo de controle sobre o tempo de processamento de agentes BDI, este trabalho apresenta um modelo de agente BDI chamado Anytime BDI. Esse modelo utiliza algoritmos anytime, técnicas de profiling e técnicas de otimização multiobjetivo para garantir que o agente execute ações no ambiente dentro de um limite de tempo pré estabelecido, minimizando a perda de qualidade das ações decorrente desse controle. Através da implementação deste modelo na linguagem Jason e de validações estatísticas apropriadas, mostramos que existem cenários onde conseguimos aumentar a qualidade do agente e cenários onde conseguimos reduzir o tempo de processamento do agente sem prejuízo na resposta do mesmo.

Palavras-chave: agentes autônomos. agentes BDI. algoritmos anytime. profiling de agentes.

Abstract

Márcio Fernando Stabile Junior. **Anytime BDI: *A time-bounded agent architecture***. Thesis (Doctorate). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

When integrating BDI agents into environments where the agent's response time interferes with the quality of their actions, the problem of lack of control over the agent's processing time becomes apparent. As there is no way to perform this control, there is no guarantee that the agent will be able to deliberate on the perceived information and perform an action in the environment within an expected time-bound. In order to provide this type of control over the processing time of BDI agents, this work presents a BDI agent model called Anytime BDI. This model uses anytime algorithms, profiling techniques, and multiobjective optimization techniques to ensure that the agent executes actions in the environment within a pre-established time-bound, minimizing the loss of quality of actions resulting from this control. Through the implementation of this model in Jason language and appropriate statistical validations, we show that there are scenarios where we can increase the agent's quality and scenarios where we can reduce the agent's processing time without prejudice to the agent's response.

Keywords: autonomous agents. BDI agents. anytime algorithms. agent profiling.

List of abbreviations

IME	Instituto de Matemática e Estatística
USP	Universidade de São Paulo
BDI	Belief Desire Intention
MAPC	Multi-agent programming contest
MCDM	Multiple Criteria Decision Making

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Motivation

The agent-based modeling paradigm originates from the concept of components that interact autonomously, maintain control over their state, and can perform tasks. When agents can incorporate reasoning and learning mechanisms, they become intelligent agents. Among the most known agent models, there is the BDI model (Belief, Desire, Intention) proposed by Bratman (1987) and formalized by Rao and Georgeff (1991). Ziafati and Dastani, 2013 reasons that the authors designed this model to implement autonomous systems with deliberative behaviors inspired by the model of practical human reasoning.

Due to this capability, Tweedale *et al.*, 2007 states that the BDI agent model is the right paradigm to be chosen when using agents in complex systems because it combines several desirable attributes. Such as:

a) The authors based it on a respected philosophical theory - Bratman, 1987's theory of human reasoning;

b) It has been implemented several times in several languages;

c) Many complex applications use it;

d) The BDI theory has been rigorously formalized.

On the other hand, according to Kostiadis and Hu, 2000, conducting competitions between intelligent agents such as RoboCup[1] and MAPC[2] is an attempt to foster AI and research in intelligent robotics, providing a standard problem where a wide range of technologies can be integrated and examined. Some fields include multi-agent collaboration, strategy acquisition, real-time planning and reasoning, strategic decision making, and machine learning. The use of BDI agents in this type of competition allows the development of behaviors closer to human reasoning, increasing the realism in representing these behaviors.

[1] RoboCup: http://www.robocup.org/

[2] MAPC: https://www.multiagentcontest.org/

In addition to simulation systems, there is a growing interest in the use of embedded BDI agents, either in UAVs (Santos *et al.*, 2015), or in land vehicles (Pantoja *et al.*, 2016) and robots (Ziafati and Dastani, 2013). Whether in simulation systems or embedded in robots, the BDI model is a fundamental tool for developing intelligent agents with a high level of reasoning.

One obstacle that prevents the perfect use of BDI agents in these environments is that BDI architectures traditionally do not have control over their execution time. So, when an agent starts to process its actions, there is no guarantee whether it will finish its processing and perform an action in a bounded time. When integrating BDI agents with simulators, the agent response time can negatively influence their performance. This influence is even more concerning when we look at discrete event simulators. The main characteristic of discrete event simulators is that they execute in steps and the simulator constantly pauses its processing. During the intervals between the steps, the agents receive information from the simulator containing their percepts, and have to decide on their actions and communicate them back to the simulator until the beginning of the next step. As a result, the time taken by the agent to find the best action is often longer than the time-bound set by the simulator, which leads to the agent's loss of action and inertia throughout the step. In addition, the more complex the agents become, the longer the time taken to decide an action, further aggravating the problem. This problem occurs similarly in embedded BDI agents. For example, an agent embedded in a UAV can spend all of its battery time processing its actions and crash before being able to execute them. In contrast, an agent embedded in a land vehicle can collide with other objects if it does not dodge in time.

A commonly applied methodology to alleviate the problem is the use of hybrid agent architectures. These architectures combine reactive and cognitive capabilities to enable a shorter response time if there is a need for the agent to act quickly. The most famous of these architectures is the "Architecture of Three Layers" with models described by Gat, 1998 and Ferguson, 1992. In this type of architecture, there are three different processing layers. Usually, the bottom layer (or execution layer) is reactive to allow for a quick response. In contrast, the upper layer is cognitive to allow a better representation of the world and a higher level of reasoning. The problem with this type of architecture, according to Ziafati and Dastani, 2013 is first that the redundancy of having to keep the same information in different models puts an additional burden on maintaining the system and can lead to inconsistencies. Second, diagnosing plan failures can be difficult, as the deliberative component may not have relevant information about the causes of failure of an action taken by another component. Third, the execution of the plan may be less efficient because the execution layer does not have a global view of the plan. An agent with a single deliberative layer would be free of these problems.

In order to employ BDI architectures without the risks presented, it is then necessary to do some form of control on the agent's response time. This control would ensure that the reasoning time of the agent does not exceed some pre-established bound. This solution would allow for better integration of these agents into simulation systems and embedded environments such as those described above, reducing the risk of using BDI agents. This reduction could encourage using BDI architectures and provide greater realism to the behaviors presented.

1.2 Objectives

This research aims to propose a model and an implementation of a BDI agent architecture capable of controlling its reasoning execution time to execute within a pre-established time-bound.

In particular, we want to be able to answer three research questions:

- **Q1:** Given a specific time response upper bound, is it possible to guarantee that a BDI agent can often enough process perceptions, deliberate on them, and determine the action it wants to perform within the time limit while simultaneously guaranteeing a minimum quality of actions?
- **Q2:** How to define in advance the minimum time necessary for the agent to produce a response with the desired quality?
- **Q3:** What is the impact of processing time on response quality?

The first step to achieving these objectives is to formalize an agent execution model with controllable time. Since we want to set an upper bound on execution time and a minimum limit on response quality, indiscriminately reducing processing time can cause a significant drop in response quality. Thus, a more sophisticated control of the reasoning cycle is necessary to answer the main question.

Second, to evaluate the proposed architecture and answer the secondary issues, it will be necessary to implement the architecture accompanied by the implementation of agents based on this architecture. Finally, the analysis of the agents' response times, jointly with the analysis of the responses, should make it possible to identify the minimum response times and the interference of the response time control in the quality of the generated actions.

1.3 Methodology

As previously described, this work presents three research questions. These questions are related to the ability to control or adapt the runtime of a BDI agent. The main question (Q1) contains the core of the problem to be solved. The secondary questions (Q2 and Q3) describe examinations of the built architecture to identify its limitations and validate if the proposed architecture solves the proposed problem. Thus, questions Q2 and Q3 directly depend on the solution of question Q1.

To answer Q1, we propose some activities that include:

1. Analysis of the literature on anytime algorithms, parallel algorithms, and other techniques that may influence the execution time of the BDI architecture;
2. Formalization of a BDI architecture whose maximum execution time is controllable;
3. Implementation of the formalized architecture;
4. Creation of test scenarios to analyze the quality of actions performed by agents;
5. Dissemination of results through one or more publications.

To answer Q2 and Q3, we propose:

1. Creation or modification of test scenarios and definition of action quality metrics;
2. Use of data analysis techniques experiments, as in Jain, 1991 to analyze the impact of processing time on the quality of responses and possible impacts caused by other factors.
3. Comparison of agents implemented in the proposed architecture with agents implemented in other BDI architectures to evaluate the difference between the execution times and the quality of the actions performed;

1.4 Contributions

The main contributions of this work are:

- To bring results from anytime algorithms to the BDI model, allowing control of the execution time of BDI agents. Thus, improving the performance in scenarios where the agent's processing time impacts the result of its actions.
- To propose a model of BDI agent that controls its execution time, formalizing the investigated concepts in a model that is sufficiently generic to be used in any application of BDI agents.
- To define an architecture that describes the functioning of this model, suggesting possible implementation methods.
- To implement the proposed model in an existing BDI agent programming language (Jason) to assess whether the proposed architecture achieves its objectives through appropriate statistical methods.

Finally, we provide the implementation of the architecture at https://github.com/mfstabile/AnytimeJason.

1.5 Document structure

In the following chapters, we describe the details of the proposed BDI model and its implementation. In Chapter 2, we introduce the concepts of agent-oriented programming, including the BDI model and agent-oriented programming languages. In Chapter 3, we present concepts related to anytime algorithms, such as performance profiles and the compilation process of anytime algorithms. Chapter 5 contains the details of the systematic review carried out at the beginning of the project, looking for works that would help us solve the investigated problem. We present in Chapter 6 the formal description of the proposed model and in Chapter 7 the implementation of this model in Jason language. Then, Chapter 8 contains related works and details on how our proposal differs from existing works. Next, in Chapter 9, we describe the methodology of analysis of experiments that guided the evaluations carried out. After that, Chapter 10 contains a set of experiments in a simple scenario created to allow an in-depth evaluation of the proposed model. Similarly, Chapter 11 contains a set of experiments carried out in the MAPC simulator, analyzing

more complex agents. Finally, Chapter 12 presents the conclusions and the next steps of the research.

Part I

Background

Chapter 2

Agent Oriented Programming

According to Wooldridge, 2009, Shoham, 1993 proposed a new paradigm for programming, which was called Agent-Oriented Programming (AOP), considered as a specialization of Object Oriented Programming (OOP). In this paradigm, developers can design intelligent agents using mental states, such as beliefs, desires, and intentions. The reason for this proposal was the use of these concepts by human beings as an abstraction mechanism to represent the properties of complex systems in the same way they are used to explain human behavior.

According to Shoham, 1993, while OOP proposes to consider a computer system as a set of modules that communicate with each other, AOP adds mental states to each module, now calling them agents that contain belief and decision-making mechanisms. These agents can then exchange information, make requests and offers, compete and help each other. The 2.1 table summarizes the relationship between AOP and OOP.

	OOP	AOP
Basic unit	object	agent
Parameters defining state of basic unit	unconstrained	beliefs, commitments,, capabilities, choices, ...
Process of computation	message passing and response methods	message passing and response methods
Types of message	unconstrained	inform, request, offer, promise, decline, ...
Constraints on methods	none	honesty, consistency, ...

Table 2.1: *Comparative table between OOP and AOP presented by Shoham, 1993*

2.1 BDI Model

According to Wooldridge, 1997, one of the most successful agent theories is the belief-desire-intention (BDI) model of Rao and Georgeff, 1991. The beliefs represent

the agent's information about the world, such as its perceptions. The desires are tasks assigned to the agent. An agent may not be able to achieve all its desires, and as in human beings, its desires may even be inconsistent, in the sense that achieving one of them may make it impossible to achieve the others. As agents are not generally able to achieve all of their desires (even consistent ones), they must therefore choose some subset of these desires and commit to achieving them. The agent's intentions represent the desires it has committed to achieving. These intentions will then be part of future decision-making where, for example, an agent must not adopt intentions that conflict with those to which it is committed.

For the BDI model, Wooldridge, 2000, p. 26 proposes a basic agent design. We present the more formalized version of this design in Algorithm 1. In this basic design, the agent is supposed to observe the world (line 4) and update its internal model based on this observation (line 5). It should then deliberate about what intentions to achieve next given its current world model (line 6), use means-ends reasoning to get a plan for the intentions (line 7), and finally, execute the generated plan (line 8).

Algorithm 1 A basic control loop from Wooldridge, 2000, p. 31.

1: $B := B_0$; /* B_0 *are the initial beliefs* */
2: $I := I_0$; /* I_0 *are the initial intentions* */
3: *while true do*
4: *get next percept* ρ;
5: $B := brf(B, \rho)$;
6: $I := deliberate(B)$;
7: $\pi := plan(B, I)$;
8: $execute(\pi)$;
9: *end – while*

The model present in Algorithm 1 is, according to the author, a basic model and does not allow some desirable behaviors to the agent, such as commitment strategies and intention reconsideration. To make the agents present these behaviors, the author constructs a more sophisticated version of the same model adding these new capabilities. We present this version in Algorithm 2.

The first change happens in the deliberation process. The author suggests that in the deliberation process, an agent typically analyzes its options, chooses some, and finally commits to some. Given this behavior, he suggests separating the *deliberate* function into two. One is for option generation, where the agent generates a set of possible alternatives, and another is called filtering, where the agent chooses among the generated alternatives and commits to achieving them. Thus, two functions replace the *deliberate* function in the model. These are the *options* and *filter* functions, present in lines 6 and 7 respectively.

The second change concerns the agent's commitment to its plans. In the basic model of Algorithm 1, once the agent chooses a plan to execute, it will perform all steps of that plan. This behavior can bring many issues. As we discussed earlier, while the agent performs his actions, it is possible that some of them fail or that the environment changes in a way that they are impossible to perform. To address this problem, Wooldridge, 2000 adds mechanisms in the Algorithm 2 model for reactivity and dropping intentions.

Algorithm 2 A more sophisticated control loop from Wooldridge, 2000, p. 37.

```
1:  B := B_0; /* B_0 are the initial beliefs */
2:  I := I_0; /* I_0 are the initial intentions */
3:  while true do
4:     get next percept ρ;
5:     B := brf(B, ρ);
6:     D := options(B, I);
7:     I := filter(B, D, I);
8:     π := plan(B, I);
9:     while not (empty(π) or succeeded(I, B) or impossible(I, B)) do
10:       α := head(π);
11:       execute(α);
12:       π := tail(π);
13:       get next percept ρ;
14:       B := brf(B, ρ);
15:       if reconsider(I, B) then
16:          D := options(B, I);
17:          I := filter(B, D, I);
18:       end − if
19:       if not sound(π, I, B) then
20:          π := plan(B, I);
21:       end − if
22:    end − while
23: end − while
```

For this, he defined a loop, present in lines 9 to 22 that modifies the *execute* function. Instead of executing the entire plan at one time, line 10 selects only the first action to be performed, which is performed on line 11 and removed from the plan in line 12. This loop allows performing other verifications between the execution of two actions. One of these verifications is to analyze whether the plan failed. After acting, the agent again perceives the environment on lines 13 and 14, and on line 19, the agent checks if the plan failed through the *sound* function. If the agent believes the plan failed, it will try to find a new plan to reach its goal (line 20). Another verification is to analyze intentions. The functions *succeeded* and *impossible* are added on line 9. If the agent realizes its goal is complete while executing the plan, it does not have to carry on execution. Likewise, if it believes the goal is impossible to achieve, it is not rational to continue executing the plan. These two functions then make these verifications respectively and terminate the plan execution in these cases.

According to the author, the basic agent never stops to reconsider whether its intentions are still appropriate. Once the agent decides to reach a goal, it will only stop when it reaches the objective. This type of agent is said to have a blind commitment. This type of commitment may not be the most appropriate in some situations. Thus, the third and final change intends to add the possibility of reconsidering its intentions to the agent. The author defines the *reconsider* function in line 15 to make this possible. This function has the purpose of deciding whether the agent should stop to reconsider its actions (which can

be a costly process) or not. If it judges that the agent should reconsider, the *options* and *filter* functions on lines 16 and 17 are performed to reevaluate the agent's intentions.

Similarly, the *sound* function verifies if the plan can still succeed. If not, the *plan* function generates a new plan. According to how this structure was defined, the agent only generates a new plan if the current plan fails. Hence, If the environment changes during the execution of a plan, and such a change makes a second plan much more efficient, that second plan will not be analyzed if the first one is sound.

2.2 3APL

3APL (An Abstract Agent Programming Language) is a programming language for implementing cognitive agents designed by Hindriks *et al.*, 1999 apud Bordini, Braubach, *et al.*, 2006. In it, agents have beliefs, goals, and plans as mental attitudes. Agents can generate and review their plans to achieve goals and interact with each other and the environment they share with other agents.

One of the main features of 3APL is the implementation of an agent's mental attitudes and the deliberation process that manipulates them. In particular, 3APL allows the direct specification of mental attitudes, such as beliefs, goals, plans, actions, and rules of reasoning.

The 3APL programming language design respects software engineering and programming principles. Such principles include separation of concerns, modularity, abstractions, and reusability. It also allows integration with the Prolog (declarative) and Java (Object Oriented) programming languages.

An example of an agent developed in 3APL was presented by Dastani *et al.*, 2003 and found in the code 3. In this example, a robot aims to transport boxes to a specific room while minimizing the cost of transport. In the first and second actions (lines 3 to 5 and 7 to 9) the robot can go from the room $R1$ to room $R2$ if it is already in room $R1$ ($pos(R1)$). After executing the move, if the agent is prohibited from making that move, its cost is 5. Otherwise, its cost is only 1. The third and fourth actions (lines 11 to 13 and 15 to 17) represent the transition when the agent picks up or places a box on the floor. The belief base (lines 19 to 22) shows the initial state of the agent, containing its initial position, the position of the boxes, and which movements are prohibited. This agent has only one objective (line 24): transport boxes. To achieve it, the agent uses its rule base (lines 26 to 28) to plan its movements.

2.3 Jadex

Jadex is a software framework for creating goal-oriented agents that follow the belief-desire-intention (BDI) model that was developed by Braubach *et al.*, 2003 apud Bordini, Braubach, *et al.*, 2006. The framework is composed of a rational agent that sits in a layer on top of a middleware agent infrastructure like JADE by Bellifemine *et al.*, 1999 and allows agent development with well-established technologies such as Java and XML.

Jadex has been used to build applications in different domains like simulation, task

Algorithm 3 3APL agent example from Dastani *et al.*, 2003

```
1:  PROGRAM }}transport robot 1ε
2:  CAPABILITIES :
3:        pos(R1), cost(X), not forbid(R1, R2)
4:              Go(R1, R2)
5:        not pos(R1), pos(R2), cost(X + 1),
6:
7:        pos(R1), cost(X), forbid(R1, R2)
8:              Go(R1, R2)
9:        not pos(R1), pos(R2), cost(X + 5),
10:
11:       box(self), delpos(R), pos(R)
12:             PutBox()
13:       not Box(self), box(R),
14:
15:       pos(R), box(R)
16:             GetBox()
17:       not box(R), box(self)
18: BELIEFBASE :
19:       pos(r4), box(r2), delpos(r1), gain(5), cost(0), forbid(r4, r2),
20:       door(r1, r2), door(r1, r3), door(r2, r4), door(r3, r4),
21:       door(R1, R2) : −(R2, R1),
22:       forbid(R1, R2) : −forbid(R2, R1)
23: GOALBASE :
24:       transportBox()
25: RULEBASE :
26:       transportBox() ← pos(R1), box(R2), delpos(R3)|goxy(R1, R2); GetBox(); goxy(R2, R3); PutBox(),
27:       goxy(R1, R2) ← pos(R1), door(R1, R3), notR1 = R2|Go(R1, R3); goxy(R3, R2),
28:       goxy(R1, R2) ← R1 = R2|SKIP.
```

planning, and mobile computing. For example, Jadex was used to develop a multi-agent application for negotiating treatment schedules in hospitals.

2.4 AgentSpeak

The AgentSpeak language is an abstract agent-oriented programming language based on a constrained first-order language with events and actions. It was created by Rao, 1996 to allow programming of the behavior of BDI agents and multi-agent systems.

Rao, 1996 defines that the language consists of a set of base beliefs (or facts, in the context of logic programming) and a set of plans. Plans are context-sensitive, event-invoked recipes that allow the hierarchical decomposition of objectives and the execution of actions. The plans aim to achieve the agent's goals. Thus, the plans selected for execution would be the agent's intentions.

However, the AgentSpeak language is abstract, and its authors did not build a compiler

or interpreter for the language, thus holding no practical use.

2.5 Jason

Jason is an open-source interpreter implemented in the Java language for an extended version of the AgentSpeak language, developed by Bordini, Hübner, and Wooldridge (2007), which implements the operational semantics of the language and provides a platform for the development of multi-agent systems.

According to Bordini, Braubach, *et al.*, 2006, besides those defined by the AgentSpeak language, some of the features available in Jason are:

1. Communication between agents based on speech acts (and annotations on beliefs about information sources);
2. Annotations on plan labels, which more elaborate selection functions can use;
3. Selection functions, trust functions, and general agent architecture (perception, belief revision, communication between agents and actions) fully customizable (in Java);
4. Extendability (and use of legacy code) through user-defined "internal actions";
5. A clear notion of environment for multi-agent scenarios, implemented in Java.

Reasoning cycle

The Jason framework executes the agents through a reasoning cycle divided into ten main steps, as seen in Figure 2.1. During each reasoning cycle, the agent receives the percepts from the environment and the messages from other agents, deliberates, chooses an intention, and performs an action.

The internal process is described by Bordini, Hübner, *et al.*, 2007 as follows:

1. **Perceiving the Environment**
   The first step in executing the agent is to perceive the environment. Perceptions are a set of literals where each literal represents a property in the current state of the environment.
2. **Updating the Belief Base**
   The belief base is updated to reflect changes in the environment by a customizable *update* function. The default Jason's implementation assumes that everything the agent can perceive will be on the percepts set sent to the agent by the environment. So the standard belief update function goes through the belief base and percepts set to include the new literals, remove the literals that are no longer perceived, and keep the ones that were already known. Each literal inclusion or exclusion generates an event and adds it to an event list.
3. **Receiving Communication from Other Agents**
   At this stage, the interpreter checks if the agent has received any messages from another agent. If so, one message is selected to be processed.

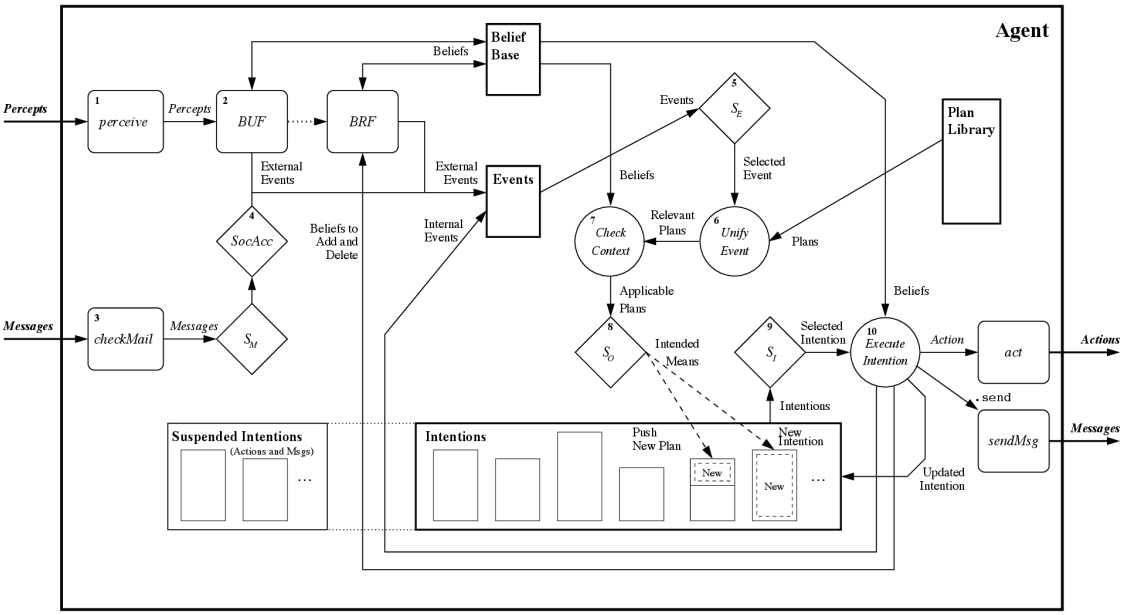


Figure 2.1: *Reasoning cycle of a Jason agent presented by* Bordini, Hübner, et al., 2007.

4. **Selecting “Socially Acceptable" Messages**
   Before being processed, messages go through a check to determine whether or not they should be accepted by the agent, verifying the sender and the content through a customizable function.

5. **Selecting an Event**
   From the list of events created in the belief update phase, the event selection function chooses an event for execution. The default selection function chooses the oldest event in the list.

6. **Retrieving all Relevant Plans**
   Once an event is selected, it is necessary to find a plan that allows the agent to act to deal with the event. The first step is to go through the list of plans and find all those invoked by the selected event.

7. **Determining the Applicable Plans**
   With the list of relevant plans, it is necessary to verify which of them can be executed, that is, whose preconditions are valid.

8. **Selecting One Applicable Plan**
   At this point, the architecture has determined all possible plans to handle the selected event. So, theoretically, any plan can be used for this purpose. The architecture then uses another customizable selection function to decide which plan to execute and adds the selected one to the existing list of intentions.

9. **Selecting an Intention for Further Execution**
   During execution, an agent typically has more than one intention to execute, each representing a different focus of attention. However, the architecture can carry out only one intention at a time. Thus, it is necessary to choose a single intention.

10. **Executing One Step of an Intention**
    In this last stage, the agent has updated its information about the environment, dealt with one of the generated events, and must now perform an action. From the chosen intention, the first action not yet performed is selected and executed.

Jason's compiler performs this process through nine Semantic Rules [1]. They are Java methods that implement the following behaviors:

- **buf**
  It receives a list of new environmental perceptions and updates the belief base by adding new beliefs, removing beliefs that are no longer perceived, and generating the respective events.
- **ProcMsg** If there are messages, select one, check the sender and content and add the events resulting from the message.
- **SelEv** Selects one event from the list and stores it in a variable to be analyzed by the *FindOp* function. Without customization, this function always selects the oldest event not yet analyzed.
- **FindOp** This function identifies the plans relevant to the event chosen by the *SelEv* function (which have the same trigger) and analyzes them looking for an applicable plan. The first applicable plan found is selected and stored.
- **AddIM** Creates a new intention by merging the chosen event in *SelEv* and the chosen plan in *FindOp* and adds that intention to the agent's intention set.
- **ProcAct** Checks for existence and analyzes so-called Feedback Actions. These objects contain the result of the execution of an action in the environment performed in some previous reasoning cycle (if it was successful or unsuccessful). After, in case of successful execution, the intention can continue to execute. An unsuccessful execution generates a failure event and removes the intention.
- **SelInt** Selects one of the agent's intentions and separate it for execution by removing it from the agent's intention list. Jason's default behavior defines the intention chosen as the one that is not chosen the longest, as in a round-robin mechanism.
- **ExecInt** Identifies the next step of the plan selected in the *SelInt* function and executes it. The corresponding behavior is carried out for each type of command, whether adding or removing a belief or objective, performing rule tests, or other internal actions. One difference is concerning external actions. Instead of immediately acting, the *ExecInt* function creates an ActionExec type object that stores the intention and the action to be performed.
- **ClrInt** Inspections if the plan has not finished and if the intention the plan seeks to achieve is still active. If both conditions are valid, the rule returns the plan to the set so it can be selected again.

We present the execution flow of these functions in Figure 2.2. A Jason agent's reasoning cycle begins with the *Sense* function, which calls the *buf* function and then the *ProcMsg*

[1] Although the buf function is not a semantic rule, we describe it together with semantic rules for simplicity.

function. Then, the Deliberate function calls the *SelEv* function. Then, the execution proceeds to the Act function if there are no events to handle. If there is, the *FindOp* functions are executed, and then the *AddIM* function before passing the execution to the Act function.

Similarly, the *Act* function calls the *ProcAct* function and then the *SelInt* function. The reasoning cycle ends if there are no intentions/plans to execute. If there are intentions, *ExecInt* and *ClrInt* execute. After executing the *ClrInt* function, the *Act* function checks for the existence of an *ActionExec* type object that the *ExecInt* function may have created. If it exists, it performs this action. Thus the reasoning cycle of a Jason agent is composed of the sequential execution of the functions described above.

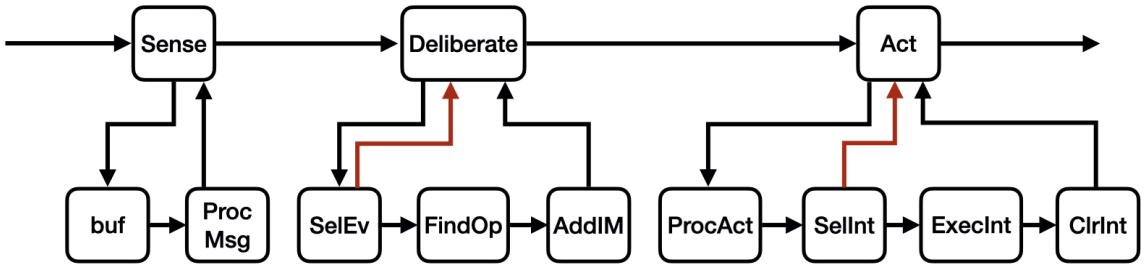


Figure 2.2: *Jason Semantic Rules execution flow.*

Code example

We present an example of an agent built in Jason in code 4. In this example, an agent is responsible for cleaning a room, provided with Jason[2] source code. The agent performs this task by going through the grid's squares, and when it finds garbage in one of the squares, it carries this garbage to the incinerator. The agent will perceive the environment at each step and obtain information about its current position "$pos(r1, 0, 0)$", the position of the incinerator "$pos(r2, 10, 10)$" and whether there is garbage in the space where "$garbage(r1)$" is. At the beginning of the execution, the agent has the objective of traversing the grid in search of garbage (line 7). The agent's plan checks for garbage in the same space as the agent (lines 11 to 14). If empty, the agent moves to the next space. If the agent notices that there is garbage in the space it occupies (line 16), it takes the garbage (line 17), carries it to the incinerator (lines 19 to 24) and returns to the same position where it previously was (lines 25 to 27).

Since the Jason interpreter is implemented in the Java language, it is possible to use methods developed in the Java language for tasks where using a BDI abstraction would make its execution inefficient. Such tasks include mathematical calculations and file readings.

Using these mechanisms, Jason can run the agents so that they interact with the environment when necessary and perform the specified tasks.

[2] http://jason.sourceforge.net/wp/

Algorithm 4 Jason agent code example

```
/* Initial beliefs */

at(P) : -pos(P, X, Y) & pos(r1, X, Y).

/* Initial goal */

!check(slots).

/* Plans */

+!check(slots) : notgarbage(r1)
   <- next(slot);
        !!check(slots).
        +!check(slots).

+garbage(r1) : not .desire(carry_to(r2))
   <- !carry_to(r2).

+!carry_to(R)
   <-//remember where to go back
        ?pos(r1, X, Y);
        -+pos(last, X, Y);
        //carry garbage to r2
        !take(garb, R);
        //goes back and continue to check
        !at(last);
        !!check(slots).

+!take(S, L) : true
   <- !ensure_pick(S);
        !at(L);
        drop(S).

+!ensure_pick(S) : garbage(r1)
   <- pick(garb);
        !ensure_pick(S).
        +!ensure_pick(_).

+!at(L) : at(L).
+!at(L) <- ?pos(L, X, Y);
                move_towards(X, Y);
                !at(L).
```

Chapter 3

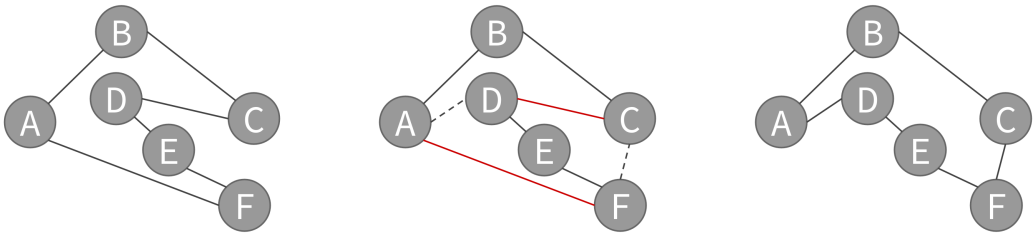
Anytime Algorithhms

The term anytime algorithm emerged in research conducted by Dean and Boddy, 1988. They studied the problem of planning actions in scenarios where the time available to generate a plan is variable, and the decision process for formulating the plans is complex. Dean and Boddy, 1988 named anytime algorithms those algorithms that had two characteristics: *a)* The algorithm can be terminated at any time and will return some response; *b)* The returned answers improve in some well-behaved way as a function of time.

Building on the work of Dean and Boddy, 1988, Zilberstein, 1993 proposed using this class of algorithms for planning and all kinds of bounded rationality. With this, he proposed ways to analyze and compose different *anytime* algorithms to create complex systems formed by multiple *anytime* algorithms that maintain the same property of allowing the balance between processing time and quality of results. With that, he examined the problem of real-time decision-making by intelligent agents. This examination resulted in an efficient bounded optimization model based on *anytime* computation, offline compilation, and runtime monitoring.

To illustrate how anytime algorithms work, we present an example of an anytime algorithm that seeks to solve the classic traveling salesperson problem based on the example presented by Zilberstein, 1993. In this problem, the salesperson must go through all the cities and return to the city of origin. When initializing this algorithm, a random path between the cities would be generated, like the one in Figure 3.1a. That way, it would be possible to generate an answer rather quickly. Thus, if the algorithm terminates, it can already provide an answer to the problem, even if it is not the best one. If there is still processing time, the algorithm will randomly select two edges from the path and replace them with two new ones that are not in the current path, as illustrated in Figure 3.1b. If the new path is worse than the existing one, it is discarded. If it is better, the new path is stored, as in Figure 3.1c. If the algorithm is interrupted at this point, it will provide an answer better than the previous one. Thus, the algorithm will look for better answers as long as there is processing time.

There are two categories of anytime algorithms: interruptible algorithms and contract algorithms. Interruptible algorithms can be interrupted at any point in their execution and will produce results with the expected quality. Contrary to interruptible algorithms,



(a) *Random path* **(b)** *Random change* **(c)** *New best path*

Figure 3.1: *Traveling salesman anytime solution.*

contract algorithms require the time allocation to be known in advance to provide outputs whose quality varies with the time allocation. A contract algorithm may produce useless results if it terminates at any moment before the contract period. However, despite this disadvantage, it is much easier to build contract algorithms than interruptible algorithms. Thus, it is necessary to consider the advantages and disadvantages of each of these categories when building an anytime algorithm.

3.1 Performance profiles

The main characteristic of anytime algorithms is the ability to exchange processing time for the quality of results. So, it is desirable to know the expected quality of the result for a given available execution time and what is the minimum necessary time to reach the desired minimum quality. We can then describe an anytime algorithm's performance profiles to achieve these goals. Based on the activation of an anytime algorithm with a given time allocation, these performance profiles provide a range of values for the quality of the algorithm's result.

There are three main methods for calculating performance profiles. The first is through a structural analysis of the algorithm. In many iterative algorithms, we can calculate the result's error as a function of the number of iterations performed by the algorithm. So, we can calculate its performance by knowing the time taken to execute each iteration. The problem, however, is when the iterations of the algorithm do not all take the same amount of time to execute or when the increase in quality cannot be well described. In these cases, we can calculate the performance profile through a series of simulations. These simulations use representative domain cases and analyze statistics from different executions. Finally, a third method combines the simulation technique with a learning technique. In this method, we create an approximate performance profile using the simulation method. Then, as the system runs in the environment, it updates the performance profile according to its experience.

Figure 3.2 shows a possible quality map for an anytime algorithm. Each point on the plot corresponds to one execution. This map is a base to generate the equivalent performance profile, shown in Figure 3.3.

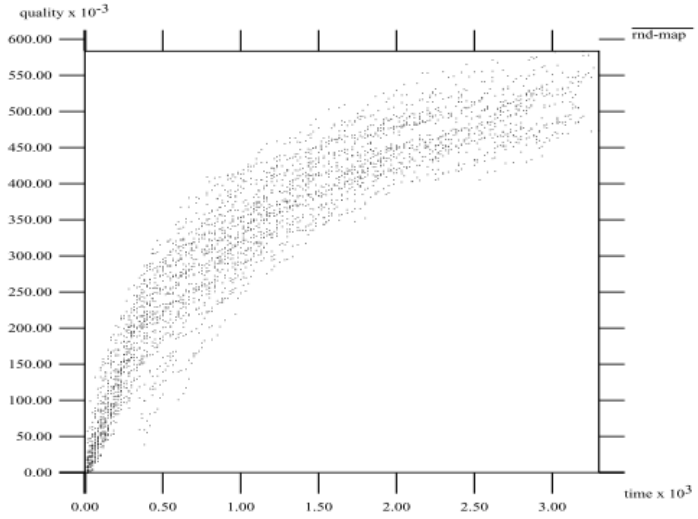


Figure 3.2: *Quality map of an anytime algorithm. (From Zilberstein, 1993)*

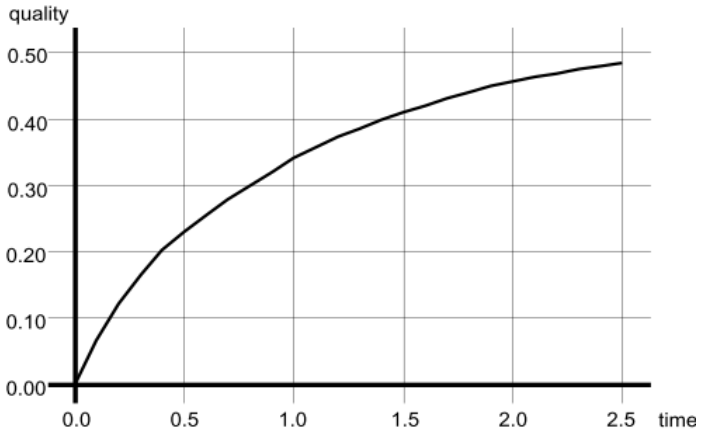


Figure 3.3: *Performance profile of an anytime algorithm. (From Zilberstein, 1993)*

3.2 Compilation

Zilberstein, 1993 describes a process for creating complex systems by putting together several so-called elementary anytime algorithms. Elementary anytime algorithms are those not formed by combining other anytime algorithms. Zilberstein, 1993 calls this creation process **compilation of anytime algorithms**.

Figure 3.4 illustrates a compilation example. The compiler receives a module with some elementary anytime algorithms and their respective performance profiles as input. The compiler must then parse this input and produce an anytime executable module consisting of a compiled version of the original module, a predefined runtime monitor, and the system performance profile, which may include some auxiliary time allocation information. The compiler analyzes the performance profiles of each elementary anytime algorithm. Then, it calculates for each possible execution time what time allocation for each elementary anytime algorithm will maximize the quality of the final result. As an example, in Figure 3.4, it is possible to verify from the performance profiles that the $AA2$ and $AA4$ algorithms present higher quality results in less time. Thus, the compiler may allocate more time to

algorithms $AA1$ and $AA3$ to compensate for the delay in obtaining improvements in the quality of the result. If the performance profiles of the elementary algorithms change (as in the learning method), the compilation must happen again.

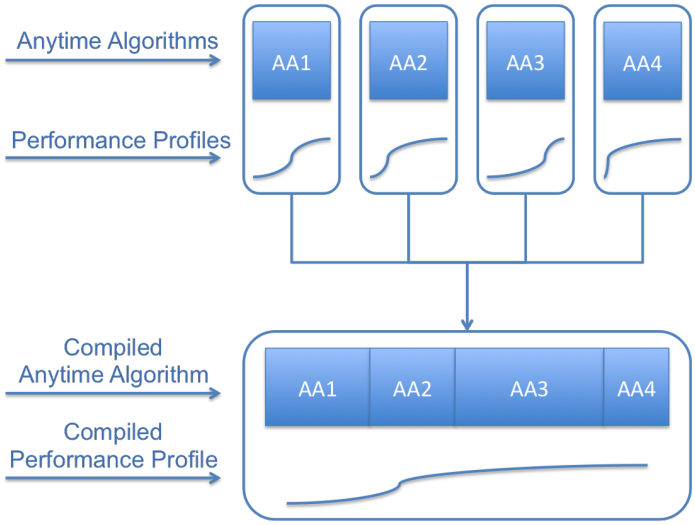


Figure 3.4: *Anytime algorithm compilation example.*

3.3 Programming environment

Beyond performance and compilation profiles of anytime algorithms, in Zilberstein, 1993 the authors describe how to execute anytime algorithms on a standard computer. For this execution, the programming language used to implement the algorithms, the operating system that executes the algorithm, and the scheduler of processes of the operating system must show the following characteristics:

1. The programming language must support:
   (a) Functions as first class objects.
   (b) Functions can take optional and keyword arguments.
   (c) Execution is deterministic over time. That is, every deterministic function's run-time is consistent across activations using the same input.
2. The operating system must support the following operations:
   (a) A program can create processes and control their execution.
   (b) The scheduling of processes is based on priorities. At each point of time the running process is the one with the highest priority among all the ready processes.
   (c) The system maintains a real-time clock.
   (d) A process can sleep until a certain event occurs. The process becomes ready immediately after the event occurs.
   (e) Events can be triggered by any process or by the real-time clock.

3. The scheduler of processes of the operating system must behave as follows:

   (a) The scheduler must be event-driven. The process with the greatest priority continues to run if no event happens. One ready process is chosen at random for execution if two ready processes have the same priority.

   (b) When a process is active and another with a higher priority becomes available, the later instantly becomes active.

   (c) The scheduler's overhead must have very little impact on the performance profiles of the active algorithms.

Analyzing the restrictions, although possible, the execution of truly anytime algorithms depends on the choice of an operating system and a programming language capable of meeting all these restrictions. Conventional operating systems are incapable of meeting these requirements because the scheduling of processes is not only based on priorities, for example. Many programming languages use garbage collection mechanisms, making execution time not deterministic. Programming languages and operating systems specially designed for real-time applications are necessary for true anytime execution.

Chapter 4

Multiobjective optimization

According to Miettinen (2008), many planning and decision-making tasks entail various competing priorities that need to be taken into account concurrently. These tasks are frequently referred to as multiple criteria decision making (MCDM) problems. Depending on the specific situation at hand, there are numerous ways that MCDM problems might be categorized.

Scenarios where there is a set of unknown possible solutions, and that can be described by constraint functions are called multiobjective optimization problems. No single solution to a multiobjective optimization problem exists, but a number of mathematically equivalent solutions can be found. These answers are referred to as Pareto optimal solutions.

The idea of addressing a multiobjective optimization problem is typically viewed as assisting a human decision maker (DM) in taking into account numerous objectives at once and in locating a Pareto optimal solution that appeals to him/her the most adequate. As a result, the DM must participate in the solution process by providing preference information, and in one way or another, his or her choices will affect the ultimate solution. That is, preference data is used to create a more or less explicit preference model, which is then used to find solutions that better suit the DM's preferences.

Two methods are so widely used that they have become known as basic methods. In the following sections we explain their functionality.

4.1 Weighting Method

Miettinen (2008) describes the weighting method as solving the problem:

$$\begin{aligned} &\text{minimize} \quad \sum_{i=1}^{k} w_i f_i(x) \\ &\text{subject to} \quad x \in S \end{aligned}$$

where $wi \geq 0$ for all $i = 1, ..., k$, S and, typically, $\sum_{i=1}^{k} w_i = 1$.

In this method, the result of each of the k functions f_i on the parameters x is weighted by a weight w_i. The values of x that minimize the sum are the possible solutions to the problem.

The weighting method can be employed as an a posteriori method, in which case the DM is asked to choose the best Pareto optimal option after various weights are applied to produce various Pareto optimal solutions. As an alternative, the method can be utilized as an a priori method by asking the DM to provide the weights.

However, the weighting approach has a significant flaw. It has been demonstrated that all Pareto optimal solutions can only be discovered by adjusting the weights in convex problems. As a result, it is possible that no matter how the weights are chosen, some Pareto optimal solutions to nonconvex problems cannot be discovered. This is due to the possibility that when changing the weights, the technique may leap from one vertex to another, hence failing to discover intermediate solutions.

4.2 ϵ-constraint method

In the ϵ-constraint method, according to Miettinen (2008), one of the objective functions is selected to be optimized, and the others are converted into constraints. Therefore, the problem gets the following form:

$$\begin{aligned} &\text{minimize} && f_l(x) \\ &\text{subject to} && f_j(x) \leq \epsilon_j \text{ for all } j = 1, ..., k, j \neq l, \\ &&& x \in S, \end{aligned}$$

where $l \in \{1, ..., k\}$ and ϵ_j are upper bounds for the objectives ($j \neq l$).

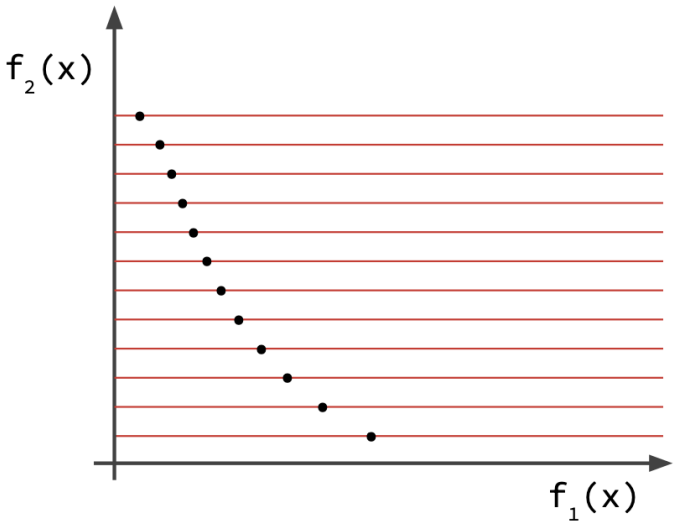


Figure 4.1: *ϵ-constraint method example.*

We present in Figure 4.1 an example with two functions. In this example, function

f_1 is optimized and function f_2 is converted into a constraint. The red lines symbolize possible constraints for f_2 and the point on the line is the point that minimizes f_1 within the imposed constraint. By varying the constraints, we can find all the Pareto optimal solutions.

Finding any Pareto optimal solution does not require convexity, which is a positive factor when compared with the weighting method. Consequently, both convex and nonconvex problems can be solved using this approach.

Chapter 5

Systematic literature review

In 2018, we conducted a systematic literature review to answer the proposed research questions and identify existing works in related areas. After identifying such works, it became possible to verify the techniques developed to solve similar problems and verify the aspects not yet solved. In addition, it was also possible to validate whether the research questions were meaningful.

5.1 Protocol

Since questions 2 and 3 are directly dependent on the answer to the first question, the focus of the review was to answer the first research question:

- **Q1**: Given a specific time response upper bound, is it possible to guarantee that a BDI agent can often enough process perceptions, deliberate on them, and determine the action it wants to perform within the time limit while simultaneously guaranteeing a minimum quality of actions?

In order to try to find answers to this question, the review focused on the following topics:

- Use of BDI agents in simulators and embedded in robots;
- Analysis of performance and techniques to reduce the execution time of BDI agents;
- Runtime controlled BDI agents.

5.2 Data sources

We performed searches in the following knowledge bases:

- ACM Digital Library;
- Engineering Village;
- IEEE Xplore;

- Science Direct
- Springer
- Web of Science

We used four search strings in each base according to the syntax accepted by each one.

For the “ACM Digital Library” base:

- **String 1**: +(bdi) +(robot robotic simulator simulated)
- **String 2**: +(bdi) +(parallel performance)
- **String 3**: +(bdi) +(anytime "real-time" "time limited" "time bounded")
- **String 4**: +(agent bdi "autonomous system") +(anytime "real-time" "time limited" "time bounded") +(robot robotic simulator simulated) +(parallel performance)

For the “Engineering Village”, “IEEE Xplore”, “Science Direct”, “ Springer” and “Web of Science” databases:

- **String 1**: (bdi) AND (robot OR robotic OR simulator OR simulated)
- **String 2**: (bdi) AND (parallel OR performance)
- **String 3**: (bdi) AND (anytime OR real-time OR "time limited" OR "time bounded")
- **String 4**: (agent OR bdi OR "autonomous system") AND (anytime OR real-time OR "time limited" OR "time bounded") AND (robot OR robotic OR simulator OR simulated) AND (parallel OR performance)

5.3 Search results

We present the results of applying the search strings in the knowledge bases in Table 5.1.

Base	SB1	SB2	SB3	SB4
ACM Digital Library	60	46	15	147
Engineering Village	532	400	104	1744
IEEE Xplore	68	65	19	302
Science Direct	539	840	5868	17884
Springer	4868	10928	2660	41397
Web of Science	145	769	73	360

Table 5.1: *Result of search strings in each knowledge base*

In some of the knowledge bases, searches returned thousands of results. After some analysis, we found that some databases show within the search results works that do not present all the searched terms. For example, a search for “(bdi) AND (parallel OR performance)” displayed results where the term BDI was absent. In other cases, the acronym BDI matched terms from other areas, such as psychology.

In order to solve this sort of problem, we added some inclusion criteria to help us identify the relevant research. The titles of the 300 most relevant works of each search string in each knowledge base were analyzed (when more than 300 results were present) according to the following criteria:

- Works whose area is computing;
- Works that help to understand the BDI paradigm;
- Jobs dealing with limiting, controlling, or analyzing an agent's runtime;
- Jobs that address the time aspect when linking a BDI agent to a simulator or robot.

The first step was listing all titles and removing any duplicated works. Afterward, we evaluated each paper's abstracts to verify if they fit the inclusion criteria. We present the number of papers in each step in Table 5.2.

Filter	**SB1**	**SB2**	**SB3**	**SB4**
Total results	6212	13048	8739	61834
Results analyzed	1405	1411	811	1709
Selected titles	61	42	39	23
Selected abstracts	45	26	24	11

Table 5.2: *Articles analyzed*

Following, we separated the selected articles in each survey by year. This separation is presented in Table 5.3.

5.4 Analysis

Among the analyzed articles, those that presented the greatest correlation with this research were the works of Kostiadis and Hu, 2000, Zatelli *et al.*, 2016 and multiple papers by Zhang and Huang, including Zhang and Huang, 2005 and Zhang and Huang, 2007. The content of the articles and how they compare to the model proposed in this work will be presented in Chapter 8.

The evaluation of the selected articles made it possible to identify some interesting information. An analysis of the years of publications shows that about 90% of the works were published in the previous 15 years. This fact suggests that although the BDI paradigm emerged from the work of Rao and Georgeff, 1991, the concern with execution time and the attempt to control or adapt it to be used in applications with real-time characteristics is recent, and there is still interest from researchers in the subject.

Another verified fact is that the problem encountered when using BDI agents in time-bounded simulators also occurs when trying to use BDI agents embedded in robots. Both require the time interval between executing two actions by the agent to be small. In simulators, a high interval will make the agent stop performing its actions. In a robot, although nothing deliberately prevents it from performing its action, it may no longer be possible due to changes in the environment caused by the dynamic characteristic of the real world. This fact indicates that works developed for the robotics area may have

Year	SB1	SB2	SB3	SB4
1994	0	0	0	1
1995	1	1	0	0
1998	0	1	0	1
1999	2	1	1	0
2001	0	1	0	0
2003	1	1	2	0
2004	1	1	1	0
2005	5	2	3	1
2006	2	2	3	0
2007	0	2	0	0
2008	4	0	1	0
2009	0	0	3	0
2010	6	1	6	3
2011	3	2	1	0
2012	3	3	1	2
2013	3	2	1	1
2014	2	0	0	0
2015	1	1	0	1
2016	7	2	1	1
2017	4	2	0	0
2018	0	1	0	0

Table 5.3: *Publications per year*

beneficial results for the research questions of this work. Furthermore, the answers to the research questions could also represent exciting results in the robotics community.

Part II

Proposal

Chapter 6

Anytime BDI Agent

In this chapter, we describe our BDI agent model that allows control of its runtime and was initially introduced in Stabile Jr., 2022. We present both its formal description and possible implementation of this model.

6.1 General View

One of the significant issues of the BDI architecture is the need for deliberation and means-ends reasoning to compute in a small amount of time. The main characteristic of anytime algorithms is the possibility of controlling the execution time by achieving sub-optimal results. Therefore, it seems logical to use the anytime algorithms approach to design more efficient BDI agents.

In the sequence, we consider that perception acquisition and action execution depend on the environment in which the agent is inserted and can hardly be constrained. For example, a BDI agent embedded in a robot can not perform a moving action faster by simply activating its motor wheels faster than its maximum limit. Based on this fact, we assume the perception and execution steps take constant time. So, we focus here on controlling the three central parts of the agent's execution, updating the agent's internal model, deliberating, and means-ends reasoning (lines 5, 6, and 7, from Algorithm 1).

We can then define the functions of updating the internal agent model (belief revision function - brf), deliberation ($deliberate$), and means-ends reasoning ($findPlan$) as contract anytime algorithms. Thus, we can define time values t_{brf}, t_{delib} e t_{plan}, respectively for the *brf*, *deliberation*, and *findPlan*, which will represent the time that each of these functions will execute. Algorithm 5 presents the concept of the `Anytime BDI` control mechanism, which is the idea behind the architecture proposed in Section 6.2.

Assuming the perception acquisition will take a time t_p and an action execution a time t_a to perform, the total time T for an anytime cycle to complete (executing lines 3 to 8) will be $T = t_p + t_{brf} + t_{delib} + t_{plan} + t_a$. By controlling t_{bm}, t_{delib}, and t_{plan}, we have a better control of T. Consequently, we can increase or reduce the response time needed to improve the reasoning response.

Algorithm 5 Simple anytime BDI control mechanism.

1: $B := B_0$; /* B_0 *are the initial beliefs* */
2: $I := I_0$; /* I_0 *are the initial intentions* */
3: *while true do*
4: *get next percept* ρ;
5: $B := brf(B, \rho, t_{brf})$;
6: $I := deliberate(B, t_{delib})$;
7: $\pi := findPlan(B, I, t_{plan})$;
8: $execute(\pi)$;
9: *end − while*

The next question is then how to choose t_{brf}, t_{delib}, and t_{plan}? Since we are focused in the three intermediate processes, we define $t_\Delta = t_{brf} + t_{delib} + t_{plan}$. What we want is to find out which values of t_{brf}, t_{delib}, and t_{plan} maximize the utility value of the agent, given a value of t_Δ. For this, Zilberstein, 1993 proposes using performance profiles and compiling anytime algorithms. According to Zilberstein, 1996, *"A Performance Profile of an anytime algorithm, Q(t), denotes the expected output quality with execution time t"*. Thus, we can perform simulations in the brf, deliberate, and plan algorithms to identify the performance profiles of each one, that is, the expected quality of the result for different time allocations. The compilation process of the anytime algorithm will then define for a range of values in t_Δ, which are the values of t_{brf}, t_{delib}, $and t_{plan}$ that maximize the expected output.

In order to execute anytime algorithms, Zilberstein, 1995 describes the need for a monitoring mechanism. According to the author, *"Without such a mechanism, anytime components of a system are worthless."* The monitoring mechanism is responsible for controlling the execution of anytime algorithms. It controls the moment when they begin to run and the moment when they should stop. According to Zilberstein, 1995, there are two types of monitoring, the passive and the active ones: *"A monitoring scheme is said to be passive if the corresponding time allocation mapping is completely determined prior to the activation of the system. A monitoring scheme is said to be active if it is not passive. The corresponding time allocation mapping is partially determined while the system is active."* Thus, we must add a monitoring mechanism to the model, which will be responsible for the correct activation of the brf, deliberate, and plan algorithms.

When we look at the changes between the Basic Algorithm (Algorithm 1) and the Complete Algorithm (Algorithm 2), we can see that the ability to reconsider the intentions and the current plan arises from the fact that the plan stops being executed entirely and starts to be executed one action at a time. The primary evidence is that there are no new functions with new behaviors. When we analyze the added functions to the Complete Algorithm, we verify that their function is to avoid the execution of the *option*, *filter*, and *plan* functions, which are considered computationally costly. Since we want to be able to control the execution time of the functions, we no longer need to avoid executing them. We can then describe a new anytime control loop with the same capabilities to reconsider plans and intentions without using extra functions. Furthermore, to reduce the agent processing effort, instead of the agent being responsible for planning, which is usually a very computationally expensive task, the agent will consult a pre-built plan library. Finally,

we present this new **`Anytime BDI`** control mechanism in Algorithm 6.

Algorithm 6 Anytime BDI control mechanism.

```
1: B := B_0; /* B_0 are the initial beliefs */
2: I := I_0; /* I_0 are the initial intentions */
3: while true do
4:     get next percept ρ;
5:     B := brf(B, ρ, t_brf);
6:     I := deliberate(B, t_delib);
7:     π := findPlan(B, I, t_plan);
8:     execute(head(π));
9: end − while
```

6.2 Formal Description

In order to achieve the goal of allowing the execution of a BDI agent with control of its execution time, in this section we formalize the model we call Anytime BDI Agent. First, it is necessary to define three terms: Internal actions, external actions, and plans. As defined by Schut *et al.*, 2004, external actions are the ones that affect the agent's environment. Internal actions are the ones that affect the internal state of the agent. We formally define these elements below:

Definition 6.2.1. *External action (α): An external action (α) is an action that the agent performs in the environment.*

Examples of external actions include agent movement, activating a button, and carrying an item.

Definition 6.2.2. *Internal action (β): An internal action (β) is an action that the agent performs and does not directly change the environment. Instead, it affects the internal state of the agent.*

Internal actions include creating a new belief or acquiring a new goal.

Definition 6.2.3. *Plan (π): A plan (π) is a pre-defined sequence of external (α) and internal (β) actions.*

Having made these definitions, we formalize the model of the Anytime BDI Agent:

Definition 6.2.4. *(ABDIA) An Anytime BDI Agent is an intelligent agent architecture composed of two layers. One is the Agent Data Layer, which contains all the agent's data structures. The other is the Agent Control Layer, composed of the mechanisms that control the agent execution. We define an ABDIA as:*

$$ABDIA = \langle ADL, ACtrL \rangle$$

where

- The Agent Definition Layer $ADL = \langle P, B, \Pi, D, I, \alpha_{default}, t_\Delta, DI, PP \rangle$ is the set of structures that store the necessary data for the execution of the agent;
- The Agent Control Layer $ACtrL = \langle BM, IG, IE, M, HP \rangle$ is the set of control structures used to control the agent's execution;

Definition 6.2.5. *(ADL) The Agent Data Layer comprises all the structures that store the necessary data for the agent's execution. We define the ADL as:*

$$ADL = \langle P, B, \Pi, D, I, \alpha_{default}, t_\Delta, DI, PP \rangle$$

where

- P is the agent's set of percepts;
- B is the agent's set of beliefs;
- Π is the agent's set of plans;
- D is the agent's set of desires;
- $I = \{i_1, i_2, ..., i_n | i = \langle d, \pi \rangle \wedge \ d \in D \wedge \pi \in \Pi\}$ is the agent's set of intentions, where each one is a pair formed by a desire ($d \in D$) and a plan ($\pi \in \Pi$);
- $\alpha_{default}$ is the default action executed by the agent if it was unable to find a better action;
- t_Δ is the maximum time interval between the execution of two external actions (α) by the agent;
- $DI \subseteq I$ is a queue of delayed intentions ordered by function HP. Delayed intentions are the intentions whose external actions the agent is waiting to execute;
- PP is the set of performance profiles for the agent components;

Definition 6.2.6. *(ACtrL) The Agent Control Layer comprises the processes responsible for running the Anytime BDI Agent and ensuring its execution within the specified time. We define the ACtrL as:*

$$ACtrL = \langle BM, IG, IE, M, HP \rangle$$

where

- The Belief Manager $BM : (B \times P \longrightarrow \mathcal{B})$ is the anytime component responsible for based on current beliefs (B) and perceptions (P), produce a new set of beliefs (B);
- The Intention Generator $IG : (B \times D \times I \times \Pi \longrightarrow I)$ is the anytime component responsible for based on current beliefs (B), desires (D), and current intentions (I), produce a new set of intentions (I) and their respective plans (Π);
- The Intention Executor $IE : (B \times I \times HP \times DI \longrightarrow \alpha, DI)$ is the anytime component responsible for based on current beliefs (B), intentions (I), priority function (HP) and delayed actions (DI), choose an external action (α) to be executed on the environment and select actions to be executed in the future (DI);

- The Monitor $M : (PP \times t_\Delta \longrightarrow t_{bm}, t_{ig}, t_{ie})$ is the component responsible for based on the performance profiles and t_Δ, start and finish the BM, IG, and IE modules executions in order to execute an external action α by the end of time t_Δ. Each module runs for a time determined by the values of t_{bm}, t_{ig}, and t_{ie} respectively, that are calculated by the Monitor;
- The highest priority function $HP : (\{\alpha_1, \alpha_2, ...\} \longrightarrow \alpha_x)$ returns the external action with highest priority from a set.

This model aims to make the agent perform an external action α in the environment at each time interval t_Δ. For this to happen, three modules were defined, BM, IG, and IE, which are anytime algorithms that are responsible for analyzing perceptions, beliefs, desires, plans, and intentions in order to generate an external action to perform in the environment. In order to ensure that these modules run within the time-bound, the monitor module analyzes the performance profiles of each algorithm involved and executes them for the calculated time. When the time ends, the monitor module receives the generated external action and executes it in the environment. We illustrate this concept in Figure 6.1.

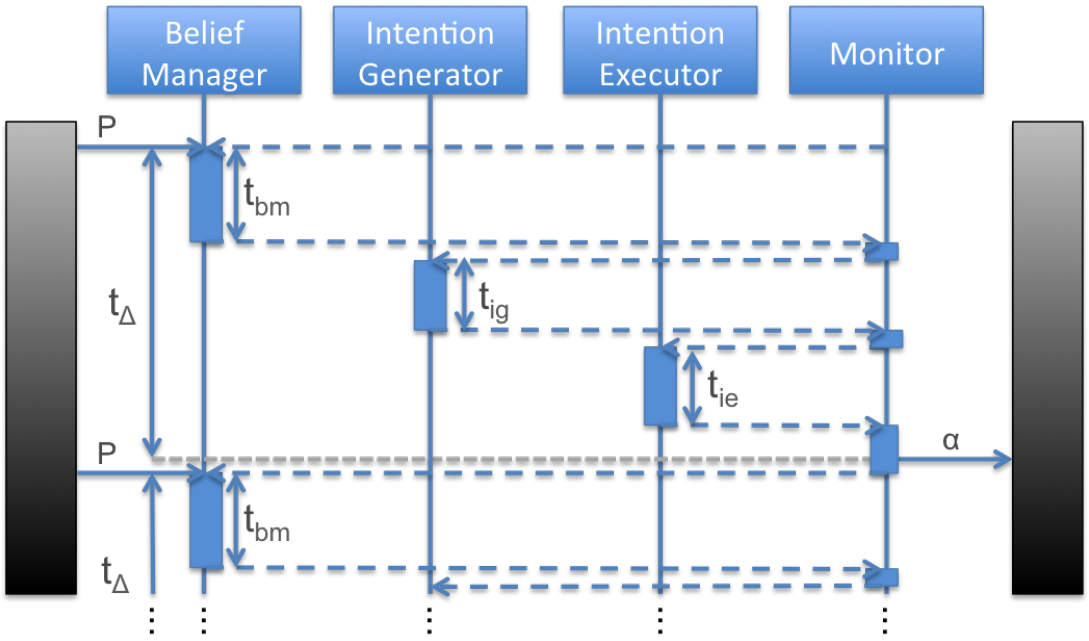


Figure 6.1: *Agent execution.*

As described earlier, runtime control is possible because the *Belief Manager*, *Intention Generator*, and *Intention Executor* components are anytime algorithms. Algorithms 7, 8, and 9, presented in the sequence, describe the operation of these components.

6.3 Belief Manager

The *Belief Manager*, described in Algorithm 7, assumes that there is a list of percepts to evaluate. As long as the processing time set by the Monitor is not over, the component uses the *getNextPercept* function (line 3) to choose the subsequent perception to analyze.

This selection follows a user-defined policy based on the remaining perceptions and time. Next, the *brf* function incorporates this perception into the belief base (line 4). Finally, line 5 removes the perception from the list of pending perceptions. This behavior allows the agent to analyze the most critical perceptions in case there is no time to analyze all of them.

These user-defined policies vary according to the scenario and the agent created. For example, in the *getNextPercept* function, we can have a priority mechanism where the most critical perception of the agent's reasoning is selected first. An example would be an agent who drives a vehicle first to analyze the perception that there is a wall in front of it and only later if there is time, the perception of a tree beside it. Another possibility is the choice of perception based on the time remaining. For example, the agent may decide not to choose a particular perception if it knows that the perception would need more time than available to be included in the belief base through the brf method. A third possibility would be the inclusion of perception filters, as described by Stabile Jr and Sichman, 2015b. Depending on the time remaining, it would be possible to use more restrictive or less restrictive perception filters to make the selection of perceptions.

For the *brf* function, it is possible to define behaviors based on time. One possibility is that if there is not enough time, to not analyze the entire belief base in search for inconsistencies. Another possibility is to perform a more straightforward operation, such as updating the belief base, performing an insertion, or removing a belief without analyzing inconsistencies.

Algorithm 7 Belief Manager

1: $timer.start()$
2: $while\ elapsedTime() < t_{bm}\ and\ P \neq \emptyset$
3: $\quad p := getNextPercept(B, P, t_{bm} - timer.elapsedTime())$
4: $\quad B := brf(B, p, P, t_{bm} - timer.elapsedTime())$
5: $\quad removeFrom(P, p)$
6: $end - while$

6.4 Intention Generator

The *Intention Generator* presented in Algorithm 8 analyses the Beliefs, Desires, Plans, and Intentions sets and controls the inclusion and exclusion of intentions in the Intentions set, joining desires and plans that the agent must execute in order to achieve them. While there is time left, the agent iterates through the algorithm, and in each iteration, the *reconsider* function on line 3 decides if the time of the iteration should be used to generate new intentions or to reconsider one of the current intentions. If the function chooses to reconsider, line 4 selects an intention, line 5 analyzes the intention and checks if the agent should keep it. If not, line 6 drops the intention. If the decision is to keep it, line 8 checks if there is a better plan to achieve the intention. Conversely, if the function chooses not to reconsider, the function *getDesire* selects a new desire in line 11. The function *evaluate* (line 12) checks if the selected desire can become an intention, for instance, by verifying if the desire does not cause conflict with other intentions or if it is possible to achieve it. If

this evaluation succeeds, the algorithm finds a plan to achieve the desire (line 13), creating an intention i composed of the desire d and the chosen plan π and adds it to the intention set (line 14).

Algorithm 8 Intention Generator

```
1:  timer.start()
2:  while timer.elapsedTime() < t_ig
3:     if reconsider(B, I, t_ig − timer.elapsedTime()) then
4:        i = getIntention(B, I, t_ig − timer.elapsedTime())
5:        if drop(B, i, t_ig − timer.elapsedTime())
6:           removeFrom(I, i)
7:        else
8:           choosePlan(B, i, Π, t_ig − timer.elapsedTime())
9:        end − if
10:    else
11:       d := getDesire(B, D, I, t_ig − timer.elapsedTime())
12:       if(evaluate(B, d, I, t_ig − timer.elapsedTime()))
13:          i_{d,π} := choosePlan(B, d, Π, t_ig − timer.elapsedTime())
14:          addTo(I, i_{d,π})
15:    end − if
16: end − while
```

6.5 Intention Executor

As our goal is to ensure that the agent executes an action in the environment by the end of t_Δ time, the *Intention Executor* must be able to choose an external action to be executed by the Monitor before the time runs out. The proposed *Intention Executor* uses a concept we call *default action*. If t_{ie} is not long enough for the *Intention Executor* to find an action to perform, the agent will, at the end of time t_{ie}, perform this default action. This action is domain-dependent, and the agent designer is responsible for defining it in the agent description. However, nothing prevents it from being changed during agent execution. This default action ensures that at the end of time t_Δ, the agent will act on the environment.

In order to execute the selected plans, we propose a mechanism for the *Intention Executor* where the algorithm iterates through the plan's actions contained in the agent's intentions, executing their internal actions and adding the external actions to a priority queue of delayed intentions ($\boldsymbol{DI}$). At the end of the time t_{ie}, the monitor module executes the action with the most significant priority value.

Another concept used in *Intention Executor* is the concept of Delayed Intention. During the execution of the *Intention Executor*, the agent may have more than one intention, which means that there is more than one plan being executed by the agent simultaneously. Thus, given that the purpose of the *Intention Executor* is to select a single external action to be performed, in case the two plans have external actions, only one can be selected for each execution. To not waste processing time, we created a memory mechanism called

Delayed Intention Queue (*DI*) that stores the information that the following action in the plan present in the intention is external. Thus, the next time the *Intention Executor* executes, this information can be retrieved quickly so that with low time consumption, the component can select an action for α. Being a priority queue, we can store intentions in the desired order to retrieve first the most relevant intentions according to the highest priority function HP.

We present the *Intention Executor* on Algorithm 9. The objective of this module is to choose an external action to be α so that the monitor component executes this α action in the environment.

Algorithm 9 Intention Executor

```
1:  timer.start()
2:  α = defaultAction

3:  while not empty(DelayedI)
4:     i = pop(DelayedI)
5:     if sound(B, I, i, t_ie − timer.elapsedTime())
6:        α = head(i_π)
7:        break
8:     else
9:        reconsider(i)
10:    end − if
11: end − if

12: while timer.elapsedTime() < t_ie
13:    i := nextIntention(I, DelayedI, α, t_ie − timer.elapsedTime())
14:    if not (empty(i_π) or succeeded(B, I, i, t_ie − timer.elapsedTime())
             or impossible(B, I, i, t_ie − timer.elapsedTime()))
15:       if type(head(i_π)) = externalAction
16:          if HighestPriority(i_π, α) = i_π
17:             addTo(DelayedI, i_α)
18:             α = head(i_π)
19:          else
20:             addTo(DelayedI, i)
21:          end − if
22:       else
23:          execute(head(i_π))
24:       end − if
25:    else
26:       reconsider(i)
27:    end − if
28: end − while
```

Line 2 illustrates the default action concept. The α variable stores the default action $\alpha_{default}$. If by the end of time t_{ie} no better action was found and stored on this variable, the Monitor will execute the $\alpha_{default}$ action on the environment. The rest of the algorithm

consists of two steps. A step that evaluates actions found in previous executions and a step that searches for new actions to perform.

Lines 3 through 11 feature the step where actions found in previous executions are evaluated. If DI is not empty (line 3), the intention with the highest priority is selected for evaluation (line 4). This evaluation (line 5) checks if executing the action in the environment is still possible. If positive, the action is selected for execution being stored in α (line 6). If not, the algorithm marks this intention for reconsideration, as the current plan is no longer sound (line 9).

Lines 12 through 28 feature the step responsible for executing internal actions and seeking new external actions. While there is time left (line 12), the algorithm chooses an intention i for execution that is both not in the *delayed intention queue* DI and not in alpha (line 13). Not selecting intentions in DI or alpha means that analyzed intentions whose external action was not yet performed are not analyzed. Then, the algorithm checks whether it can carry on with the intention by analyzing whether the intention's plan (i_π) is empty, the intention was achieved, or the intention is impossible to achieve through this plan (line 14). If the algorithm does not find any of these situations, it checks whether the type of the first action in the intention's plan is internal or external (line 15). If it is an internal action, the agent performs it at line 23. If it is external, the action is analyzed, and using the function HP, the component compares the action's priority with the priority of the current action selected for execution (line 16). In case this new action from i has a higher priority, the algorithm selects it for execution by storing the first action of the intention's plan in the α variable (line 17) and adds the intention of the previously selected action (i_α) to the *delayed intention queue* (line 18). If the new action from i has a lower priority, the algorithm adds this intention to the *delayed intention queue* (line 20). If the plan is empty, the intention was achieved, or the intention is impossible to achieve through this plan, the component marks the intention for reconsidering (line 26).

6.6 Monitor

The Monitor component shown in Algorithm 10 is responsible for controlling the execution of the other modules of the architecture. In line 1, the algorithm divides the time t_Δ for each of the modules in the best way found. In lines 3 to 5, the algorithm executes each module with its own execution time. The monitor can then execute each module according to that choice and interrupt execution at the end of the determined time.

As soon as the modules' execution times end, the monitor algorithm executes the action stored in the α variable by the Intention Executor (line 6). This action may be the default action or an action of some plan, ensuring that an action is performed in the environment after t_Δ time has passed. The action's execution by the monitor, instead of the *Intention Executor*, guarantees that the action will execute precisely at the end of t_Δ time. Executing the action through the *Intention Executor* could generate a situation where some other section of the module had a longer execution time than expected, which would delay the execution of the action, loosing the expected time guarantee.

Algorithm 10 Monitor

1: $t_{bm}, t_{ig}, t_{ie} = calculateTime(t_{\Delta})$
2: $while(alive(agent))$
3: $\quad execute(beliefManager, t_{bm})$
4: $\quad execute(intentionGenerator, t_{ig})$
5: $\quad execute(intentionExecutor, t_{ie})$
6: $\quad act(\alpha)$
7: $end - while$

Chapter 7

Anytime Jason

In order to validate the model and show that it allows achieving the research objectives, we chose to implement the model and carry out experiments using the BDI Jason agent programming language. The main reasons are its popularity in the area of multi-agent systems and because it is a well-documented open-source language. Using an existing and well-accepted language allows us to compare the agents' behavior and identify the proposed architecture's positive and negative points.

We then adopted as a starting point the architecture described in section 2.5 and made the necessary changes to create a Jason architecture whose execution follows the theoretical model described in section 2.1. We named this architecture `Anytime Jason`. However, it is impossible to change how the language works entirely. For example, Jason, being an implementation of the AgentSpeak language, uses events as one of its primary mechanisms. Neither the model proposed in Wooldridge, 2000 nor the model we defined in the section 6 has any mention of events.

On the other hand, ignoring the event mechanism would completely mischaracterize the Jason language, indicating that the model defined here is not flexible enough to be implemented by the existing languages. We then decided to draw the best possible parallels to bring the language as close to the theoretical model presented.

In the following sections, we will present the implementation made for the construction of `Anytime Jason` with the parallel components based on Jason version 2.5 available at http://jason.sourceforge.net/.

The first and foremost necessary modification is due to the time requirement. Rather than repeatedly executing the steps in the reasoning cycle, as done by default architecture, it is necessary to link the agent's reasoning cycle to the agent's expected response time. Thus, we draw a parallel between the *Belief Manager* and the *Sense* function, the *Intention Generator* and the *Deliberate* function, and the *Intention Executor* and *Act* function. In this way, by implementing the behaviors defined for the *Belief Manager*, *Intention Generator*, and *Intention Executor* modules to the *Sense*, *Deliberate*, and *Act* methods, we can control the execution of the agent's reasoning cycle. It is also interesting to elucidate the difference between the classic Jason reasoning cycle and the `Anytime Jason` cycle. In the reasoning cycle of a Jason agent, the architecture executes each of the internal functions (Sense,

Deliberate, and Act) only once. That is, it analyses one event, creates a maximum of one intention, and executes only one action (internal or external). In the `Anytime Jason` cycle, the architecture may evaluate multiple events, create multiple intentions, and execute multiple internal actions. What indicates the end of an `Anytime Jason` cycle is the execution of an action in the environment.

The first step was to make the calling of these functions and their execution time managed by the monitor module. For this, when creating an `Anytime Jason` agent, the Jason compiler creates a monitor module and informs it of the response time defined in the agent's creation code. The monitor then calculates the time the action has to execute (the current time plus the maximum response time) and starts executing the *Sense*, *Deliberate*, and *Act* functions.

The second modification aims to make the *Sense*, *Deliberate*, and *Act* functions reach one of the desired capabilities of Anytime Algorithms. The algorithm can be interrupted and continued later. This capability allows us a behavior where, during the execution of the agent, if time runs out in the middle of a process, when the compiler gets back to it, it can continue executing instead of having to restart it. Let us use the *Deliberate* function as an example again. For example, suppose during the execution of *Deliberate*, the Semantic Rules *SelEv* and *FindOp* concluded, but *AddIM* has not executed. It means that the compiler has selected an event and already found the plan it will use to respond to it but has not yet added this information to the agent's intentions. If the time runs out at that point, we would have to interrupt the execution, and we would lose information about the event and the chosen plan since the execution of the *Deliberate* function always starts with the semantic rule *SelEv*. To solve this issue, we changed the operation of each function so that if they are interrupted before the execution of all Semantic Rules, they will continue where they left off instead of starting the process again. This behavior also gives us a compelling advantage in cases where the agent's response time is shorter than the time to execute all Semantic Rules. For example, suppose the response time is only sufficient to execute a few Semantic Rules. In this case, the default Jason architecture would execute all the Semantic Rules and never execute an action in time. However, the `Anytime Jason` architecture would execute those few Semantic Rules and perform a default action. On the next cycle, it would then continue from the last executed Semantic Rule and find an action that moves the agent towards its goal. This feature reduces the minimum processing time required for the agent to be used compared to the default architecture.

However, making the *Sense*, *Deliberate*, and *Act* functions interruptable at any moment is a more complex problem to be solved. This complexity happens because the Semantic Rules make several changes in the agent's state, either by removing elements from queues and lists or changing state variables. Besides, the Java language used in Jason's implementation neither allows a function to be interrupted at any time nor to resume execution later. As a result, interrupting an execution is exceptionally problematic. Even if it was viable, the interruption might occur in code sections where relevant information changed. The inability to resume execution from the point where it stopped causes a high risk of generating inconsistent states, where beliefs, messages, events, or intentions are inconsistent. What can be done in this case is to select some particular points in the code where the interruption can happen safely. The agent executes, and if the time runs out, the execution terminates as soon as it reaches an interruption point. In the next cycle, the

agent continues from the point of interruption. Creating these breakpoints is a complex task; the more points, the more complex the code.

Additionally, inserting too many breakpoints can cause much time to be wasted while trying to figure out whether the component should stop or not. It is, therefore, necessary to balance the number of points. The intuitive idea is to place these points between the Semantic Rules since, by definition, each one executes independently and is responsible for specific behavior. We then define that once a Semantic Rule starts, it will execute until the end. When finished, if there is still time to process, the next Semantic Rule is executed, and so on.

However, this behavior is not ideal since we would like the execution to stop immediately after the time-bound expires. Instead, we would have the modules continue to run for an indefinite time after the time-bound expires. The solution we proposed to this problem is only to execute the next Semantic Rule if there is enough time for it to execute before time runs out. Thus, we would guarantee that all modules have finished processing at the end of the time-bound. This behavior is related to one of the requirements of implementing anytime algorithms: "Execution is deterministic over time". That is, the elapsed runtime of any deterministic function is consistent over repeated activations with the same input. Thus, knowing the entry, we could predict precisely how long it would take to execute the next Semantic Rule. The problem is that the Java language does not guarantee this property. Besides, the most used operating systems also do not guarantee other properties required to guarantee this property, like the following ones:

- Scheduling processes are based on priorities. At each moment, the process with the highest priority among all ready processes is running;
- If no events occur, the highest priority process will remain active. If two ready processes have the same priority, one of them will be selected at random for execution;
- When a process is running and a process with a higher priority is ready, the latter becomes immediately active;
- A process can sleep until a specific event occurs. The process is ready immediately when the event occurs.

As such, we have no guarantee of how long a Semantic Rule will take to run. So we can estimate how likely a given Semantic Rule is to finish before time runs out. If the probability is large enough, the Semantic Rule executes. We calculate this probability according to the time the Semantic Rule takes to execute. However, since we do not have a deterministic execution concerning time, we can only use the time taken by previous executions as a basis for calculating these probabilities.

In order to acquire information on the execution times, we have two possibilities. One is to measure the time while the agent runs and create a mechanism that updates the probabilities. However, this type of behavior has a significant disadvantage: the overhead produced by the inclusion of mechanisms that will execute during the reasoning cycle of the agent. Since we want to make the agent more responsive, adding even more mechanisms does not seem to be the ideal solution. Besides that, when the agent starts, there would be no information available on the execution time of the Semantic Rules for a decision to be

made.

The second possibility, adopted in this work, is to create a profiling step for the agent. In this step, we execute the agent in the environment (or in a simulation) to measure the execution times, record them, and calculate all the probabilities. In this way, it is not necessary to use the agent's processing time to calculate them. Performing time measurement during the agent's execution has the advantage that we can take into account information such as the agent's domain, its definition (code), and the computational capacity of the computer on which it will execute. Besides that, one may dynamically identify how much the passage of time affects agent processing times. For instance, more execution time can lead to a more extensive base of beliefs, resulting in a longer belief update. In the profiling stage, the agent executes similarly to the default architecture, where each Semantic Rule is executed once for each reasoning cycle, and its execution time is stored. Also, we store information about the number of percepts evaluated, events generated, intentions created, internal and external actions, and other performance data for the agent. Based on this information, we generate the performance profiles of each component in each agent, which are estimates of the performance of each one. The quality of the Sense component is measured by how many percent of the total beliefs it was able to assess. Deliberate's measures how many percent of the events it would evaluate. On the other hand, the quality of the Act's response is measured based on the probability of finding an external action within the given time. As the quality of the result of a component depends on the quality of the previous, we use for the Deliberate and Act components a type of performance profile called Conditional Performance Profile, described in Zilberstein, 1993. Thus, we have three quality functions (performance profiles), and we aim to maximize the qualities of the responses given by the components.

We then use the ϵ-constraint method described in Miettinen, 2008 to optimize the time allocation. This method optimizes one of the functions while defining constraints for the others. So, in our case, the agent designer can, for example, state that the Sense component can never evaluate less than 20% of the total perceptions. Thus, the ϵ-constraint method will calculate a set of time allocations that maximize the quality of the responses within these restrictions. Then, based on the user restrictions, the Monitor $\mathcal{M}$ selects one of the time allocations in the set and executes the other modules accordingly.

With this information, the architecture can better calculate where to allocate the available time and guarantee that the processing will finish before the time-bound with a probability that we can also control.

The following sections present the changes to the *Sense*, *Deliberate*, and *Act* functions to behave according to the definitions of *Belief Manager*, *Intention Generator*, and *Intention Executor*. We detail the code changes required to make a Jason agent run through the `Anytime Jason` architecture in Appendix B.

7.1 Belief Manager Implementation

The next objective was to change the behavior of the *Sense* function to suit the Algorithm 7. The most noticeable problem is that within the *Sense* function, there is not only the updating of beliefs but also the Semantic Rule ProcMsg that deals with receiving and

analyzing messages. Removing this Semantic Rule would be theoretically possible since communication between agents could carry out through the environment. However, it is not in our interest to remove language capabilities. Even more, because it is a widely used capacity in multi-agent systems. Therefore, we proposed to divide time between the *buf* and *ProcMsg* functions, depending on the agent's domain. In scenarios with little communication and the agent having many perceptions, assigning more execution time to the *buf* function is possible. In contrast, in the opposite case, in negotiation scenarios, for example, it is possible to analyze several messages in the same `Anytime Jason` cycle. This division can be automatically calculated by considering the number of perceptions, the size of the belief base, and the number of messages received.

The next step is to change the *buf* function. The Algorithm 11 presents a simplified pseudo-code that illustrates the operation of the *buf* function.

Algorithm 11 Jason *buf* function

```
1:  let perc be the set of perceptions received as parameter
2:  for each perception b in the BeliefBase
3:      if b is not in perc
4:          remove b from the BeliefBase
5:          generate trigger for b removal
6:      else
7:          remove b from perc
8:      end – if
9:  end – for
10: for each perception p in perc
11:         add p to the BeliefBase
12:         generate trigger for b insertion
13: end – for
```

As analyzed by Stabile Jr and Sichman, 2015b, the *buf* function is one of the most time-consuming functions in all of Jason's architecture. With this information, it is not feasible to execute the *buf* function only when there is time for it to complete, as there is a high chance that the time needed would be longer than the time-bound. Therefore, it is necessary to add breakpoints within the function. The problem with adding breakpoints in the function is that all beliefs that came from perceptions are removed, and new perceptions are evaluated only after that. Therefore, if there is no time to execute the entire function, the section that will not execute is the section that brings new information to the agent, negatively affecting its reactivity.

In order to reduce the occurrence of this problem and bring the *buf* function closer to the theoretical definition of *Belief Manager*, we inverted the logic and added breakpoints so that perceptions evaluate first. We present the simplified pseudo-code of the modified *buf* function in Algorithm 12.

With this construction of the algorithm, we can analyze the perceptions first, increasing the reactivity of the agent since the insertion events are generated before the removal events, allowing the *Intention Generator* to choose them earlier. Besides, through the

Algorithm 12 Anytime *buf* function

```
1: let perc be the set of perceptions received as parameter
2: oldBB = Belief Base
3: remove all perceptions from Belief Base
4: while perc is not empty
5:     if there is not enough time for the loop, stop
6:     p = getNextPercept(perc, t)
7:     add p to the Belief Base
8:     if p is in oldBB
9:         remove p from oldBB
10:    else
11:        generate event for p creation
12:    end − if
13: end − while
14: for each perception b in oldBB
15:    if there is not enough time for the loop, stop
16:    generate event for b removal
17: end − for
```

getNextPercept function, it is possible to choose the most critical perceptions for the agent. If there is not enough time to select all perceptions, the most important ones will be analyzed, and the agent will have the most relevant information for its decision-making. This function can take into account if the perception is used in triggers, if it is part of a plan context, it can contain perception filters such as those described by Stabile Jr and Sichman, 2015b and many other behaviors that can be customized depending on the agent's domain.

We also proposed a mechanism of relevance for perception filters based on the work by Lorini and Piunti, 2010 and Stabile Jr and Sichman, 2015a. In this mechanism, the agent designer can define a series of perception filters and their relevance. When one of these filters is active, the function *getNextPercept* will return first the percepts that go through the filter. As an example, the internal action ".filter.create(insectfilter,1,[[0,ne,insect]]);" would create a filter named 'ìnsectfilter" with a relevance of 1 where every percept not started with 'ìnsect" would be evaluated first. Only after that percepts started with "insect" would be evaluated. In case multiple filters are active, the function *getNextPercept* would select percepts based on the relevance value of the filters (biggest relevance first). Filters are named so the agent can create, delete, activate and deactivate filters as internal actions.

7.2 Intention Generator Implementation

Let us first remember the *Intention Generator* algorithm. The *Intention Generator* consists of two main parts. One is responsible for carrying out intention reconsidering, and the other is responsible for analyzing and including new intentions. The first problem is that Jason has no default mechanism for intention reconsideration. Creating an intention

reconsideration mechanism to make Jason fit the theoretical model would need another research. Thus, we only have the option of not considering this part of the model. For better visualization, the *Intention Generator* algorithm without intention reconsidering is presented again in Algorithm 13.

Algorithm 13 Intention Generator

1: $t = 0$
2: $while\ t < t_{ig}$
3: $\quad d := getDesire(B, D, I, t)$
4: $\quad if(evaluate(B, d, I, t_{ig} - t))$
5: $\quad\quad \pi_d := findPlan(B, d, P, t_{ig} - t)$
6: $\quad\quad addTo(I, d)$
7: $\quad\quad addTo(\Pi, \pi_d)$
8: $end - while$

Interestingly, we can trace a considerable correlation between the methods described in the theoretical model and Jason's Semantic Rules. In line 3, the getDesire method selects an agent's desire to be evaluated and possibly become an intention. Likewise, Semantic Rule SelEv selects an agent event to be evaluated and possibly become an intention. By language definition, the *SelEv* function is customizable, which allows us to implement event selection behaviors based on the agent's domain.

In line 4, we have the function evaluate that checks whether the desire should become an intention by analyzing current beliefs and intentions. In line 5, the *findPlan* function searches for a plan to achieve the desire by searching its library of plans. Similarly, Semantic Rule *FindOp* analyzes the selected event and checks if there is a plan that responds to the event according to the agent's beliefs.

Finally, lines 6 and 7 add the new intention and plan to their respective sets. Semantic Rule AddIm performs the same task.

Thus, we have to execute the *Deliberate* function, which is already very similar to the *Intention Generator*, and no significant changes are necessary to control its execution time.

7.3 Intention Executor Implementation

The first necessary modification to adapt the *Act* function to the *Intention Executor* was to add the *Delayed Intention Queue* $\mathcal{DI}$ to the architecture. Therefore, we added a priority list that stores *DelayedIntention* objects. These objects contain an intention, as defined in the formal model.

The second modification aimed to add the mechanism described in the lines 3 to 11 of the Algorithm 9, that evaluates actions found in previous executions. The algorithm goes through $\mathcal{DI}$ and analyzes the previously found actions. This mechanism works through a new Semantic Rule called *CheckDI* that is executed by the *Act* function before the other Semantic Rules start.

Following the end of the $\mathcal{DI}$ set analysis, the module executes the Semantic Rules as long as there is time. Since Semantic Action ProcAct aims to verify the result of executing an action in the environment, it does not perform more than once. Thus, in the **`Anytime Jason`** cycle, the only semantic rules that can execute multiple times are SelInt, ExecInt, and ClrInt.

Following the theoretical model, after analyzing the delayed actions, the next step is to execute internal actions and seek new external actions. The first step is to choose an intention to be executed. This selection is made on Jason by the Semantic Rule SelInt. Then, it checks if the plan is not empty, completed, or unachievable. If none of these issues appear, the module checks whether the next step will be an internal or external action. In case it is an internal action, it executes immediately. Otherwise, the module calculates its utility value, and the action is added to α or $\mathcal{DI}$ depending on the calculated utility value. In Jason, the Semantic Rule ExecInt is responsible for these processes. Thus, a change was made to the Semantic Rule ExecInt so that when detecting the execution of external action, it creates a DelayedIntention object and adds it to variable α or to the $\mathcal{DI}$ queue.

Finally, it is necessary to execute the Semantic Rule ClrInt in order to check whether the plan should continue to be executed or not (if it has finished or if the intention has been achieved, for example).

With that, we can illustrate the flow of execution of the *Act* function with Figure 7.1. In it, we show that the processing starts with the Semantic Rule *CheckDI*, followed by *ProcAct*, and after that, the functions *SelInt*, *ExecInt*, and *ClrInt* execute in a loop.

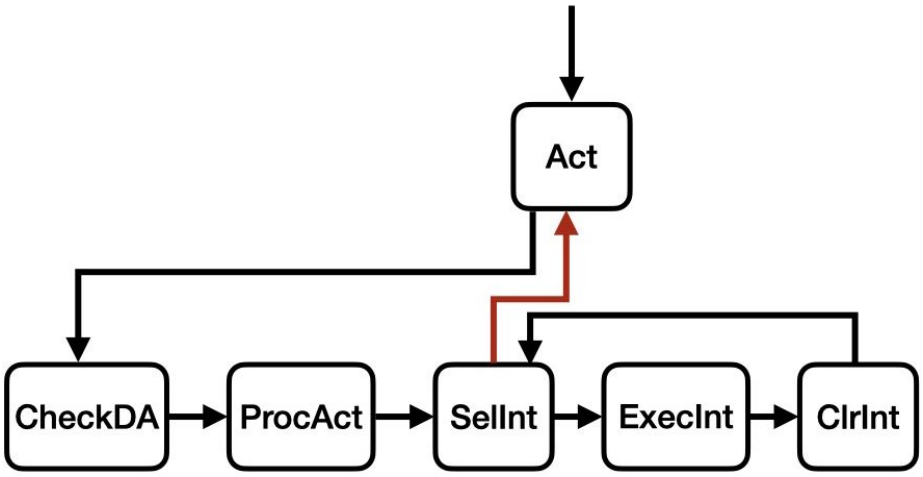


Figure 7.1: Act *function for implementing the* Intention Executor.

To illustrate how adding the Delayed Intentions queue impacts the plans execution order and how this benefits agent responsiveness, we will present an example of how Jason and **`Anytime Jason`** would execute the same set of plans. Algorithm 14 contains three plans. Each plan contains a number of internal actions (I) and external actions (E).

Jason's default architecture runs plans on a round-robin basis. Thus, the order of execution of the actions would be: $I1.1, E2.1, E3.1, I1.2, I2.2, I3.2, I1.3, I2.3, I3.3, I1.4, I2.4, E3.4, E1.5$. If this agent were being executed in a scenario such as the MAPC, where it is necessary

Algorithm 14 Plans to be executed.

1: $Plan1$: $I1.1, I1.2, I1.3, I1.4, E1.5$
2: $Plan2$: $E2.1, I2.2, I2.3, I2.4$
3: $Plan3$: $E3.1, I3.2, I3.3, E3.4$

to execute an external action every 4 seconds, the executions would be distributed as presented in Table 7.1 in each of the steps of the simulator.

Step	Actions analysed	External action executed
1	I1.1, E2.1	E2.1
2	E3.1	E3.1
3	I1.2, I2.2, I3.2, I1.3, I2.3, I3.3, I1.4, I2.4, E3.4	E3.4
4	E1.5	E1.5

Table 7.1: *Plan execution order on default Jason.*

This execution order presents two problems. The first happens in Steps 2 and 4, where the agent finds an external action so fast that it does not process anything else and waits for the action to be executed in the environment, thus wasting processing time. The second problem happens in Step 3, when the agent needs to execute many internal actions until it finds another external action. This may cause the agent to exceed the time limit to find the action $E3.4$, causing it to fail to execute in the particular simulator step.

In `Anytime Jason`, it is possible to add priorities to each plan through annotations. Thus, we can define, for example, that Plan 1 has the highest priority, followed by Plan 3 and finally, Plan 2. The execution order is presented in Table 7.2. `Anytime Jason` also executes the plans in round-robin mode. Thus, the first two actions executed are $I1.1$ and $E2.1$. The first difference is that instead of immediately executing action $E2.1$ on the environment, `Anytime Jason` stores this action in a so-called *alpha* action. This alpha action is the external action that will be executed when processing time ends. The architecture thus continues executing the actions of the plans. The following external action is $E3.1$. As this is an external action of a higher priority plan, the architecture exchanges the alpha action: now $E3.1$ is stored as the current alpha action, and $E2.1$ is placed in the DelayedIntentions queue. Continuing the processing, $I1.2$ is executed, as it is an internal action. Let us assume that the module's processing time is sufficient for analysing four actions. Thus, in Step 1 of the simulator, the $E3.1$ action, which is the alpha action, is executed in the environment.

Next, the following action that should be analyzed is action $I2.2$. However, as this plan is part of a DelayedIntention, while action $E2.1$ is not executed in the environment, the following actions of Plan 2 cannot be analyzed. Thus, the architecture performs actions $I3.2$, $I1.3$, $I3.3$, and $I1.4$. We can see that, at the end of the processing time, the architecture has not evaluated any external action. However, the architecture had already stored the $E2.1$ action as a DelayedIntention. So the architecture can run it in the environment as the action for Step 2.

Step	Actions analysed	External action executed
1	I1.1, E2.1, E3.1, I1.2	E3.1
2	I3.2, I1.3, I3.3, I1.4	E2.1
3	I2.2, E3.4, E1.5, I2.3	E1.5
4	I2.4	E3.4

Table 7.2: *Plan execution order on* `Anytime Jason`.

Now that the architecture executes action $E2.1$ on the environment, action $I2.2$ can also be executed. Thus, the execution of the rest of the actions continues in the same way. Actions $I2.2$, $E3.4$, $E1.5$, and $I2.3$ are analysed. As Plan 1 has a higher priority, $E1.5$ is chosen as the current alpha action and $E3.4$ is placed in the DelayedIntention queue. At the end of the time, the architecture executes $E1.5$ on the environment for Step 3. In Step 4, the last internal action executes ($I2.4$) and $E3.4$ is executed on the environment.

It is then possible to verify that the **`Anytime Jason`** architecture analyses the actions evenly among the Steps, reducing the incidence of both problems presented by the default architecture.

7.4 Monitor Implementation

According to the definition of the theoretical model, the Monitor module is responsible for starting the execution of the other modules and executing the action in the environment. For this, the Monitor creates one Java process for each module. Next, the Monitor calculates the total time each module will run based on the profiling data available, the response time, and the restrictions provided in the agent description file.

Based on empirical analysis, we found that the Jason standard architecture benefits from the interleaving of the three modules. Thus, to maximize agent performance, the Monitor executes each Semantic Rule only once and records the time used by the module. For example, consider a scenario where the Sense module should run for 10ms, Deliberate for 15ms, and Act for 20ms in a given **`Anytime Jason`** cycle. Then, the Monitor starts up by running the Sense module. Due to many percepts, it is impossible to update them all. Thus, after 10ms, the Monitor stops executing the Sense module. Then, Monitor executes each Semantic Rule of the Deliberate module once. This process consumes 8ms, which the Monitor logs. The Monitor then executes each Semantic Rule of the Act module once. This process consumes 5ms. At the end of this iteration, the Sense module has no more processing time, while the Deliberate module has 7ms and the Act 15ms. Starting a new iteration, the Monitor will not run the Sense module as it has run out of time. Starting the Deliberate module, the 7ms was insufficient to complete all the Semantic Rules. Thus, the Monitor interrupts the execution and registers that the execution must continue where it left off in the next **`Anytime Jason`** cycle. Then the Monitor will run the Act module until its time runs out. Once the processing times are over, the Monitor executes the best action found in the environment and starts a new cycle.

Chapter 8

Related work

8.1 Parallel agent architectures

While participating in the RoboCup soccer simulation competition, Kostiadis and Hu, 2000 encountered the same problem described in Section 1.1. Given the nature of the RoboCup simulator, the response time of a soccer player agent is critical, as the server operates with 100ms cycles to execute actions and 150ms cycles to provide perception data. Kostiadis and Hu, 2000 describe that first, the agent needs to receive sensory information from the server. Then the agent needs to "reason" to produce the desired action. Finally, the agent must send this action back to the server. Analyzing the implementations of other RoboCup teams, Kostiadis and Hu, 2000 realized that most agent implementations performed these steps using a single thread per agent in a serial processing loop, as shown in Figure 8.1.

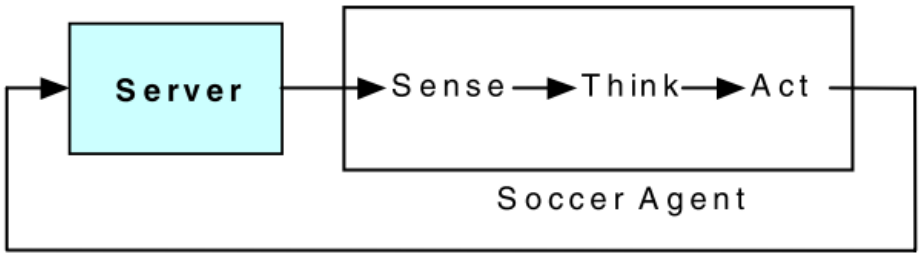


Figure 8.1: *Single-threaded agent model. (From Kostiadis and Hu, 2000)*

Kostiadis and Hu, 2000 proposed that instead of a single thread, a process can have several threads, performing different operations independently and without affecting each other. This architecture allows the agent to use a separate thread for the three tasks. Figure 8.2 shows the proposed multi-thread model.

The agent performs the three main tasks simultaneously (or in parallel on multiprocessor hardware), minimizing delays in communication operations. This way, the agent can dedicate the maximum amount of processing power and time to its reasoning process. Kostiadis and Hu, 2000 then conducted a comparative evaluation between the single

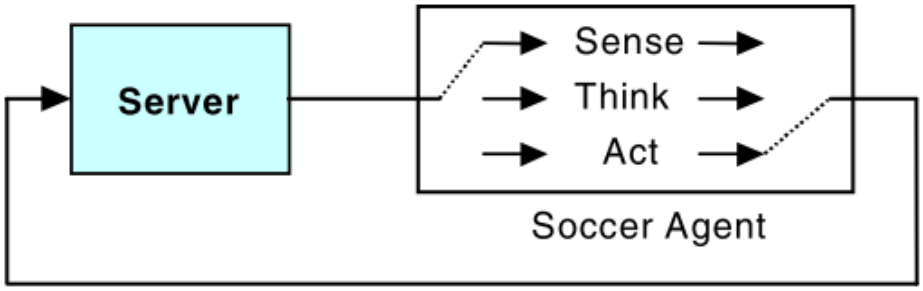


Figure 8.2: *Multithreaded agent model. (From Kostiadis and Hu, 2000)*

thread model and the multiple thread model. Its objective was to evaluate the number of actions its agents lost due to the high processing time.

The authors' results show a significant reduction in the loss of actions. This model does not solve the problem proposed in Section 1.1 because it still presents losses and is not a BDI model. However, it offers a concept of internal separation of the agent that can facilitate the control of an agent's processing time by controlling the execution of less complex parts.

Based on the work of Kostiadis and Hu, 2000, Zhang and Huang, 2005 proposed to apply a similar model to the BDI architecture since the processing of BDI agents can also be divided into "perception", "reasoning" and "action" and are commonly performed sequentially. In this work, the authors claim that an agent built with this parallel BDI architecture can deliberate on new beliefs and execute intentions simultaneously, in addition to responding quickly to changes in the environment. Thus, such an agent would have a more natural way of simulating human reasoning since humans can perform all three tasks simultaneously.

Zhang and Huang, 2007 formalize this architecture into a general framework for the parallel BDI agent model. Under this general framework, parallel BDI agents with different configurations can be built, depending on the availability of physical resources, such as sensors and actuators. Figure 8.3 illustrates this model.

The authors divide the architecture into three main components. The *Belief Manager*', the *Intention Generator*' and the *Intention Executor*'. Each major component is composed of smaller parts. Are they:

- **Belief Manager:** Responsible for detecting changes in the environment and managing the agent's beliefs;
  - **EM (Environment Monitor):** Each EM serves as a collector of information from heterogeneous sensors that an agent may have, monitoring information from the environment through some sensor or sensory organs, such as a camera or human eyes, and converts the information into an abstract representation. Each EM sends the converted information to the BG;
  - **BG (Belief Generator):** The BG merges the information passed by EMs and converts it into beliefs. For example, a person's eyes see, and the ears hear a car coming. The visual and audio information will come through two separate

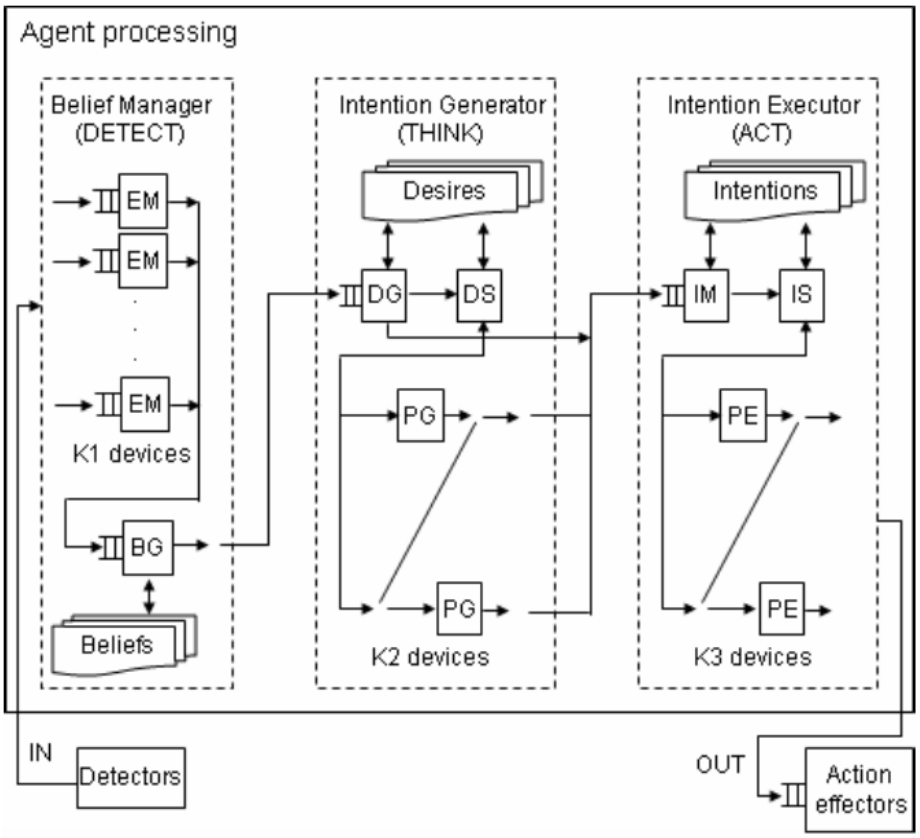


Figure 8.3: *General model for parallel BDI agents. (Taken from Zhang and Huang, 2007)*

EMs, and the BG combines the information to form a new belief. Each belief is assigned an urgency value;

- **Intention Generator:** Responsible for reasoning about new beliefs. This component includes managing the agent's desires (goals) and deliberating on plans to achieve those desires;
  - **DG (Desire Generator):** The DG produces new desires according to new beliefs. A new desire will have the same priority level as the belief that triggered its generation. A new belief can also make a current desire no longer desirable because it becomes obsolete or is not consistent with the new belief;
  - **DS (Desire Scheduler):** Identifies the most important desires to be achieved (using urgency values) and allocates them according to availability in one of the plan generator components (PG);
  - **PG (Plan Generator):** Defines the plan to be executed to achieve a specific intention, either through planning or by choosing from a plan repository. The architecture allows the use of more than one plan generator, which generates plans for different intentions;
- **Intention Executor:** Responsible for the agent's actions, interleaving and executing them;
  - **IM (Intention Manager):** Similar to the DG, this component receives the

plans generated by the PGs and can add and remove them from the intention queue.

- **IS (Intention Scheduler):** This component schedules, suspends, and resumes the execution of intentions in Plan Executors (PEs).
- **PE (Plan Executor):** Sends the plan action to the actuators to execute. The architecture allows the use of an executor for each agent's actuator.

Generally, this architecture receives information from the sensors, converts it into perceptions, re-evaluates the intentions, finds plans for them, and executes them. This architecture is interesting because the three main components (*Belief Manager*, *Intention Generator* and *Intention Executor*), just as done by Kostiadis and Hu, 2000, run in parallel. Thus, when urgent information appears, it is the first to be evaluated by the DG, is quickly inserted into the queue of intentions, and executed by the actuators. This behavior can be directly related to the main advantage of the Three-Layer Architecture. For example, when there is a need for a fast response in the Three-Layer Architecture, the reactive module provides it without the need to execute the rest of the architecture. Similarly, the architecture of Zhang and Huang, 2007 allows for immediately evaluating a high-priority belief and performing all related processing without waiting for all previous information to be processed.

Like the work of Kostiadis and Hu, 2000, this model offers a concept of internal separation of an agent, which this time is BDI, aiming to reduce its execution time. This model allows better identification of the internal components of a BDI agent in order to facilitate the control of the processing time.

From the work of Zhang and Huang, 2007, we used as a basis for our model the idea of separating the architecture into three components: *Belief Manager*, *Intention Generator* and *Intention Executor*. We identified a complex problem when we tried to combine the parallelization method with our theoretical model based on the work of Wooldridge, 2000. In the Wooldridge (2000) model, the agent receives a set of perceptions representing what the agent perceives from the environment. The analysis of perceptions is then done one at a time. For example, suppose the scenario where the agent receives a set of perceptions $p_1, p_2, p_3, ..., p_n$. Also, suppose there is an intention i that depends on information from p_1 and p_n to execute. As soon as the *Belief Manager* updates the perception p_1, the agent's belief base is in an invalid state, as the belief p_1 is up-to-date while the belief p_n is out-of-date. Thus, the i intention cannot use the information in the belief base until it is fully updated, or it would run the risk of performing an invalid action in the environment. The options for the other modules are to wait for the belief update to finish (which damages the agent's responsiveness) or to execute other intentions while waiting for the update to finish. The problem with this second option is that as the components are not synchronized, it is possible that when the other modules try again to use the belief base, it has finished the previous update and started a new one. Because of this problem, we decided not to incorporate parallelism mechanisms in the proposed architecture.

Intending to improve the Jason language's performance through parallelism, Zatelli, 2017 proposed a new agent Jason architecture called Asynchronous, inspired by Zhang and Huang, 2005. In this architecture, the *Sense*, *Deliberate*, and *Act* functions are now

executed in parallel, as shown in Figure 8.4. The internal behavior of these functions, however, remains the same. Even not using parallelism methods, the work of Zatelli, 2017 provided a basis for the implementation of our theoretical model in the Jason language by using its separation of *Sense*, *Deliberate*, and *Act* functions in different threads.

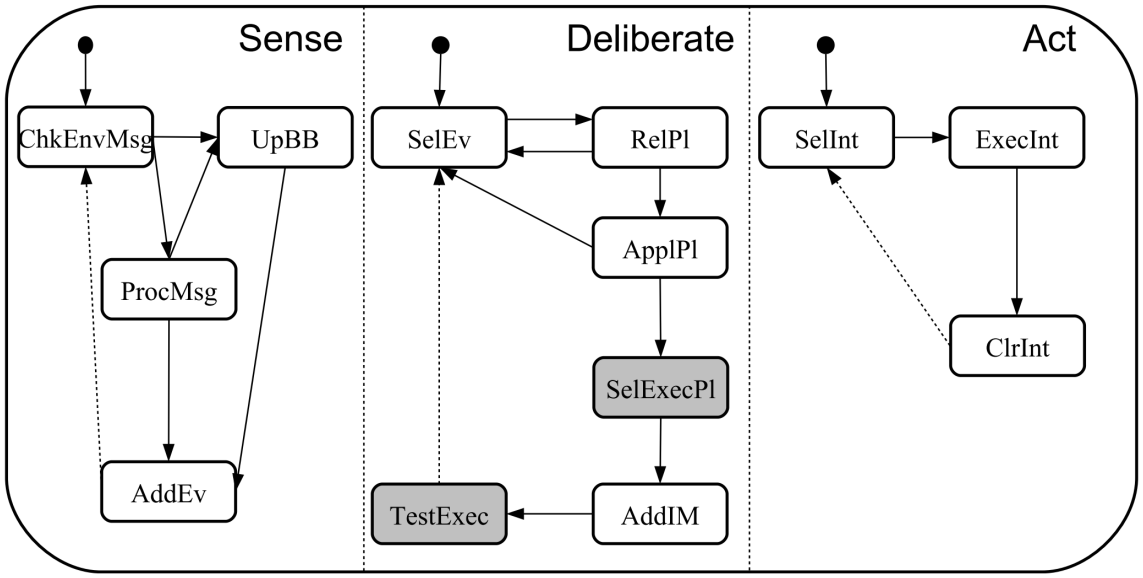


Figure 8.4: *Jason Semantic Rules execution flow by Zatelli, 2017.*

8.2 Control of reasoning time on intentions.

According to Yao and Logan, 2016, BDI agents typically pursue multiple goals in parallel. However, interleaving steps with different intents can result in conflicts. For example, executing a step in one plan can make it impossible to execute a step in another concurrent execution plan.

Previous approaches treated plans as atomic units and attempted to merge plans to minimize conflicts. Since ordering plans can not resolve some conflicts, the authors present the SA algorithm, which is an approach to Intention selection based on a Tree Search via the single-player Monte Carlo method. This algorithm is responsible for analyzing an agent's intentions and respective plans and deciding which action to take next. For this to be possible, a *tree of plans and objectives* must be built, like the one shown in Figure 8.5 that represents the relationships between goals, plans, and actions of an agent.

The root of a tree of plans and goals is a top-level goal (goal node). Its children are the plans that can achieve that goal (plan nodes). Plans can, in turn, contain subgoals (goal nodes), giving rise to a tree structure representing all possible ways an agent can achieve the higher-level goal.

Each goal plan tree records information about the conditions necessary to achieve a (sub)goal or successfully execute a plan or an action in preconditions, in-conditions, and postconditions associated with objectives, plans, and action nodes. Preconditions must be valid for executing a plan or action. In-conditions are conditions that must be maintained while pursuing an objective or plan; If a condition becomes false during the execution of

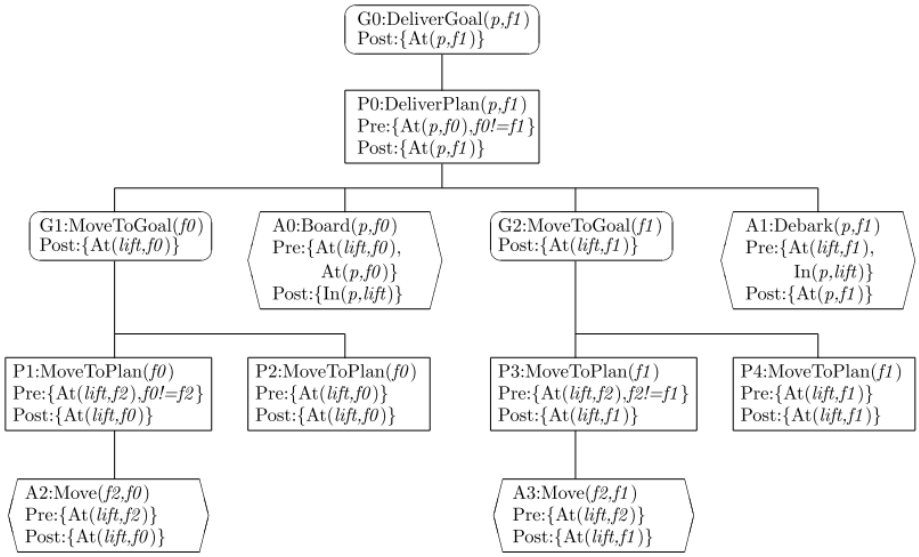


Figure 8.5: *Example of tree of plans and objectives (Taken from Yao and Logan, 2016)*

an objective or plan, the objective or plan fails. Postconditions are conditions that become true after a plan or an action executes. With access to each of the trees for the agent's intentions and beliefs, the SA algorithm will decide which action to perform.

The work of Yao and Logan, 2016 is particularly relevant for our work because the SA algorithm is an anytime algorithm. Therefore, it is possible to control the time used by an agent so that, based on its intentions, it can decide which action to perform. We then worked on extending these capabilities to the rest of the BDI architecture.

8.3 Control of reasoning time over perceptions.

According to Van Oijen and Dignum (2011), one of the problems of the BDI paradigm when linking agents to virtual environments is the lack of control over perceptions. If there is not some form of goal-directed perception, the agent will inundate with sensory information, which can result in reasoning about an enormous amount of irrelevant information. Furthermore, the absence of control is also unrealistic when we look at the physiology of human perception. Attention is considered a limited resource; for example, one cannot attend to all aspects of the environment. Also, during the execution of a task, humans tend to direct their attention to selected information from the environment that can support them in carrying out a task. This behavior suggests we should also consider a similar approach for BDI agents.

Lorini and Piunti, 2010 state that realistic cognitive agents should not waste time and energy reasoning about each piece of information obtained. For this reason, they require precise allocation strategies better to balance the use of their limited computational resources. The authors then present a computational model of a relevance-based belief update mechanism. This mechanism is responsible for filtering out all non-relevant information and considering only relevant information to the current task an agent tries to solve. The authors proposed a modification in the cycle of the BDI agent that we present

in the Algorithm 15. Before updating the belief base, each percept goes through a function that calculates its relevance based on the agent's current intentions (line 5). The more present the percept in the agent's intentions, the greater its relevance value. After being evaluated, the percept is only considered in the belief base update process if its relevance value exceeds a certain threshold.

Algorithm 15 BDI agent working cycle of Lorini and Piunti, 2010

```
1:  B := B_0;
2:  I := I_0;
3:  while true do
4:      get next percept p;
5:      if REL(I, p, B) > Δ
6:          B := brf(B, p);
7:      end if
8:      B := brf(B, p);
9:      D := options(B, I);
10:     I := filter(B, D, I);
11:     π := plan(B, I);
12:     execute(π)
13: end while
```

Van Oijen and Dignum, 2011 addresses the same problem by creating a middleware positioned between a simulator and the BDI agent. This middleware is responsible for identifying the perceptions of interest to the agent and sending it only the relevant perceptions. This process happens through a mechanism called "interest subscription management". This mechanism accesses the agent's goals and analyzes the information available in the environment. When the agent adopts an objective, the system automatically creates a set of signatures for the environment information. From that moment on, the agent will receive percepts about that information. Likewise, when an agent no longer has a goal, either because it has achieved it or given up, the system unsubscribes, ceasing to receive percepts about this information set.

In Stabile Jr and Sichman, 2015a, we proposed to give the agent the ability to control the received perceptions through the definition of predefined perception filters. We positioned these filters within the agent's execution engine, similar to the relevance filter of Lorini and Piunti, 2010. As part of its plan to achieve an intention, the agent can activate one of the perception filters that will restrict the perceptions it will receive from the environment. This mechanism allows the agent to refrain from receiving and processing information about the environment that does not affect or is of no use to the agent. Finally, we demonstrated that the use of such a mechanism could provide a significant reduction in the response time of an agent.

We used the ideas proposed in Lorini and Piunti, 2010 and Stabile Jr and Sichman, 2015a to develop the mechanism of relevance for perception filters presented in Section 7.1.

8.4 Real-Time BDI

Traldi *et al.*, 2022 recently presented research that aims to control the response time of BDI agents. In this work, the authors proposed a Real-Time BDI-based architecture that aims to ensure predictability of execution, considering concepts like computational capacity, deadline, scheduling constraints, durative actions, periodic tasks, and temporal planning deliberation. The authors structured the proposed architecture in three layers: the BDI layer, responsible for handling beliefs, desires, and intentions; the Execution and Monitoring layer, responsible for executing and monitoring plans; and the Real-Time layer, which handles the low-level execution of tasks.

Despite seeking to solve a very similar problem to ours, the architecture proposed by Traldi *et al.*, 2022 presents some undesirable aspects related to our research problem. The first problem is that the Real-Time BDI is composed of multiple layers, making it difficult for the agent to reason. The second problem is that the authors implemented the proposed model in a new agent programming language that has not yet been made available. Thus, it would be necessary to recreate the agents in a new architecture. Our work, however, seeks to use well-established languages and make it possible to control the execution time of agents with minimal change to their source code. Finally, as the language has not yet been made available, it is impossible to compare the model proposed in this work with the model proposed by Traldi *et al.*, 2022.

8.5 Synthesis

In this chapter, we present our proposed BDI agent model that allows the control of its runtime. We present both its formal description and possible implementation of this model. In this section, we present a synthesis of the works described above. The Table 8.1 contains a representation of the main aspects used by our proposal to solve the problem and how each work relates to them. The first column, *Parallel*, informs if the work makes use of algorithms executing parallelly. The *BDI* column tells you whether the solution proposed at work is related to the BDI model. The *Established language* column shows whether the job uses popular BDI agent programming languages. The *Controls intentions* column shows whether the proposed work controls the execution time of intentions and plans. Finally, the *Controls percepts* column shows whether the proposed work controls the execution time of the analysis of percepts and management of beliefs.

The work proposed by Kostiadis and Hu, 2000 is quite relevant to emphasize that there are scenarios that benefit from a shorter processing time for agents. However, it does not apply the parallelism model to BDI agents. The work of Zhang and Huang, 2007 extends the work of Kostiadis and Hu, 2000 to the BDI model but provides only theoretical tooling for this. Zatelli, 2017 then applies the proposed parallelism model to an established agent programming language, Jason; however, there is no effective control over agents' processing time.

In work presented by Yao and Logan, 2016, the authors propose a way to control the execution of intentions and plans through anytime algorithms. However, there is only control over this part of agent processing.

Paper	Parallel	BDI	Established language	Controls intentions	Controls percepts
Kostiadis and Hu, 2000	✓				
Zhang and Huang, 2007	✓	✓			
Zatelli, 2017	✓	✓	✓		
Yao and Logan, 2016		✓	✓	✓	
Lorini and Piunti, 2010		✓			✓
Van Oijen and Dignum, 2011		✓			✓
Traldi *et al.*, 2022	✓	✓		✓	
Anytime BDI architecture	✓	✓	✓	✓	✓

Table 8.1: *Synthesis of the related work.*

Similarly, Lorini and Piunti, 2010 and Van Oijen and Dignum, 2011 seek to reduce the processing time of analyzing perceptions and creating beliefs. However, there is no direct control of the processing time.

Finally, Traldi *et al.*, 2022 presented the work closest to ours. In it, the authors seek to control the execution time of the agent as a whole. However, the proposed model does not use any available BDI agent programming language, making it necessary to rewrite the agents to use it. On the other hand, our model aims to be used with any well-established agent programming language, making it possible to use already created agents without new effort.

Through this comparison, we can see that the Anytime BDI architecture described in this work is the only one that proposes the control of all aspects of the agent's reasoning through well-established agent programming languages.

Part III

Evaluation

Chapter 9

Experimental Design

To validate the model and verify if it can answer the research questions, we used techniques from the performance analysis domain in our experiments, following the experimental design guidelines presented in Jain, 1991. According to the author, the objective of a proper experiment design is to obtain the maximum information with the minimum number of experiments. The process separates the effects of various factors that can affect performance and allows us to determine whether a factor significantly affects the performance or whether the observed difference is simply due to random variations caused by measurement errors or uncontrolled parameters.

9.1 Definitions

It is important to define the meaning of four terms:

- **Response variable** is the result of an experiment. In the experiments conducted in this thesis, the response variables are the response time of the agent's actions, and the number of points scored.
- **Factors** are the variables that influence the response variable. For example, the number of perceptions affects the time it takes for an agent to respond to the server. Factors can be primary or secondary. Primary factors are those whose effects need to be quantified, while secondary factors affect performance but whose effects we do not want to quantify.
- **Levels** are the values that a factor can take. For example, the levels of the factor "number of percepts" are the number of percepts the agent receives in the experiment, e.g. 80 or 910 percepts.
- **Repetition** is a rerun of some or all of the experiments. For example, if we did three runs of the same experiment, the experiment had three repetitions.

According to Jain, 1991, the $2^k r$ factorial experimental design is one of the best experimental designs because one can check whether or not the factors have a significant effect on the response variable with the least number of experimental runs. As pointed out by the author, in the $2^k r$ factorial experimental design, the variable k stands for the number

of factors in the experiment; the number 2 stands for the number of levels in each factor, here set to two, since this is the smallest possible value to identify variations. The variable r represents the number of repetitions of each experiment. Repetitions can help determine if unanalyzed factors influence the response variable.

9.2 Example

To better illustrate the concept of the $2^k r$ factorial experimental design, a simple case with only two factors ($k = 2$) and a single repetition ($r = 1$) is helpful for illustration. Based on the example of Jain, 1991, suppose we want to evaluate the efficiency of two agents (Agent A and Agent B) in catching insects with two different tools (butterfly net and tweezers) for one minute. For this experiment, we present the results in Table 9.1.

	Tweezers	**Butterfly Net**
Agent A	15	45
Agent B	25	75

Table 9.1: *Number of insects caught after one minute.*

We must then define the variables x_A and x_B as follows:

$$x_A = \begin{cases} -1, & \text{se Tweezers} \\ 1, & \text{se Butterfly Net} \end{cases}$$

$$x_B = \begin{cases} -1, & \text{se Agent A} \\ 1, & \text{se Agent B} \end{cases}$$

We can now regress the agent efficiency over x_A and x_B using a regression model of the form:

$$\begin{aligned} 15 &= q_0 - q_A - q_B + q_{AB} \\ 45 &= q_0 + q_A - q_B - q_{AB} \\ 25 &= q_0 - q_A + q_B - q_{AB} \\ 75 &= q_0 + q_A + q_B + q_{AB} \end{aligned}$$

The first equation, for example, represents that when agent A (value -1 in x_A) uses the Tweezers (value -1 in x_B) it captures 15 insects. The value q_0 represents the capture average, being the base around which the values will vary, and q_{AB} represents the interaction between the two factors, and its value is multiplied by the value of $x_A * x_B$

We can uniquely solve these equations for the four unknown variables. The regression equation is:

$$y = 40 + 20x_A + 10x_B + 5x_Ax_B$$

We can interpret the result as follows: the average capture is 40 insects, where the effect due to tools is 20 insects, the effect due to agents is 10 insects, and the interaction between agents and tools accounts for 5 insects.

$$Total\ variation\ of\ y = SST = \sum_{i=1}^{2^2}(y_i - \bar{y})^2$$

In order to measure the importance of a factor, we need to calculate the proportion of the total variance generated by it. The first step is to calculate the total variance of response values, called **Total Sum of Squares (SST)**, where $\bar{y}$ is the average of all responses from the four experiments. Considering the $2^2 * 1$ design, we can divide the variation into three parts due to each factor and their combination:

$$SST = 2^2 q_A^2 + 2^2 q_B^2 + 2^2 q_{AB}^2$$

It is then possible to separate the equation and explain the variation of each of the factors through a fraction. For example:

$$Fraction\ of\ the\ variaion\ explained\ by\ : A = \frac{SSA}{SST} = \frac{2^2 q_A^2}{SST}$$

Using these equations, we can calculate the total variance $SST = 2100$ and then calculate each variance *SSA*, *SSSB*, and *SSAB*. Thus, we calculate that the variation attributed to the tool is 1600 (76%), the variation attributed to the agent is 400 (19%), and the variation attributed to the interaction of these factors is 100 (5%). These numbers show that the amount of insects captured is more dependent on the tool than the agent.

One problem with the 2^k factorial experimental design (when $r = 1$) is that it is impossible to estimate experimental errors. We can only quantify these errors by repeated measurements at the same factor levels. Thus, the $2^k r$ experimental design (with $r > 1$) can help classify the effects of the levels and quantify the experimental errors with a small number of experiments. For this project, we used not just one value for each factor/level combination but r values for each repetition. The values used by y are the average of the results of the r repetitions. Using this method, we can add a term to the model due to experimental error and measure the variation attributed to it using the following equation:

$$y = q_0 + q_A x_A + q_B x_B + q_{AB} x_A x_B + e$$

9.3 Validation

Finally, we need to validate the results of this type of model by visual tests presented by Jain, 1991. Two tests are required. The first is constructing a normal quantile-quantile plot, as shown in Figure 9.1. The purpose of this graph is to verify that the error distribution is normal. For this to be the case, the points of the diagram must have linearity, as in

graph (a). If the points do not have linearity, as in graph (b), the model requires a data transformation to make the error distribution normal.

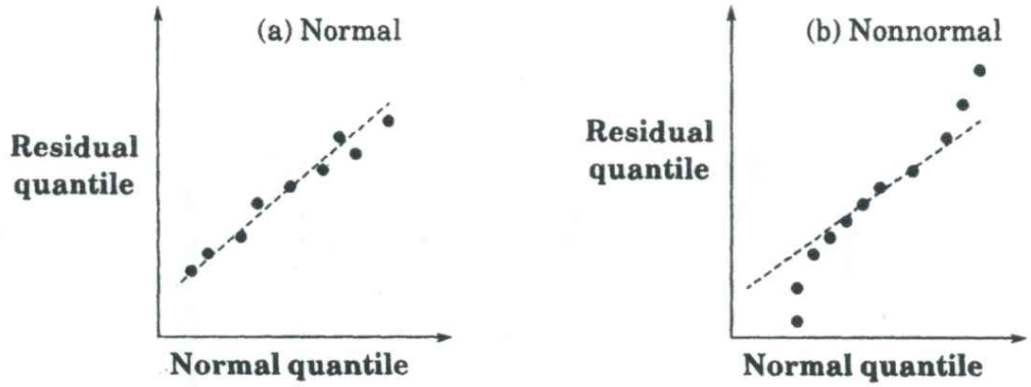


Figure 9.1: *Normal quantile-quantile plots constructed by Jain, 1991.*

The second visual test aims to check if the standard deviation of the errors is constant. To do this, we need to construct a scatter plot as shown in Figure9.2. For the standard deviation of the errors to be constant, the diagram must not show any trends, as in graph (a). In graph (b), there is a tendency for the errors to increase as the response increases. This increase means that the model does not correctly account for the effects of the factors and the error distribution is not independent. Again, the model requires a data transformation if the points do not form a graph without a trend.

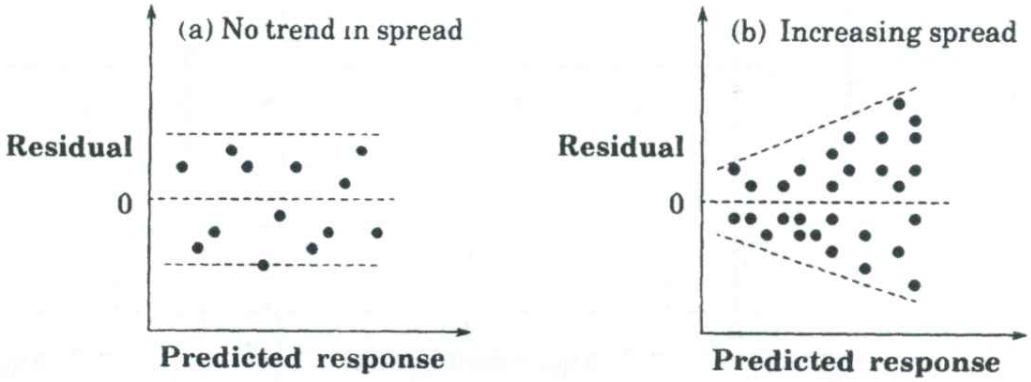


Figure 9.2: *Scatterplots constructed by Jain, 1991.*

One of the possible transformations in the data that can solve the problems of the error not being normally distributed and the standard deviation of the errors not being constant is the logarithmic transformation. To do this, we just calculate the log of the response values of the experiments. Another possible transformation is to use the square root of the results obtained. These transformations are very useful when the data values are orders of magnitude different, as is the case in some of our results.

Chapter 10

Insect capture scenario

As our interest is in scenarios where the utility of actions decreases over time, we prepared a problem with these characteristics. Our experiment scenario consists of one or more agents responsible for capturing insects in a field and transporting them to a storage container. However, to prevent the insects from getting hurt during transport, the agent will earn more points if it takes the insect faster to the container. We present the environment used in Figure 10.1. In the image, the green area in the center of the grid is the area where the insects stay. They randomly move perpendicularly and are represented by black circles. The yellow circle represents the agent. The blue squares in the corners of the grid are the containers where the agent stores the insects. The total grid size is 40x40, while the green area is 30x30. Each simulation executes 750 steps. That is, each agent can execute at most 750 actions. As the agent can only move perpendicularly, capturing an insect in the center of the grid would require it to move at least 40 times to reach a container. Using preliminary simulations, we estimated that an agent did not take more than 20 milliseconds to perform each action. Thus, we defined an upper bound of 800ms for the capture of insects. Therefore, the amount of points an agent receives per capture is 800 minus the time interval between capture and storage in milliseconds.

We performed all experiments on a MacBook Pro (16-inch, 2019) computer with a 2.6 GHz Intel Core i7 6-Core processor and 16 GB 2667 MHz DDR4 memory.

10.1 Single agent experiment

10.1.1 Bounded response time (E1)

The first experiment aims at answering the first two research questions: **Q1:** Does the AnytimeJason architecture allow us to control the time used for agent reasoning while assuring a minimum quality in the actions? **Q2:** In scenarios where the utility of actions decreases over time, does running an agent using the AnytimeJason architecture increase the utility of that agent?

For this, following the experimental design method, we must list the factors that influence the response variable (in this case, the number of points obtained by the agent). The first and most important factor is the architecture used. We want to compare the

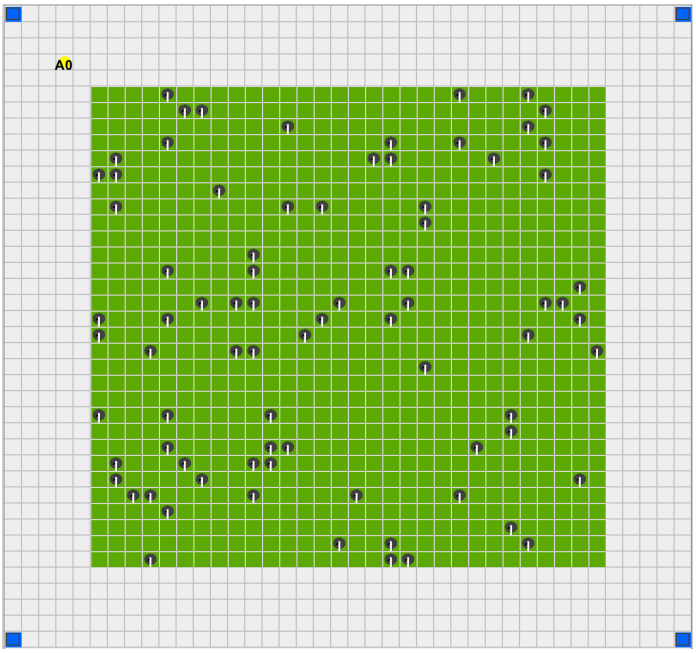


Figure 10.1: *Environment used in the experiments.*

same agent code when run by default Jason and AnytimeJason. Next, something that can make a difference is the number of insects in the environment since the amount of perceptions significantly affects the processing time Stabile Jr and Sichman, 2015b. Finally, the time-bound for the agent to execute an action can influence the score. Once the factors are defined, we have to specify the factor levels. The idea of the 2^{kr} design is that each one of the k factors has two possible levels and that each experiment runs r times. We chose the value of $r = 3$, as it is the smallest value that allows for measuring error and unobserved factors. In addition, it is interesting, when possible, that the factors are far apart, as this facilitates the perception of factor variation. So, when we choose levels, a pair of 100 and 900 insect levels is better than a pair of 400 and 500 insects for our scenario. Thus, we set the insect quantity levels to 90 and 810 insects. These values correspond to 10% and 90% of the available space. For the execution time threshold, we ran a simulation where the default Jason agent executed actions with an average time of approximately 10ms, with the 25th percentile of 7ms and the 75th percentile of 12ms. We define then two bounds for action execution. If the agent takes more than 7ms (or 12ms in the second case) to execute an action, this action will fail.

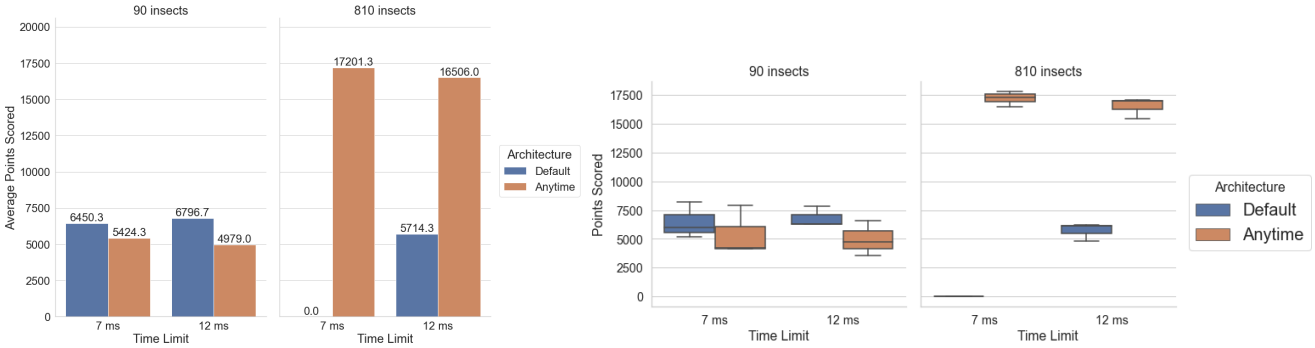
We show the factors, levels and scores obtained in Table 10.1. For ease of visualization, we display the average of points scored in the executions in Figure 10.2.

From the graph in Figure 10.2, we can see that the Anytime architecture does better in the scenario with 810 insects, and the default architecture does better with 90 insects. However, it is not enough to know that values have changed. Thus, we have to be able to show what causes the difference. For this, we calculated the influence of the factors according to the experimental design methodology. We present these influences in Figure 10.2.

Based on the calculated variation values, we can see in this experiment that the

Insects	Time-bound	Architecture	Points scored		
			r1	r2	r3
90 insects	7 ms	Default	5975	8205	5171
90 insects	7 ms	Anytime	4201	7905	4167
90 insects	12 ms	Default	6256	6314	7820
90 insects	12 ms	Anytime	6600	3592	4745
810 insects	7 ms	Default	0	0	0
810 insects	7 ms	Anytime	17833	17265	16506
810 insects	12 ms	Default	6149	4807	6187
810 insects	12 ms	Anytime	17023	17037	15458

Table 10.1: *Points scored by the agents.*



(a) *Average points scored in each level.*

(b) *Box plot of points scored in each level.*

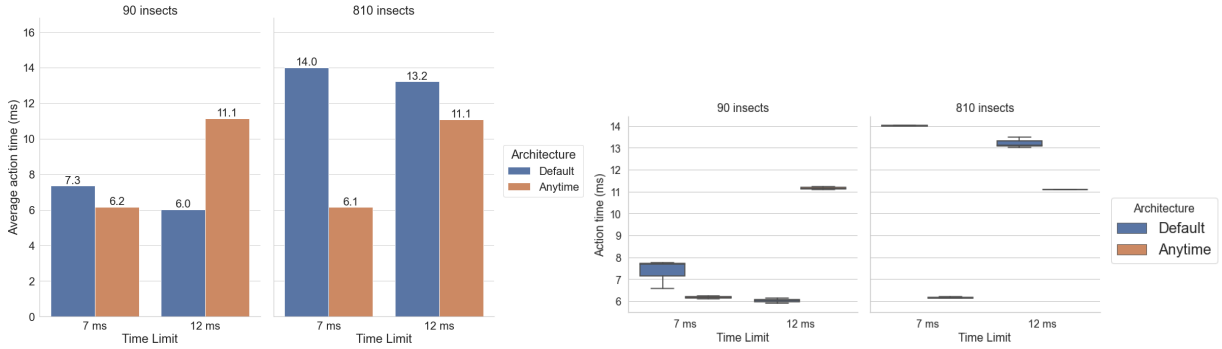
Figure 10.2: *Points scored in the single-agent bounded response time experiment.*

Factors	Influence
Variation of Insect number:	12.25%
Variation of Time-bound:	1.19%
Variation of Architecture:	31.15%
Variation of interaction of Insect number & Time-bound:	1.29%
Variation of interaction of Insect number & Architecture:	46.84%
Variation of interaction of Time-bound & Architecture:	2.55%
Variation of interaction of Insect number & Time-bound & Architecture:	1.55%
Variation of error and unobserved factors:	3.17%

Table 10.2: *Variation attributed to each factor.*

influence of the time-bound on the number of points is small. However, the number of insects and the architecture influenced the points scored. Furthermore, primarily, there is an influence of the interaction between the number of insects and the architecture. Furthermore, the variation attributed to measurement errors and other factors is negligible. Thus, we can verify that there is a scenario where the AnytimeJason architecture increases the agent's utility value.

Next, we must evaluate if the architecture controls the agent's execution time properly. To do so, we can analyze the graph in Figure 10.3. It shows the average execution time of actions for the agent at the selected levels. For this experiment, the agent needs to act in the environment up to 7ms and up to 12ms. We then defined that the anytime agent should respond in 6ms and 11ms, as it is necessary to consider the agent's communication time with the simulator. With an average time of approximately 6ms and 11ms and an average standard deviation of 1ms, we demonstrated that we achieved one of the research objectives: To build an architecture capable of controlling the agent's execution time. Also, this graph allows us to understand why the default Jason does better with 90 insects. Once we define that the agent must act at a fixed time, it will continue to process, even if it has encountered an external action. The standard agent Jason, on the other hand, may be able to act faster if there is little to process.



(a) *Average response time in each level.* **(b)** *Box plot of response time in each level.*

Figure 10.3: *Response times for the single-agent bounded response time experiment.*

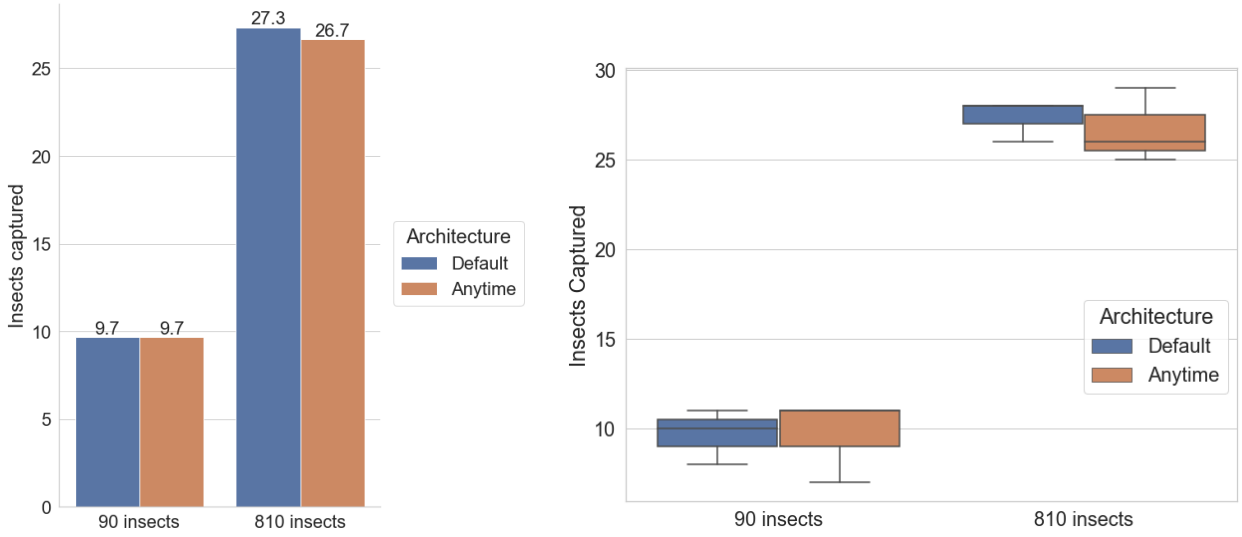
10.1.2 Unbounded response time (E2)

This second experiment aims at answering the last research question: **Q3:** Does the AnytimeJason architecture allow running an agent with a shorter processing time than the default Jason architecture, minimizing the loss of utility resulting from this reduction? For this experiment, we used the same scenario as in Experiment 10.1.1. However, we want to evaluate a scenario with no penalty due to execution time. Thus, only the number of insects and the architecture used will be factors. Also, instead of calculating time-based scores, we only consider the number of insects captured. We display the captures in the executions in Figure 10.4 and Table 10.3. Also, as we are interested in the execution time, we present the average action time in Figure 10.5 and Table 10.4.

When analyzing the result of this experiment, we noticed a 30% reduction in processing time using the anytime architecture compared to the same agent using the default architecture. At the same time, there is no impact on the number of insects agents can

Insects	Architecture	Insects captured		
		r1	r2	r3
90 insects	Default	11	8	10
90 insects	Anytime	7	11	11
810 insects	Default	28	28	26
810 insects	Anytime	25	26	29

Table 10.3: *Insects captured by the agents.*

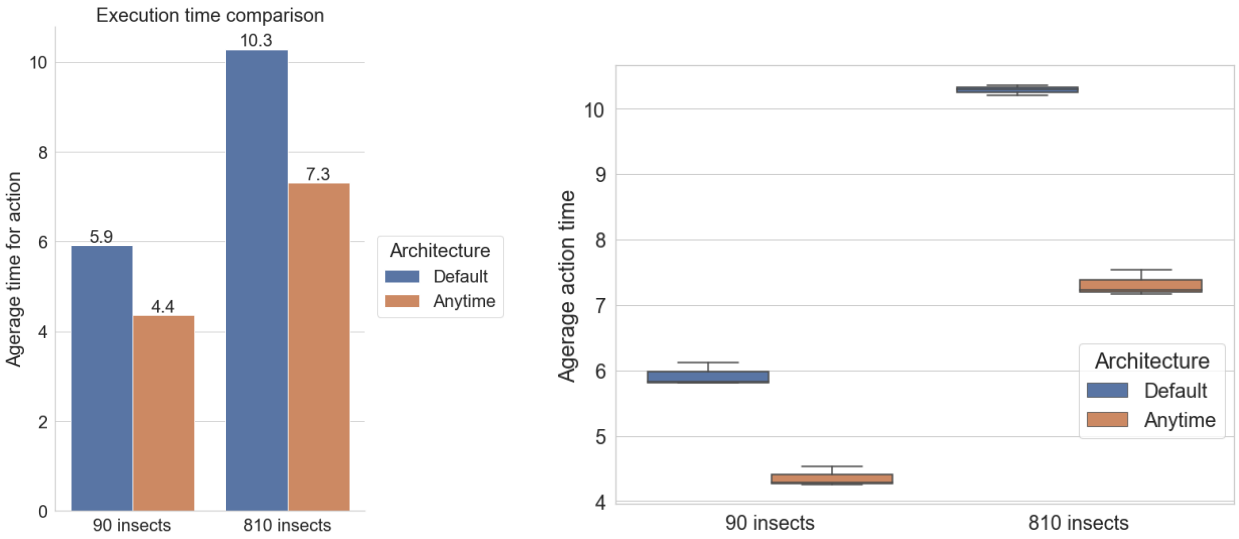


(a) *Average insects captured in each level.* **(b)** *Box plot of insects captured in each level.*

Figure 10.4: *Insects captured in the single-agent unbounded response time experiment.*

Insects	Architecture	Response time		
		r1	r2	r3
90 insects	Default	5.80	6.12	5.82
90 insects	Anytime	4.28	4.53	4.26
810 insects	Default	10.20	10.29	10.35
810 insects	Anytime	7.23	7.53	7.17

Table 10.4: *Average response time by the agents.*



(a) *Average response time in each level.* **(b)** *Box plot of response time in each level.*

Figure 10.5: *Response time in the single-agent unbounded response time experiment.*

capture. It remains for us then to use the validations available in the experimental design to attribute these variations to the factors. Table 10.5 shows that the architecture and its interaction with the number of insects are responsible for only 0.08% of the variation. That is, the architecture change does not sufficiently affect the number of captures. Table 10.6 shows that the sum of variations in processing time attributed to architecture and its interaction is 29.47%. Thus, we demonstrate that in this scenario, the architecture impacted the agent's processing time reduction. At the same time, there was no reduction in the agent's ability to fulfill its objective of capturing insects.

Factors	**Influence**
Variation of Insect number:	97.06%
Variation of Architecture:	0.04%
Variation of interaction of Insect number & Architecture:	0.04%
Variation of error and unobserved factors:	2.87%

Table 10.5: *Variation attributed to each factor regarding captures.*

Factors	**Influence**
Variation of Insect number:	70.19%
Variation of Architecture:	26.86%
Variation of interaction of Insect number & Architecture:	2.61%
Variation of error and unobserved factors:	0.34%

Table 10.6: *Variation attributed to each factor regarding response time.*

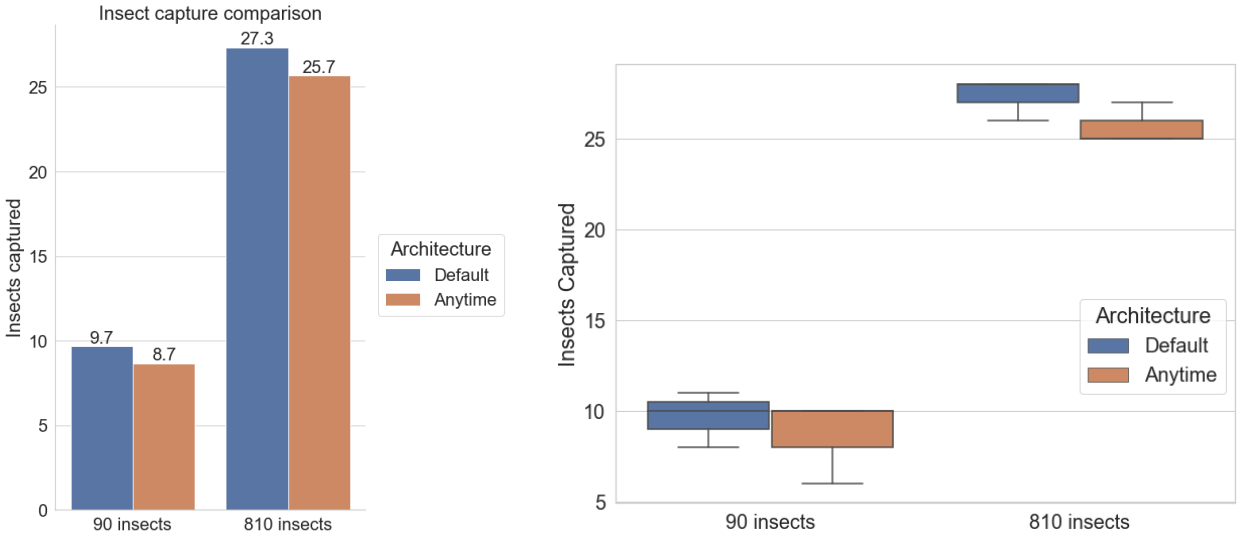
10.1.2.1 Reducing utility by 10%

In order to explore the limits of the architecture, we sought to identify what percentage reduction in processing time would be possible if we accepted a reduction of up to 10% in

the number of insects captured. Table 10.7, Figure 10.6, Table 10.8, Figure 10.7 show the results obtained in this experiment.

Insects	Architecture	Insects captured		
		r1	r2	r3
90 insects	Default	11	8	10
90 insects	Anytime	10	10	6
810 insects	Default	28	28	26
810 insects	Anytime	25	27	25

Table 10.7: *Insects captured by the agents.*



(a) *Average insects captured in each level.* **(b)** *Box plot of insects captured in each level.*

Figure 10.6: *Insects captured in the second single-agent unbounded response time experiment.*

Based on the results presented and on the variation assignments contained in Tables 10.9 and 10.10, we demonstrate that in this scenario, the agent could perform actions 50% faster when using the anytime architecture with a reduction of up to 10% in the number of insects captured.

10.2 Multi-agent experiment

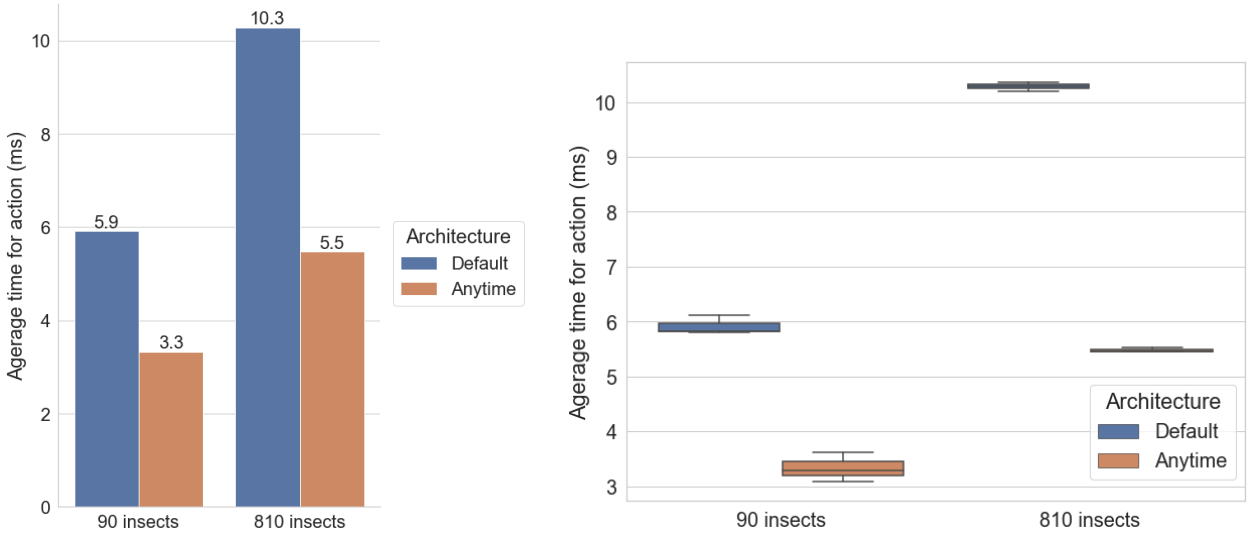
10.2.1 Bounded response time (E3)

In this experiment, our interest is to verify if the results obtained in Experiment 10.1.1 remain in a multi-agent scenario. For this, we added a new factor, the number of agents. The selected levels were 2 and 4 agents. Also, we increased the time-bound to 8ms and 16ms to reflect the multi-agent 25 percentile and 75 percentile. Agents do not coordinate their actions, and the score accumulates for all agents in the environment. Table 10.11, Figure 10.8, and Figure 10.9 shows the points scored in the experiment.

For this scenario, the architecture and its interactions account for more than 50% of the agents' score variation. This variation is a strong indicator that the architecture

Insects	Architecture	Response time		
		r1	r2	r3
90 insects	Default	5.80	6.12	5.82
90 insects	Anytime	3.61	3.28	3.09
810 insects	Default	10.20	10.29	10.35
810 insects	Anytime	5.45	5.52	5.44

Table 10.8: *Average response time by the agents.*



(a) *Average response time in each level.* **(b)** *Box plot of response time in each level.*

Figure 10.7: *Response time in the second single-agent unbounded response time experiment.*

Factors	Influence
Variation of Insect number:	97.16%
Variation of Architecture:	0.57 %
Variation of interaction of Insect number & Architecture:	0.04%
Variation of error and unobserved factors:	2.23%

Table 10.9: *Variation attributed to each factor regarding captures.*

Factors	Influence
Variation of Insect number:	41.45%
Variation of Architecture:	53.46%
Variation of interaction of Insect number & Architecture:	4.80%
Variation of error and unobserved factors:	0.28%

Table 10.10: *Variation attributed to each factor regarding response time.*

Insects	Time-bound	Agents	Architecture	Points scored		
				r1	r2	r3
90 insects	8 ms	2	Default	5090	1953	722
810 insects	8 ms	2	Default	0	0	0
90 insects	16 ms	2	Default	11404	4714	7073
810 insects	16 ms	2	Default	20277	16692	24821
90 insects	8 ms	4	Default	0	0	0
810 insects	8 ms	4	Default	0	0	0
90 insects	16 ms	4	Default	16438	13043	15392
810 insects	16 ms	4	Default	22089	16628	13307
90 insects	8 ms	2	Anytime	7077	8058	7681
810 insects	8 ms	2	Anytime	36640	36753	36726
90 insects	16 ms	2	Anytime	7341	3929	8581
810 insects	16 ms	2	Anytime	32552	34021	34021
90 insects	8 ms	4	Anytime	16851	10938	11021
810 insects	8 ms	4	Anytime	54499	62679	60071
90 insects	16 ms	4	Anytime	12575	10143	6564
810 insects	16 ms	4	Anytime	58697	56568	53127

Table 10.11: *Points scored by the agents.*

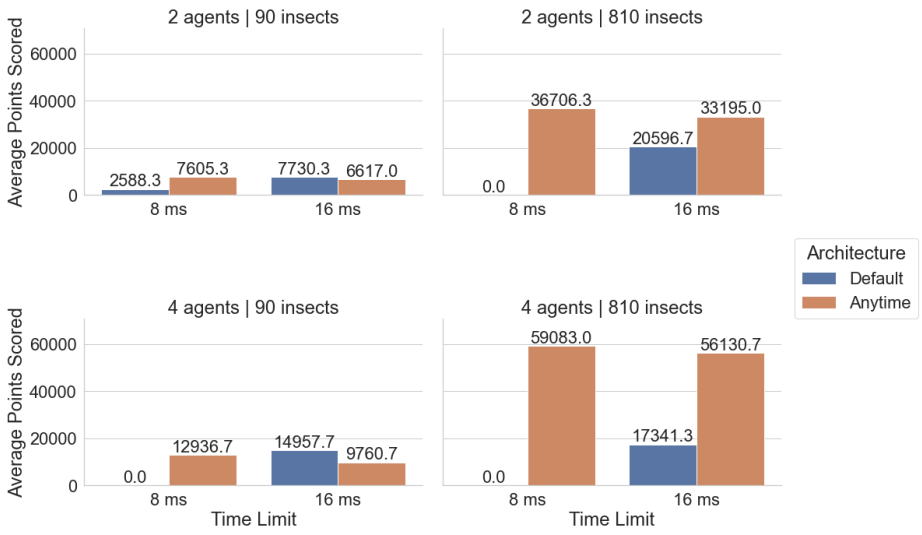


Figure 10.8: *Average points scored in the multi-agent bounded response time experiment.*

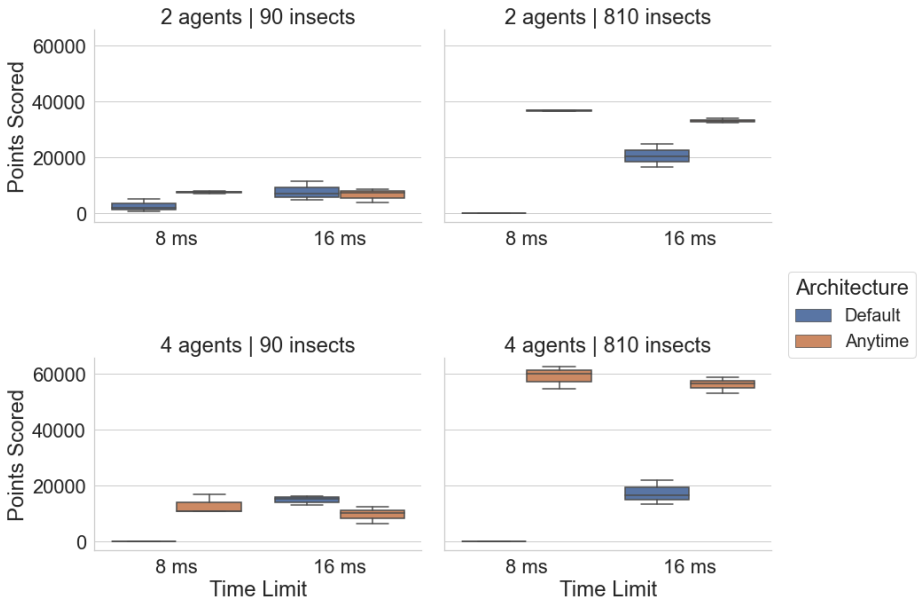


Figure 10.9: *Box plot of points scored in the multi-agent bounded response time experiment.*

Factors	**Influence**
Variation of Insect number:	29.64%
Variation of Time-bound:	2.57%
Variation of Agent Number:	3.49%
Variation of Architecture:	28.89%
Variation of Insect number & Time-bound:	0.28%
Variation of Insect number & Agent Number:	0.96%
Variation of Insect number & Architecture:	21.04%
Variation of Time-bound & Agent Number:	0.03%
Variation of Time-bound & Architecture:	5.40%
Variation of Agent Number & Architecture:	3.15%
Variation of Insect number & Time-bound & Agent Number:	0.12%
Variation of Insect number & Time-bound & Architecture:	0.46%
Variation of Insect number & Agent Number & Architecture:	2.29%
Variation of Time-bound & Agent Number & Architecture:	0.08%
Variation of all factors:	0.29%
Variation of error and unobserved factors:	1.33%

Table 10.12: *Variation attributed to each factor.*

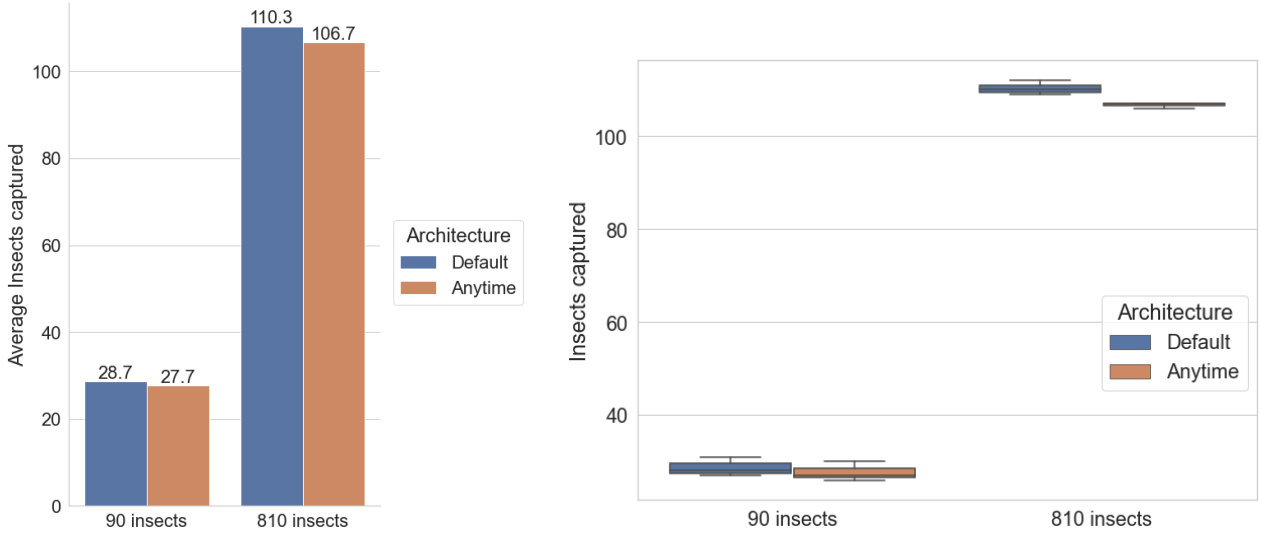
can increase the agent's utility in scenarios where the utility of actions decreases over time, as expected. Another interesting point arises when we analyze Table 10.11. It is possible to see that in some scenarios where the agent's response time is limited to 8ms, the default agent could not score any points. This absence of score shows that the limit is very restrictive for these agents, who cannot process in time to perform actions. In comparison, the anytime agent can act in these scenarios and score points because of its ability to control its execution time.

10.2.2 Unbounded response time (E4)

In this experiment, our interest is to verify if the results obtained in Experiment 10.1.2 remain in a multi-agent scenario. Nevertheless, since the variation attributed to the number of agents in experiment E3 was minimal (as shown in Table 10.12), we always used four agents for this experiment. Table 10.13 and Figure 10.10 shows the points scored in the experiment. Table 10.14 and Figure 10.11 shows the average response time of the agents. Finally, Tables 10.15 and 10.16 show the variances attributed to each factor regarding the points and the response time respectively.

Insects	Architecture	Insects captured		
		r1	r2	r3
90 insects	Default	31	27	28
90 insects	Anytime	30	26	27
810 insects	Default	110	109	112
810 insects	Anytime	107	106	107

Table 10.13: *Insects captured by the agents.*



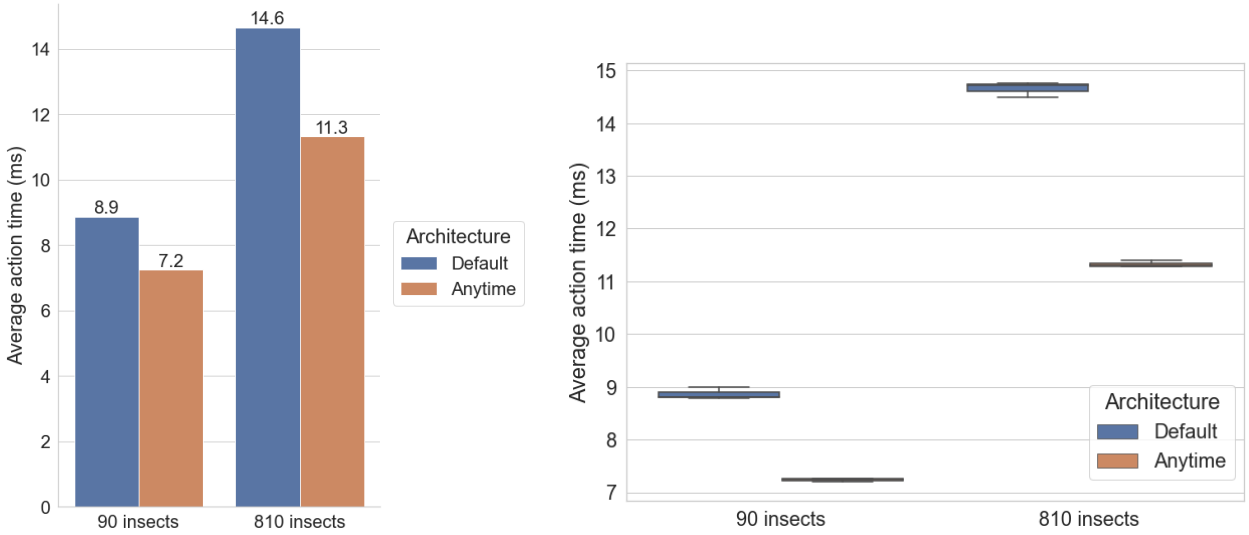
(a) *Average insects captured in each level.* **(b)** *Box plot of insects captured in each level.*

Figure 10.10: *Insects captured in the multi-agent unbounded response time experiment.*

As can be seen in the Figures 10.10 and 10.11 and in Tables 10.15 and 10.16, as in experiment E2, it was possible to reduce agent processing time without reducing the

Insects	Architecture	Response time		
		r1	r2	r3
90 insects	Default	8.80	8.79	8.98
90 insects	Anytime	7.21	7.26	7.24
810 insects	Default	14.71	14.75	14.48
810 insects	Anytime	11.29	11.27	11.39

Table 10.14: *Average response time by the agents.*



(a) *Average response time in each level.* **(b)** *Box plot of response time in each level.*

Figure 10.11: *Response time in the multi-agent unbounded response time experiment.*

Factors	Influence
Variation of Insect number:	99.77%
Variation of Architecture:	0.08%
Variation of interaction of Insect number & Architecture:	0.03%
Variation of error and unobserved factors:	0.12%

Table 10.15: *Variation attributed to each factor regarding captures.*

Factors	Influence
Variation of Insect number:	78.00%
Variation of Architecture:	19.59%
Variation of interaction of Insect number & Architecture:	2.33%
Variation of error and unobserved factors:	0.08%

Table 10.16: *Variation attributed to each factor regarding response time.*

number of insects captured. However, only an approximately 20% reduction was possible this time.

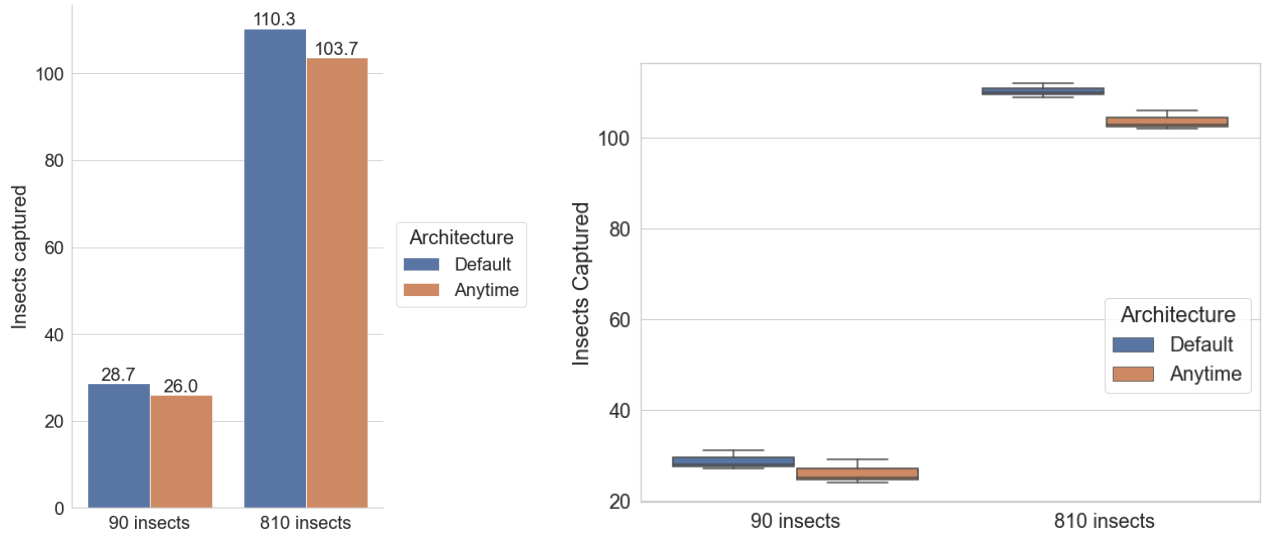
10.2.2.1 Reducing the processing time by 30%

As in the E2 experiment, we want to show that achieving an even higher reduction is possible when we accept a slight reduction in the number of captures. Table 10.17 and Figure 10.12 shows the amount of insects captured by the agents. Table 10.18 and Figure 10.13 shows the average response time of the agents. Finally, Tables 10.19 and 10.20 show the variances attributed to each factor regarding the points and the response time respectively.

Unlike the result obtained in the E2 experiment, where there was no reduction in the number of captures when we reduced the agent's response time by 30%, in the multi-agent scenario, the same reduction caused an average decrease of about 10% in the captures performed by the agents.

Insects	Architecture	Insects captured		
		r1	r2	r3
90 insects	Default	31	27	28
90 insects	Anytime	25	24	29
810 insects	Default	110	109	112
810 insects	Anytime	103	106	102

Table 10.17: *Insects captured by the agents.*



(a) *Average insects captured in each level.* **(b)** *Box plot of insects captured in each level.*

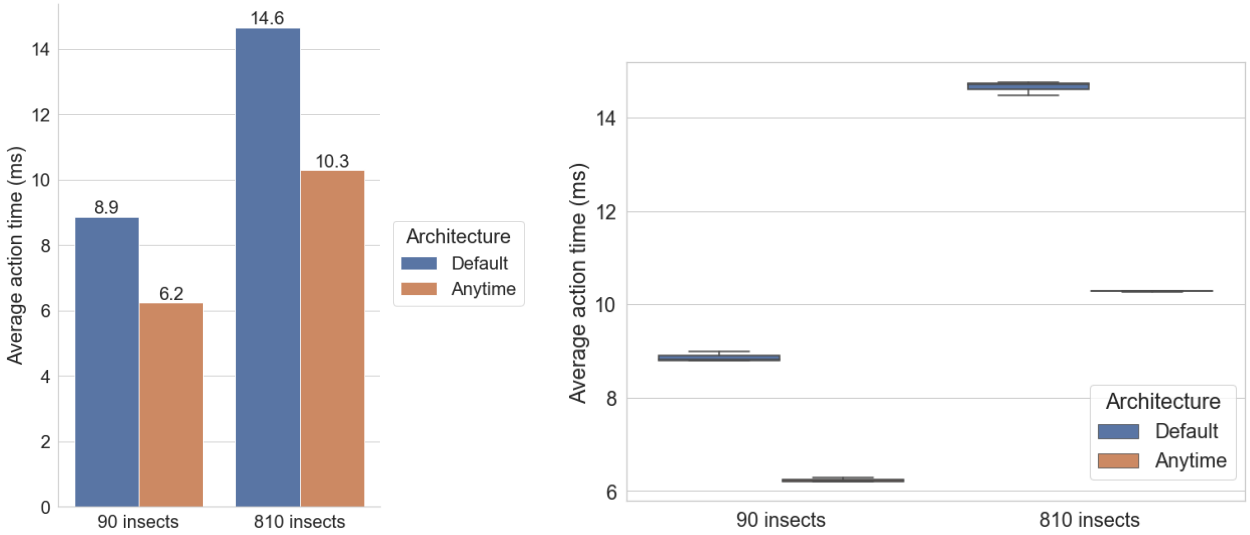
Figure 10.12: *Insects captured in the multi-agent unbounded response time experiment.*

10.2.2.2 Reducing the processing time by 50%

For a final comparison, we analyzed agents' performance when we reduced the average response time by 50%. Table 10.21 and Figure 10.14 shows the amount of insects captured

Insects	Architecture	Response time		
		r1	r2	r3
90 insects	Default	8.80	8.79	8.98
90 insects	Anytime	6.20	6.28	6.21
810 insects	Default	14.71	14.75	14.48
810 insects	Anytime	10.26	10.29	10.29

Table 10.18: *Average response time by the agents.*



(a) *Average response time in each level.* **(b)** *Box plot of response time in each level.*

Figure 10.13: *Response time in the multi-agent unbounded response time experiment.*

Factors	Influence
Variation of Insect number:	99.41%
Variation of Architecture:	0.34%
Variation of interaction of Insect number & Architecture:	0.06%
Variation of error and unobserved factors:	0.19%

Table 10.19: *Variation attributed to each factor regarding captures.*

Factors	Influence
Variation of Insect number:	65.02%
Variation of Architecture:	32.88%
Variation of interaction of Insect number & Architecture:	2.04%
Variation of error and unobserved factors:	0.06%

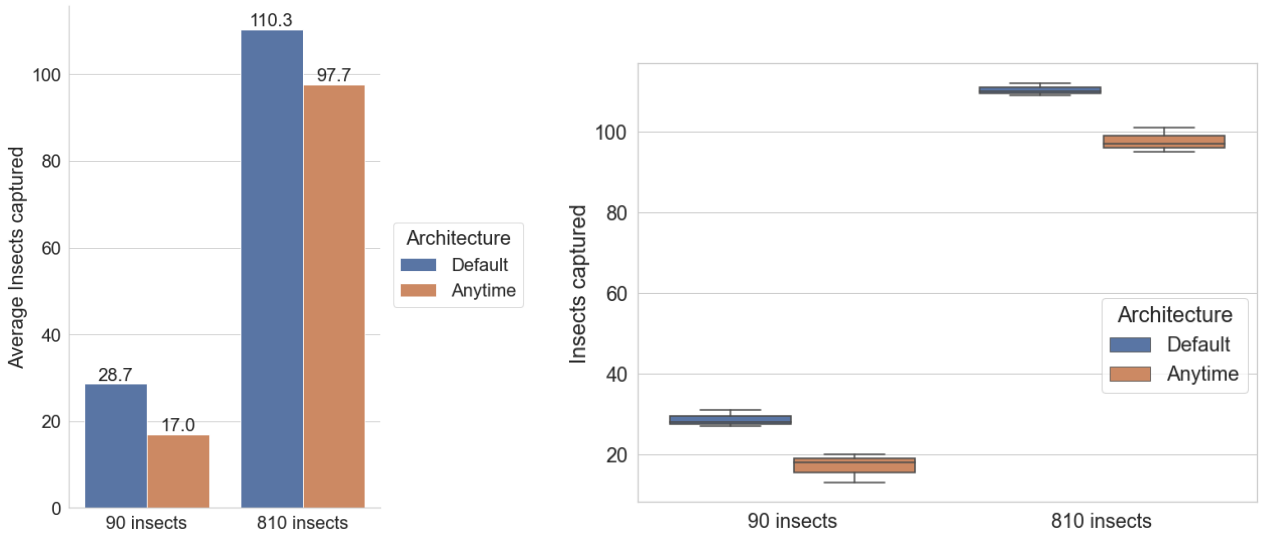
Table 10.20: *Variation attributed to each factor regarding response time.*

by the agents. Table 10.22 and Figure 10.15 shows the average response time of the agents. Finally, Tables 10.23 and 10.24 show the variances attributed to each factor regarding the points and the response time respectively.

Analyzing the results, we notice that this experiment had the highest drop in the number of captures. For example, in the environment with 90 insects, there was a decrease of approximately 40% in the average of captures. The drop was smaller in the environment with 810 insects, about 12%. This result suggests that a bound of 4 milliseconds is not enough for the agent to be able to process the reasoning cycle.

Insects	Architecture	Insects captured		
		r1	r2	r3
90 insects	Default	31	27	28
90 insects	Anytime	18	20	13
810 insects	Default	110	109	112
810 insects	Anytime	97	101	95

Table 10.21: *Insects captured by the agents.*

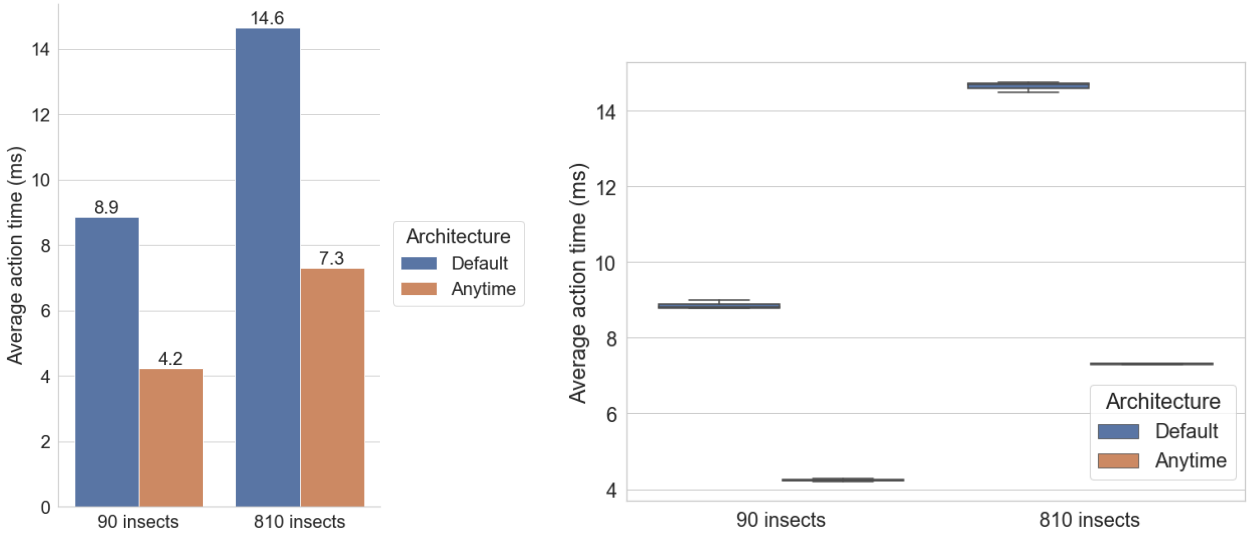


(a) *Average insects captured in each level.* **(b)** *Box plot of insects captured in each level.*

Figure 10.14: *Insects captured in the multi-agent unbounded response time experiment.*

Insects	Architecture	Response time		
		r1	r2	r3
90 insects	Default	8.80	8.79	8.98
90 insects	Anytime	4.27	4.23	4.21
810 insects	Default	14.71	14.75	14.48
810 insects	Anytime	7.29	7.31	7.31

Table 10.22: *Average response time by the agents.*



(a) *Average response time in each level.* **(b)** *Box plot of response time in each level.*

Figure 10.15: *Response time in the multi-agent unbounded response time experiment.*

Factors	Influence
Variation of Insect number:	97.52%
Variation of Architecture:	2.19%
Variation of interaction of Insect number & Architecture:	0.00%
Variation of error and unobserved factors:	0.29%

Table 10.23: *Variation attributed to each factor regarding captures.*

Factors	Influence
Variation of Insect number:	34.25%
Variation of Architecture:	62.48%
Variation of interaction of Insect number & Architecture:	3.23%
Variation of error and unobserved factors:	0.04%

Table 10.24: *Variation attributed to each factor regarding response time.*

Chapter 11

Multi-agent Programming Contest scenario

The "Multi-agent Programming Contest"[1] aims to stimulate research in multi-agent system development and programming. This stimulus is achieved by identifying key problems, collecting suitable benchmarks, and gathering test cases that require and enforce coordinated action that can serve as milestones for testing multi-agent programming languages, platforms, and tools. The organizers also expect that participating in the contest helps to debug existing systems and to identify their weak and strong aspects.

The 2020/21 contest presented by Ahlbrecht *et al.* (2021) used the Agents Assemble scenario, which consists of two teams of agents moving on a grid to explore the world and acquire blocks to assemble them into complex patterns. Agents can attach things to themselves. These attached things move or rotate when the agents move or rotate while attached. In addition, two agents can connect things attached to them to create more complex structures. The environment is a rectangular grid whose dimensions are unknown to the agents. Agents only perceive positions relative to their own within a limited distance.

The environment randomly creates tasks with a deadline, a set of blocks in a given assembly, and an award value that decreases over time. When a team correctly submits the task, the team receives points equal to the task's current award. To submit a task, an agent must move to a taskboard, act to accept the task, coordinate with other agents to assemble the correct blocks correctly, move to a goal area of the map, and perform a submission action. In addition, agents also have to deal with the fact that the other team tries to submit the task beforehand, and random events disable the agent for a few steps. At the end of 750 steps, the team with the most points wins the game. In the competition, multiple teams face each other on different maps, and the team with the most games wins is the champion.

As the contest aims to help debug existing systems and identify their weak and strong aspects, we will use the contest environment for this purpose. The MAPC environment

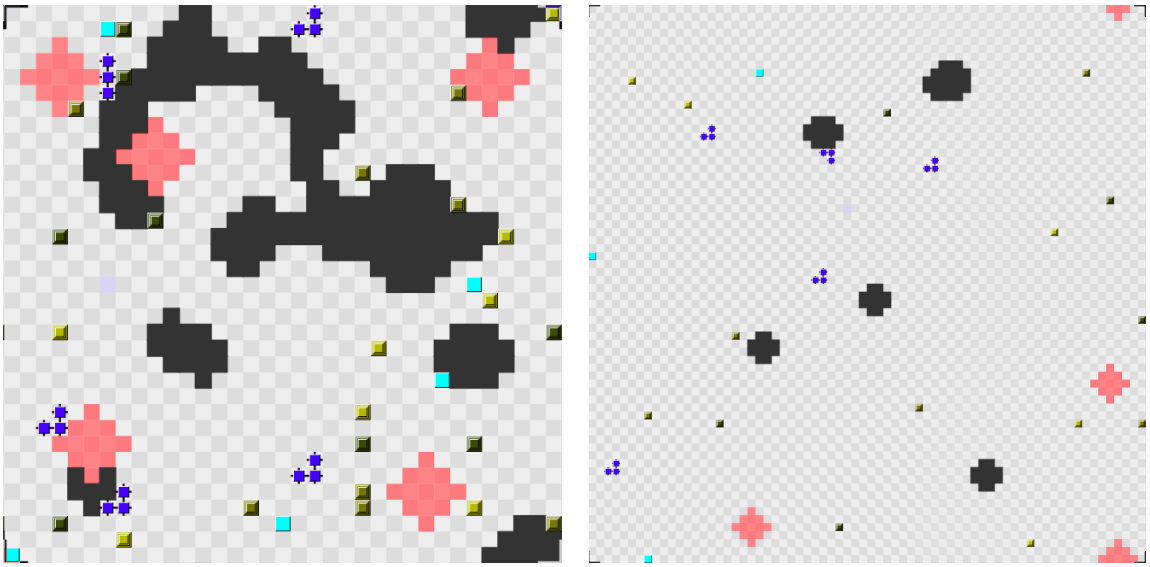
[1] MAPC: https://www.multiagentcontest.org/

presents a much more complex and demanding scenario than the insect capture scenario. However, it is much less customizable. As a result, it is impossible, for example, to analyze the performance of a single agent in the MAPC scenario since coordination between agents is a critical factor.

In the experiments presented in this chapter, we used the same code as the agents who participated in the 2020/21 contest by the "LTI-USP" team, awarded fourth place, and presented in Stabile Jr. and Sichman, 2021.

The map used has a significant impact on the score obtained by the agents because depending on the size of the map and the position and number of elements, the agents will perceive more or fewer elements, which also influences the number of messages exchanged between the agents. Finally, these variations directly impact agent processing time. Because of this, we proposed the two scenarios shown in Figure 11.1 for our experiments. We call the scenario in Figure 11.1a clustered, as it has a smaller size and a larger number of elements, such as taskboards, dispensers, and goal areas. These characteristics make the processing of agents usually take longer. Also, we named the scenario in Figure 11.1b sparse, as it is larger and contains fewer elements. Because of these differences, in this chapter, these two environments will always be levels of one of the factors.

In the following experiments, we used a MacBook Pro (16-inch, 2019) computer to run the agents with a 2.6 GHz Intel Core i7 6-Core processor and 16 GB 2667 MHz DDR4 memory. We used a MacBook Air (13-inch, Early 2014) for the server with a 1.4 GHz Intel Core i5 Dual-Core processor and 4 GB 1600 MHz DDR3 memory; the two computers were communicating on the same LAN via Wi-fi.



(a) *Clustered environment map*

(b) *Sparse environment map.*

Figure 11.1: *Maps used in the MAPC experiments.*

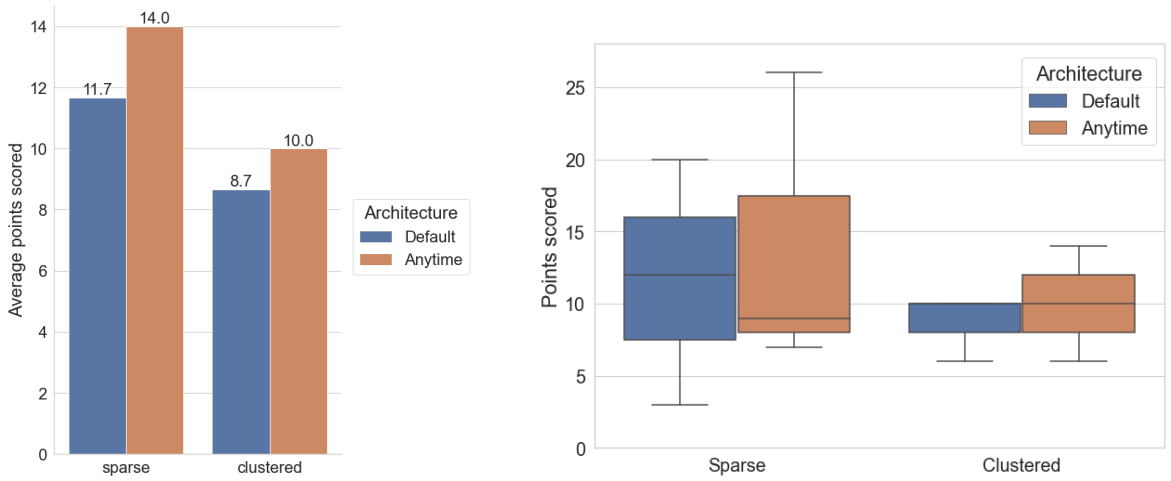
11.1 Default competition response time (E5)

In this first experiment, we used the settings closest to those used in the contest. That is, the interval between sending the perceptions by the server and the bound for executing the actions is four seconds; there is a 1% chance that each action fails randomly; each team contains 15 agents; the simulation takes 750 steps. The two factors in the experiment are environment (clustered and sparse) and architecture (default and Anytime).

For the contest, two characteristics of agents are essential. One is that agents need to be able to score as many points as possible (because that is how you win). The second is that agents must be able to perform their actions in the environment consistently. That is because agents who take too long and miss too many actions are eliminated in the qualifying phase and cannot participate in the contest. Therefore, we adopted two response variables—the team score and the number of actions lost due to timeout. The two factors in the experiment are environment (clustered and sparse) and architecture (default and Anytime).

Map	Architecture	Points scored		
		r1	r2	r3
Sparse	Default	20	3	12
Sparse	Anytime	9	7	26
Clustered	Default	10	6	10
Clustered	Anytime	6	10	14

Table 11.1: *Points scored by the agents in the default MAPC experiment.*



(a) *Average points scored in each level.*

(b) *Box plot of points scored in each level.*

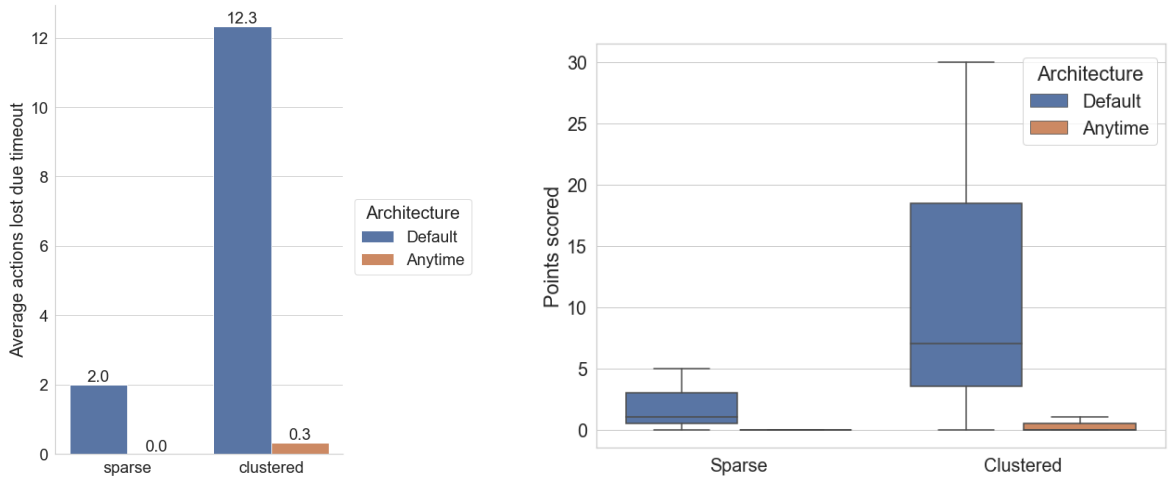
Figure 11.2: *Points scored in the default MAPC experiment.*

We start by analyzing the score. The graphs in Figure 11.2 visually present the results presented in Table 11.1. A simple analysis of the presented graphs can lead us to think that the Anytime architecture caused an increase in the agents' scores. However, when we look at the variation analysis in Table 11.2, we can see that almost all the variation is neither caused by the environment nor the architecture. That is, we cannot credit this

variation to the change in architecture. This result becomes apparent when we look at Figure 11.3 and Table 11.3, which show the number of actions lost by agents. There are 15 agents, and each can perform up to 750 actions in a match. That is a total of 11250 actions. In the execution where there was a more noteworty loss of actions, the agent in the default architecture lost 30 actions due to timeout. That is a total of approximately 0.2% of the total actions. Considering that 1% of the actions fail by a characteristic of the competition scenario, we have that about 112 actions fail randomly. Thus, variance analysis shows us that this difference in score has more to do with the randomness of the environment than with the change in architecture.

Factors	Influence
Variation of Map:	8.11%
Variation of Architecture:	2.23%
Variation of interaction of Map & Architecture:	0.17%
Variation of error and unobserved factors:	89.49%

Table 11.2: *Variation attributed to each factor.*



(a) *Average actions lost in each level.*

(b) *Box plot of actions lost in each level.*

Figure 11.3: *Actions lost in the default MAPC experiment.*

Map	Architecture	Actions lost		
		r1	r2	r3
Sparse	Default	5	1	0
Sparse	Anytime	0	0	0
Clustered	Default	7	0	30
Clustered	Anytime	0	0	1

Table 11.3: *Actions lost by the agents due to timeout in the default MAPC experiment.*

However, this finding shows that our architecture could run the agents with the same quality as the standard architecture. Moreover, the agents used in MAPC are much more complex than those that capture insects. They also use message mechanisms that are not

in the theoretical model. This result indicates that the model is robust enough to accept modifications resulting from the specific functioning of each BDI agent programming language.

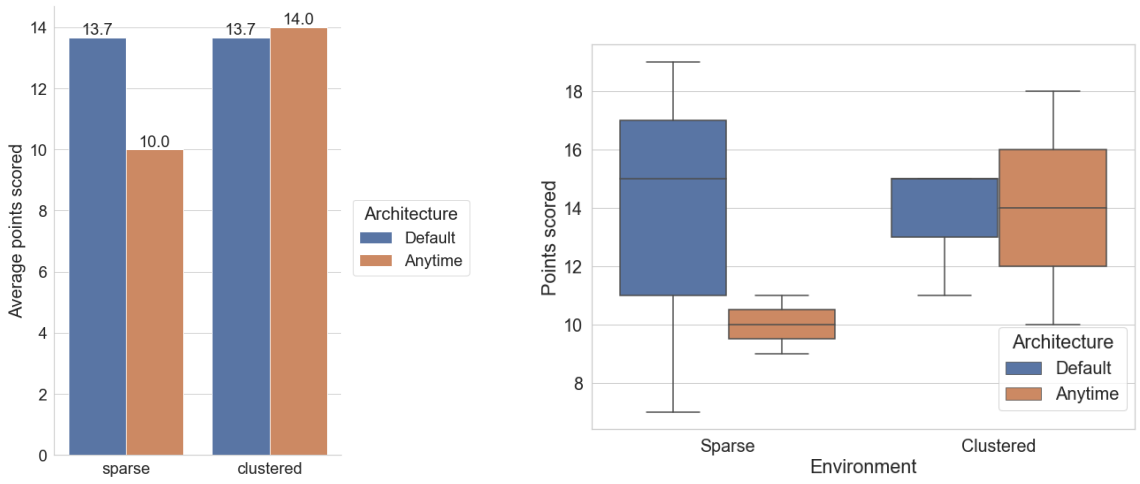
11.2 Reduced response time (E6)

In the analysis of the previous experiment, we demonstrated that the agent was sufficiently optimized so that it did not miss many actions during the simulation. Therefore, in this second experiment, we want to evaluate how a reduction in the simulator's maximum response time impacts the score of each of the evaluated architectures. So, we reduced the maximum response time from 4 seconds to 1 second to achieve this objective.

Based on the values presented in Table 11.6, we can see an increase in the number of actions lost by timeout. In this experiment, the default architecture lost, on average, 2.5% of the actions sent to the simulator. Evaluating Table 11.5, we see that the variation attributed to the architecture increased from 2.23% to 5.49%, and the variation attributed to the interaction between the two factors increased from 0.17% to 7.91%. Despite not being a very high value, this variation increase demonstrates that the anytime architecture starts to influence the agent's score as the number of actions lost by the default architecture increases.

Map	Architecture	Points scored		
		r1	r2	r3
Sparse	Default	7	15	19
Sparse	Anytime	10	11	9
Clustered	Default	11	15	15
Clustered	Anytime	14	18	10

Table 11.4: *Points scored by the agents in the reduced response time MAPC experiment.*

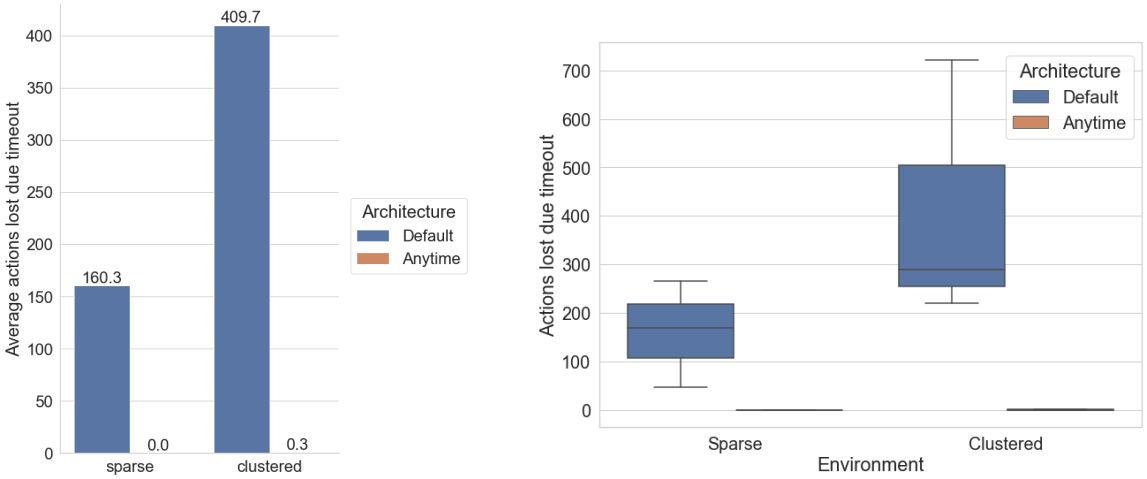


(a) *Average points scored in each level.* **(b)** *Box plot of points scored in each level.*

Figure 11.4: *Points scored in the reduced response time MAPC experiment.*

Factors	Influence
Variation of Map:	7.91 %
Variation of Architecture:	5.49%
Variation of interaction of Map & Architecture:	7.91%
Variation of error and unobserved factors:	78.68%

Table 11.5: *Variation attributed to each factor.*



(a) *Average actions lost in each level.*

(b) *Box plot of actions lost in each level.*

Figure 11.5: *Actions lost in the reduced response time MAPC experiment.*

Map	Architecture	Actions lost		
		r1	r2	r3
Sparse	Default	266	169	46
Sparse	Anytime	0	0	0
Clustered	Default	219	721	289
Clustered	Anytime	1	0	0

Table 11.6: *Actions lost by the agents due to timeout in the reduced response time MAPC experiment.*

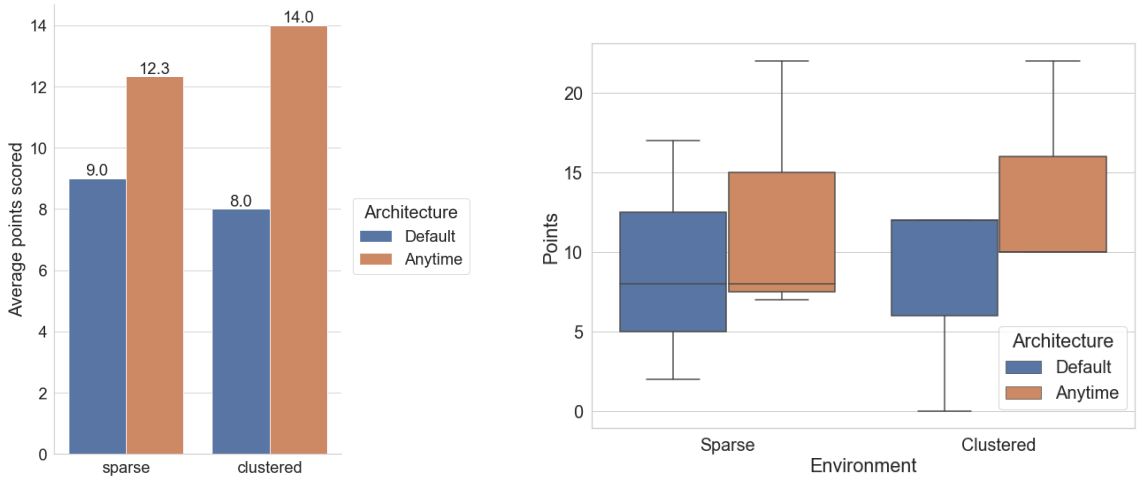
11.3 Reduced response time without random failure (E7)

In experiment E6 presented in Section 11.2, we observed that, on average, 2.5% of the actions of the agents executed by the default architecture were lost by timeout. However, agents running through the anytime architecture also miss actions because of random failures caused by the environment. For this reason, we propose a new experiment with a time-bound of 1 second and where there are no random failures to analyze this change's impact.

As we can see from the data presented in Table 11.9, there was a difference in the number of actions lost per timeout compared to experiment E6. However, as shown in Table 11.8, the variation attributed to the architecture increased to 12.62%. This increase shows that in this scenario, where there is a lower response time and the anytime agent is not affected by random failures, it tends to score more than the default agent. With this, we demonstrate that there are cases where the anytime agent can perform better than the default agent in complex scenarios, such as the Multi-Agent Programming Contest.

Map	Architecture	Points scored		
		r1	r2	r3
Sparse	Default	8	17	2
Sparse	Anytime	8	7	22
Clustered	Default	0	12	12
Clustered	Anytime	10	22	10

Table 11.7: *Points scored by the agents in the no random failure MAPC experiment.*



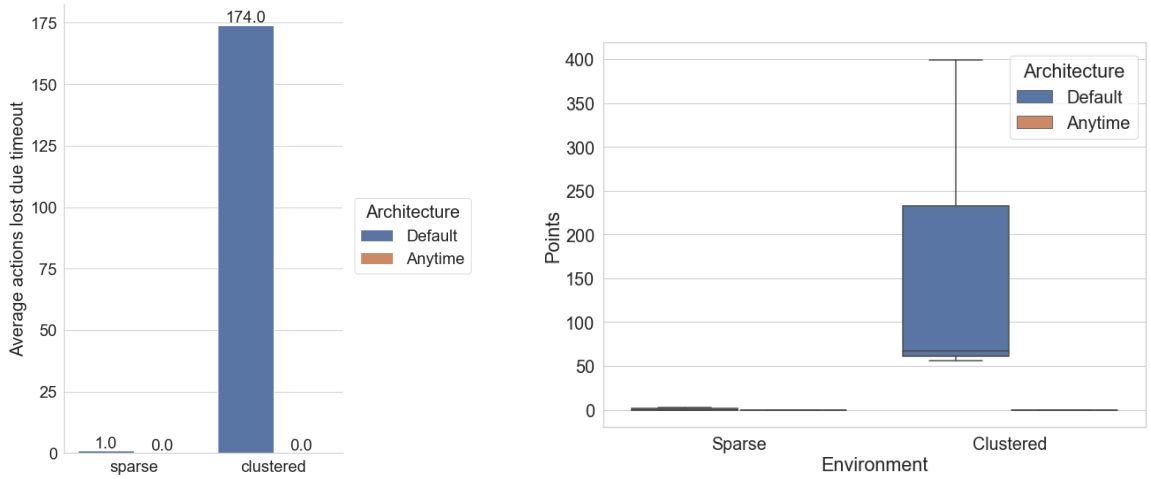
(a) *Average points scored in each level.*

(b) *Box plot of points scored in each level.*

Figure 11.6: *Points scored in the no random failure MAPC experiment.*

Factors	Influence
Variation of Map:	0.06%
Variation of Architecture:	12.62%
Variation of interaction of Map & Architecture:	1.03%
Variation of error and unobserved factors:	86.28%

Table 11.8: *Variation attributed to each factor.*



(a) *Average actions lost in each level.*

(b) *Box plot of actions lost in each level.*

Figure 11.7: *Actions lost in the no random failure MAPC experiment.*

Map	Architecture	Actions lost		
		r1	r2	r3
Sparse	Default	0	0	3
Sparse	Anytime	0	0	0
Clustered	Default	399	56	67
Clustered	Anytime	0	0	0

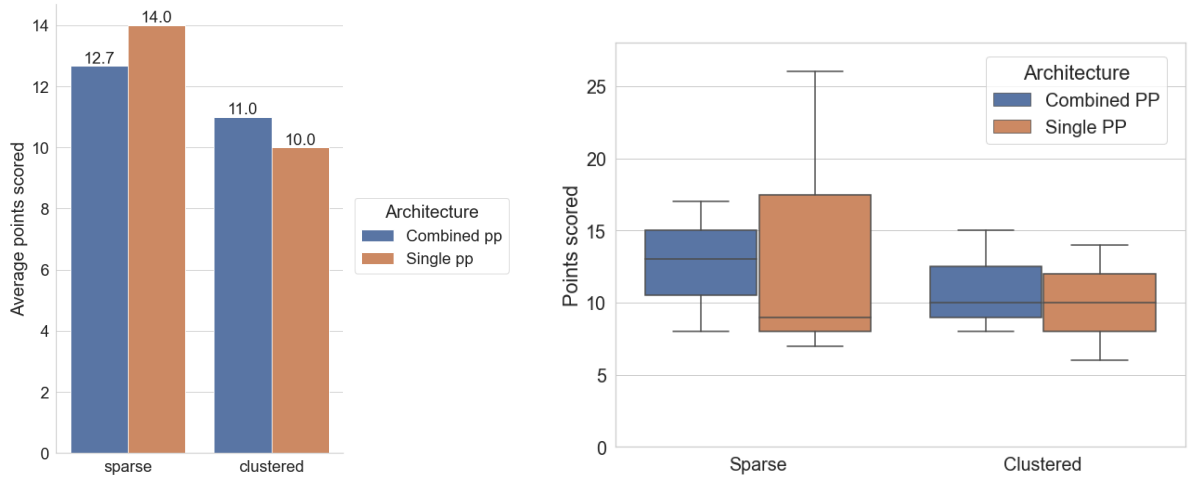
Table 11.9: *Actions lost by the agents due to timeout in the no random failure MAPC experiment.*

11.4 Performance profile evaluation (E8)

As we defined the agent profiling process, each agent records its runtimes and creates performance profiles based on them. Since all agents have the same code, the performance profiles would be very similar in theory. Because of this, we decided to investigate whether combining the profiling data from all agents and creating a single performance profile would benefit the agent. This experiment uses two factors and the same environment configuration used in the experiment E5 in Section 11.1. The first is the map, and the second is the performance profile used. At the level where each agent has its profile, we call it "Single Performance Profile." At the level where all agents use the same profile with information from all profilings combined, we call it "Combined Performance Profile." We present the agents' scores in Figure 11.8 and Table 11.10.

Map	Architecture	Points scored		
		r1	r2	r3
Sparse	Combined PP	17	13	8
Sparse	Single PP	9	7	26
Clustered	Combined PP	15	10	8
Clustered	Single PP	6	10	14

Table 11.10: *Points scored by the agents in the performance profile evaluation experiment.*



(a) *Average points scored in each level.*

(b) *Box plot of points scored in each level.*

Figure 11.8: *Points scored in the performance profile evaluation experiment.*

Factors	Influence
Variation of Map:	6.98%
Variation of Performance profile:	0.02%
Variation of interaction of Map & Performance profile:	1.18%
Variation of error and unobserved factors:	91.81%

Table 11.11: *Variation attributed to each factor.*

Based on the analysis of variations, we have that for the proposed scenario, there was no change in the agents' scores as a result of the change in the generation of performance profiles.

Chapter 12

Conclusions and further work

This work addressed the lack of control over the reasoning cycle of BDI agents. As there is no way to control the agent's reasoning cycle, the agent does not present guarantees as to the time required for it to act in the environment. This lack of guarantees significantly impacts when one wants to integrate such agents into environments that demand responses from agents within a predetermined time-bound or in scenarios where processing time negatively influences the agent's utility. In some cases, it may even be impossible to use BDI agents in competitive scenarios, such as RoboCup and MAPC.

12.1 Conclusions

Seeking to solve this problem, we proposed a new architecture of BDI agents called `Anytime BDI`. This architecture uses the idea of anytime algorithms to control the execution time of each part of the BDI model. With that, it is possible to define the agent's processing time and guarantee that it will execute an action in the environment before a predetermined time-bound.

This model was implemented in the Jason interpreter, receiving the name `Anytime Jason`. We then applied the implemented model to two scenarios. Then, we performed statistical tests using techniques from the performance analysis domain, based on the design model of experiments presented by Jain, 1991.

In the analyzed case studies, the architecture reduced the agents' processing time and, in many cases, increased the agents' scores.

Based on the results obtained in Chapters 10 and 11, we are now able to answer questions proposed in Section 1.2:

- **Q1:** *Given a specific time response upper bound, is it possible to guarantee that a BDI agent can often enough process perceptions, deliberate on them, and determine the action it wants to perform within the time limit while simultaneously guaranteeing a minimum quality of actions?*
  We proposed a BDI architecture capable of creating agents that can analyze perceptions, reason about beliefs, desires, and intentions, and act according to plans

to achieve their goals within a predefined time upper bound. By converting the Sense, Deliberate and Act mechanisms into anytime algorithms and adding profiling and monitoring mechanisms as presented in Chapter 6 we can control the execution time of BDI agents.

- **Q2:** *How to define in advance the minimum time necessary for the agent to produce a response with the desired quality?*
  We provided the proposed architecture with a mechanism capable of estimating an agent's minimum execution time by using profiling techniques and constructing performance profilers for the agents' internal mechanisms. Also, calculating the minimum time allows for evaluating if applying the agent in the desired domain is possible.

- **Q3:** *What is the impact of processing time on response quality?*
  We evaluated the quality of actions generated by the proposed architecture. In this evaluation, we compares the loss of quality caused by the execution time reduction with the loss of quality caused by the agent not having enough time to deliberate. As analyzed by the experiments in Chapters 10 and 11: In scenarios where the agent's utility decreases over time, the AnytimeBDI architecture can increase the agent's utility. Furthermore, in scenarios where there is no such increase, the use of the architecture can also be beneficial in reducing agent processing time at the cost of a controlled loss in quality.

By answering the three research questions, this work establishes a solid architecture whose objective is to provide the ability to control the response time of BDI agents while guaranteeing a certain predefined level of quality in the actions. This capability makes it possible to use BDI agents in the most diverse domains where it is necessary to guarantee an upper bound on the agent's response time. These domains include simulations with a rigid time restriction and embedded agents where an agent not spending too much time without acting on the environment is desired. We expect this architecture to be a new incentive for using BDI agents with discrete event simulators and in the robotics area, resulting in further development and popularization of the BDI model.

It is also important to note two things about these results. The first is that these results are valid in the proposed scenarios. Thus, there is no guarantee that the results will be the same in other scenarios. However, as the results were consistent for both proposed scenarios, we are confident that these would hold in many other scenarios. Second, we did not explore all possible settings for splitting the agents' response time. Thus, there may be settings for AnytimeJason that would further improve the agent's performance.

12.2 Future work

During the development of this work, many points were not profoundly examined. The following steps of this research may involve:

- Since there is no intention reconsideration mechanism in Jason, we have not pro-

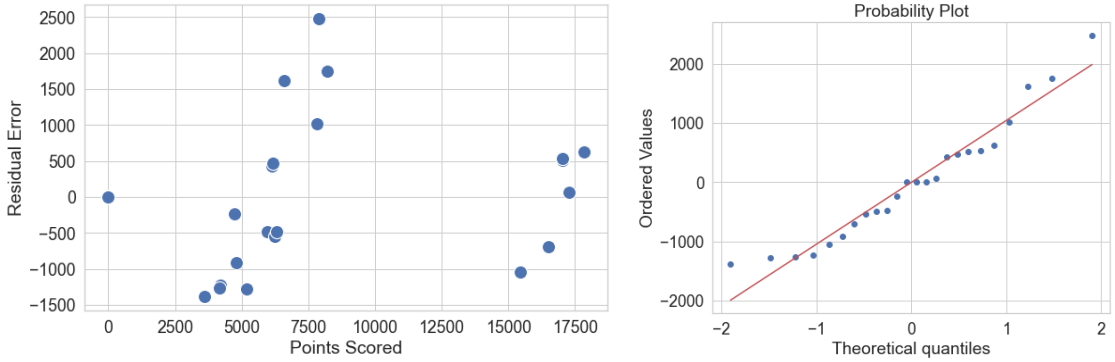
posed an anytime algorithm for intention reconsidering. Proposing this algorithm and adding it to the **Anytime Jason** implementation would bring it closer to the **Anytime BDI** model;

- We decided to implement the **Anytime BDI** architecture in the Jason language due to the language's popularity and open-source nature. However, there are several other well-accepted agent programming languages. Implementing the **Anytime BDI** architecture in other BDI agent programming languages would reinforce the idea that the theoretical model is generic enough to be implemented in several languages, making it possible to use **Anytime BDI** in agents created in other languages;
- Adapting an agent programming language to work according to the **Anytime BDI** architecture is laborious. The creation of a guide or a cookbook with directions on how best to conduct this process could facilitate the implementation of the model in new languages;
- Although we built **Anytime Jason** aiming not to need changes to the Jason agent's code, making a few changes can make the agent more efficient. These changes include creating perception filters, prioritizing plans, and adequately structuring the plans. Also, knowing how to choose the best time allocation is essential. Therefore, it would be interesting to create a guide with instructions for implementing and adapting Jason agents to **Anytime Jason**;
- In order to get good test coverage, we ran tests in two reasonably different contexts, although it is probable that some scenarios went unreported. Therefore, testing the architecture and implementation in additional scenarios with different characteristics would certainly reveal more interesting uses of the architecture;
- During the development, we kept the Monitor module very simple. It only calculated a possible time allocation and controlled each module to run for the specified time. However, it can be beneficial to sophisticate the Monitor. One possibility would be to develop a method to reuse leftover time from other modules. For example, if there are no more perceptions to be analyzed and there is still time left, this time could be reallocated to the other two modules. For this, it would be necessary for the Monitor to be prepared to reallocate this time in the best possible way;
- A typical analysis of anytime algorithms is how the algorithm's quality increases as it has more time to execute. A test scenario that analyses the agent quality as response time gradually increases could bring new insights to the architecture;
- Although the architecture calculates the optimal time allocation points, likely, they will not all present the same quality of agent execution. The development of some mechanism that could direct the agent developer to better choices of time allocations could make the development of agents faster and improve the final performance of the agent.

Appendix A

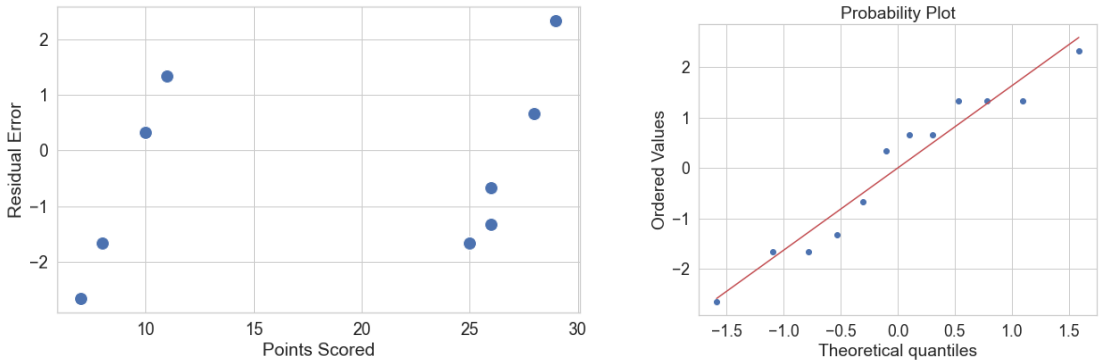
Experiments visual validations

As seen in Chapter 9, it is necessary to validate the results of the experiments through visual validations. Therefore, in this appendix, we present the validations referring to all the experiments carried out in Chapters 10 and 11. Thus, in the visual tests, the quantile-quantile plot's lack of patterns and the normality of the residuals are evident. These characteristics show that the results of the experiments are correct.



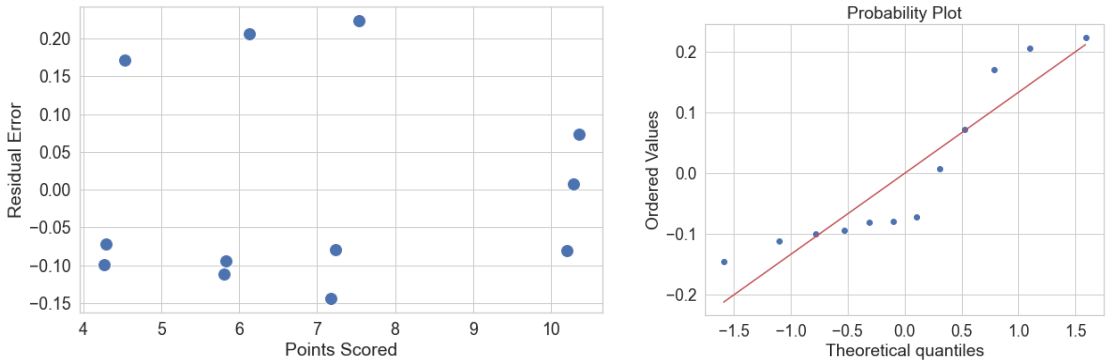
(a) *Normal quantile-quantile plot for the residuals.* **(b)** *Plot of the residuals versus predicted response.*

Figure A.1: *Visual diagnostic tests for experiment E1 in Section 10.1.1.*



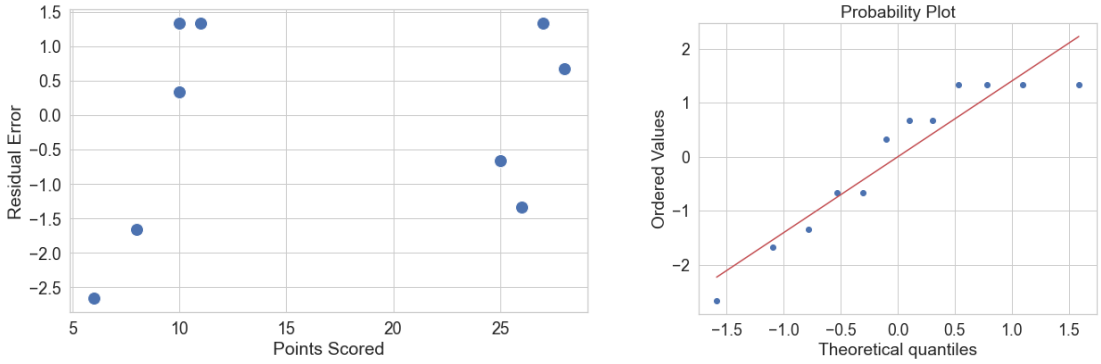
(a) *Normal quantile-quantile plot for the residuals.* **(b)** *Plot of the residuals versus predicted response.*

Figure A.2: *Visual diagnostic tests for captures variation in experiment E2 on Section 10.1.2.*



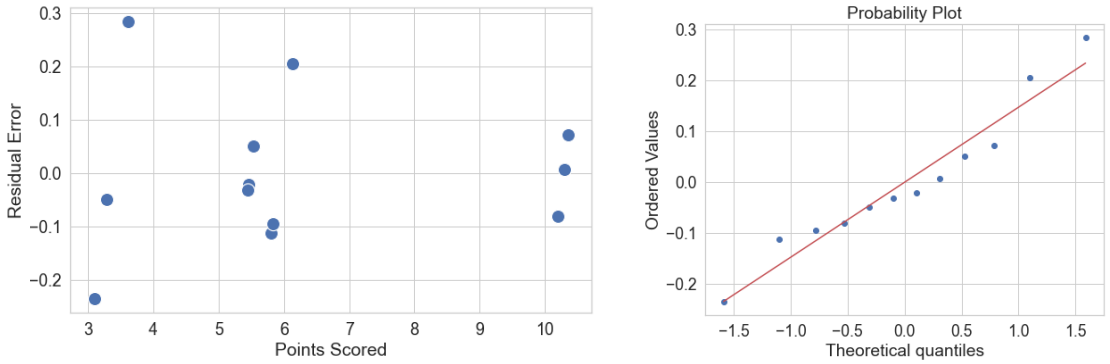
(a) *Normal quantile-quantile plot for the residuals.* **(b)** *Plot of the residuals versus predicted response.*

Figure A.3: *Visual diagnostic tests for response time variation in experiment E2 on Section 10.1.2.*



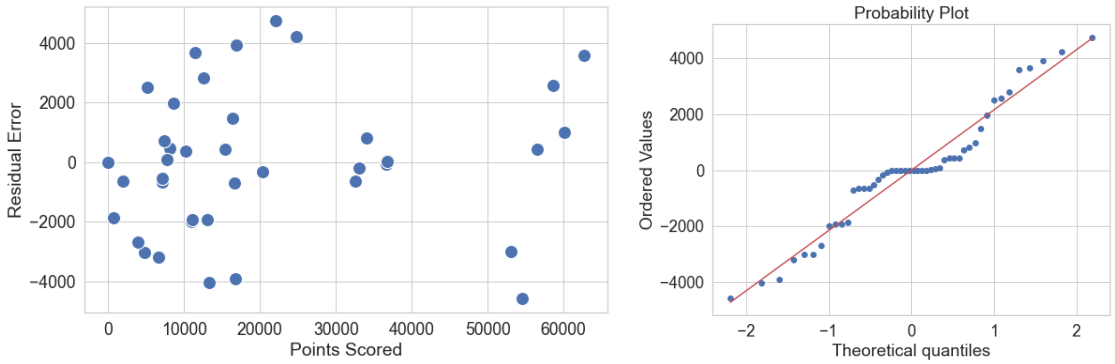
(a) *Normal quantile-quantile plot for the residuals.* **(b)** *Plot of the residuals versus predicted response.*

Figure A.4: *Visual diagnostic tests for captures variation in experiment E2 on Section 10.1.2.1.*



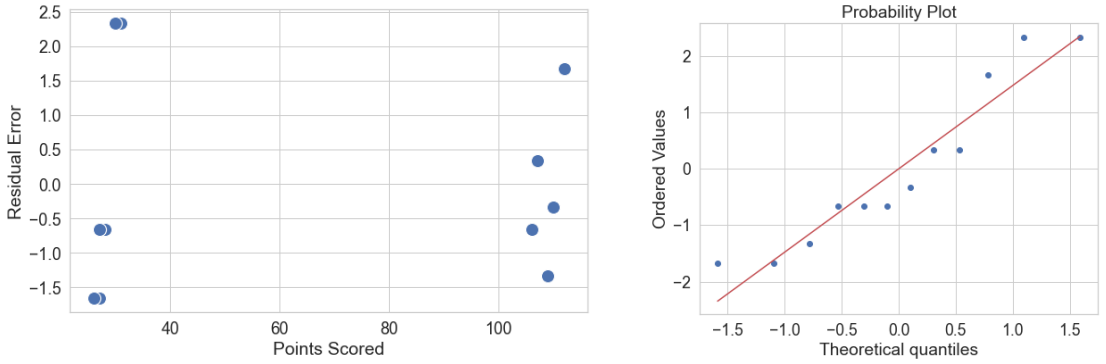
(a) *Normal quantile-quantile plot for the residuals.* **(b)** *Plot of the residuals versus predicted response.*

Figure A.5: *Visual diagnostic tests for response time variation in experiment E2 on Section 10.1.2.1.*



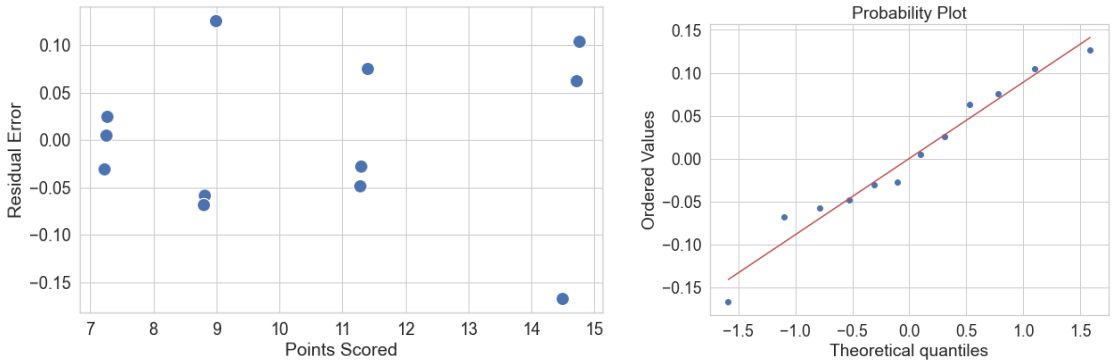
(a) *Normal quantile-quantile plot for the residuals.* **(b)** *Plot of the residuals versus predicted response.*

Figure A.6: *Visual diagnostic tests for experiment E3 in Section 10.2.1.*



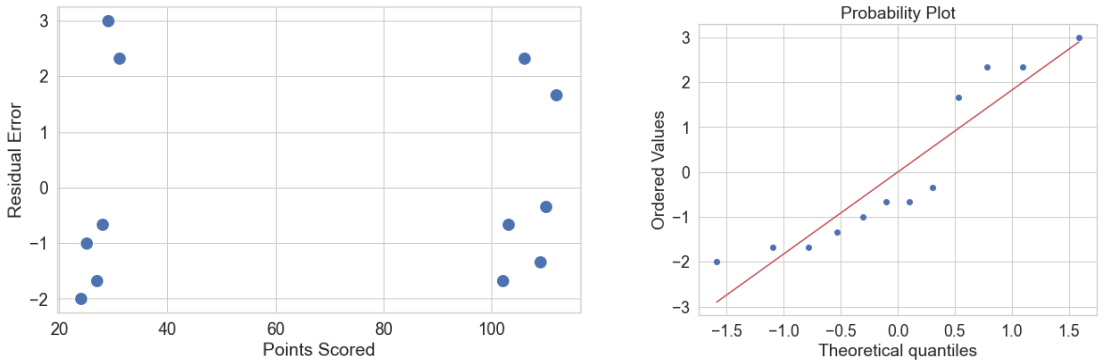
(a) *Normal quantile-quantile plot for the residuals.* **(b)** *Plot of the residuals versus predicted response.*

Figure A.7: *Visual diagnostic tests for captures variation in experiment E4 on Section 10.2.2.*



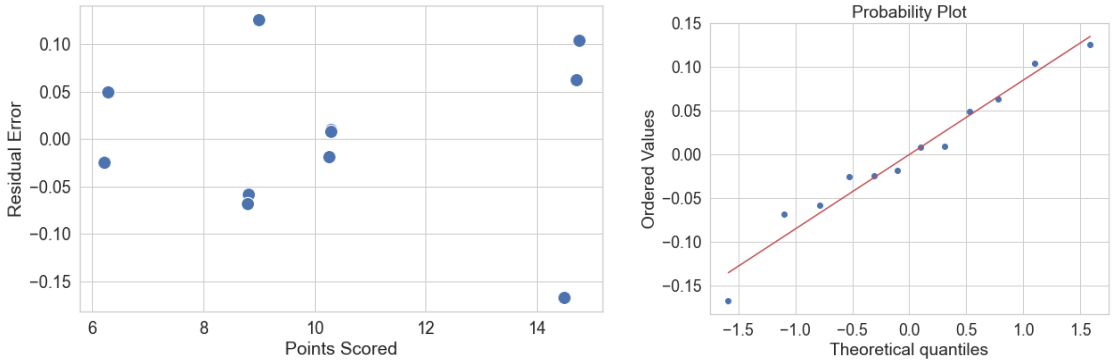
(a) *Normal quantile-quantile plot for the residuals.* **(b)** *Plot of the residuals versus predicted response.*

Figure A.8: *Visual diagnostic tests for response time variation in experiment E4 on Section 10.2.2.*



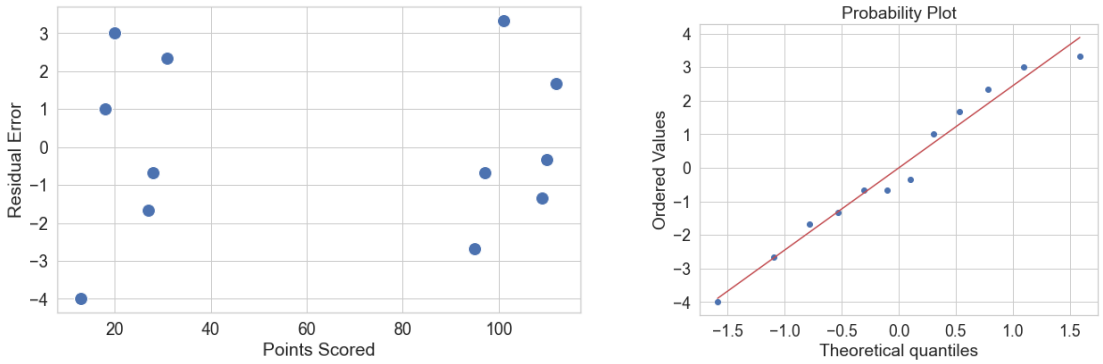
(a) *Normal quantile-quantile plot for the residuals.* **(b)** *Plot of the residuals versus predicted response.*

Figure A.9: *Visual diagnostic tests for captures variation in experiment E4 on Section 10.2.2.1.*



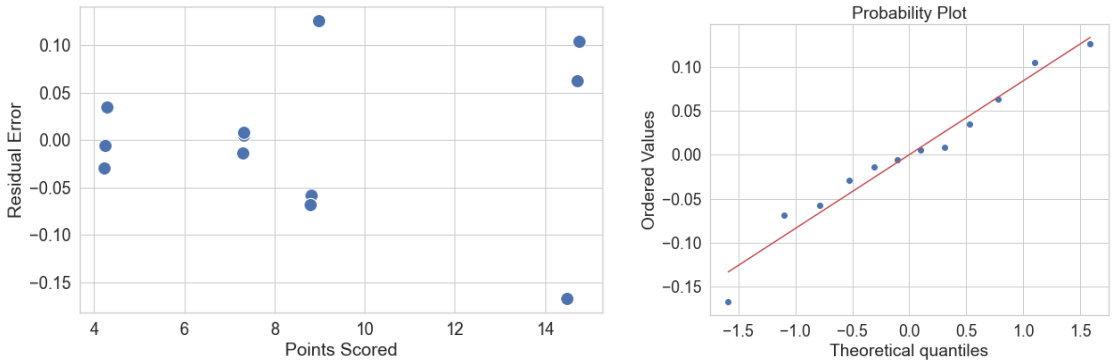
(a) *Normal quantile-quantile plot for the residuals.* **(b)** *Plot of the residuals versus predicted response.*

Figure A.10: *Visual diagnostic tests for response time variation in experiment E4 on Section 10.2.2.1.*



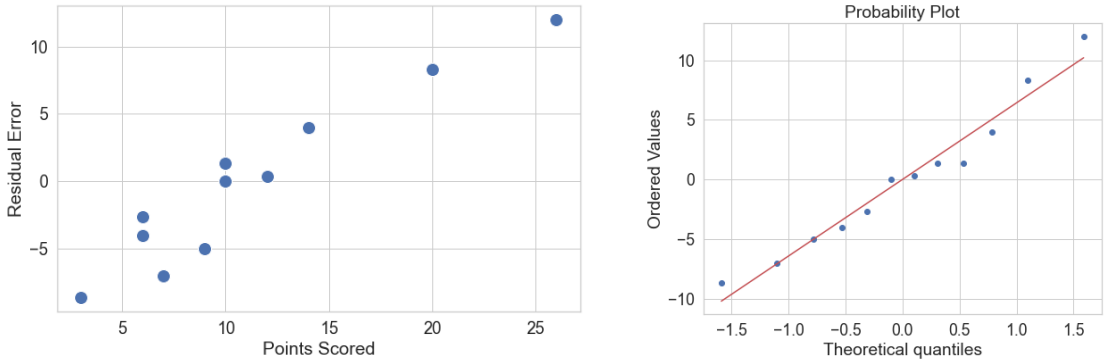
(a) *Normal quantile-quantile plot for the residuals.* **(b)** *Plot of the residuals versus predicted response.*

Figure A.11: *Visual diagnostic tests for captures variation in experiment E4 on Section 10.2.2.2.*



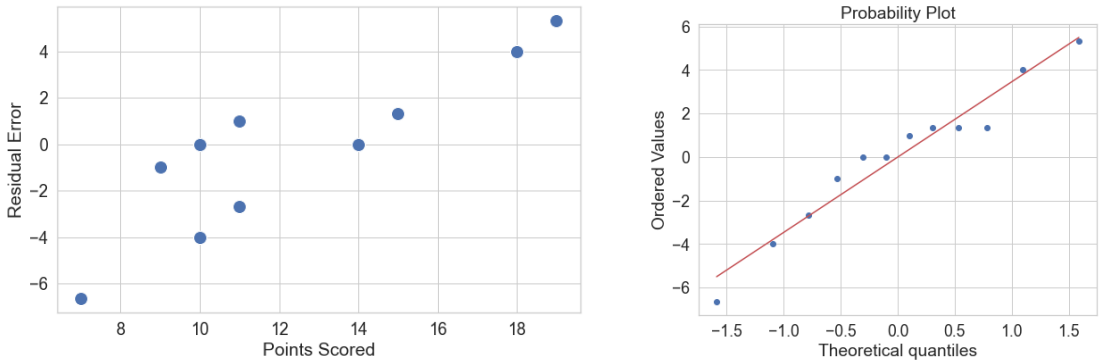
(a) *Normal quantile-quantile plot for the residuals.* **(b)** *Plot of the residuals versus predicted response.*

Figure A.12: *Visual diagnostic tests for response time variation in experiment E4 on Section 10.2.2.2.*



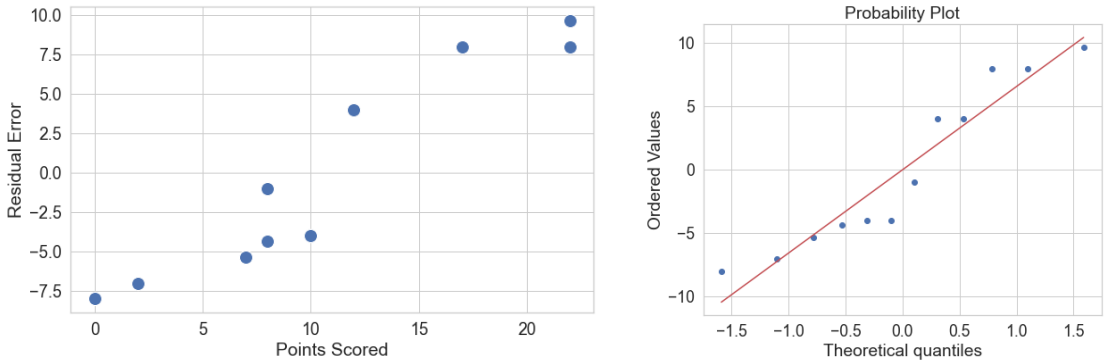
(a) *Normal quantile-quantile plot for the residuals.* **(b)** *Plot of the residuals versus predicted response.*

Figure A.13: *Visual diagnostic tests for experiment E5 in Section 11.1.*



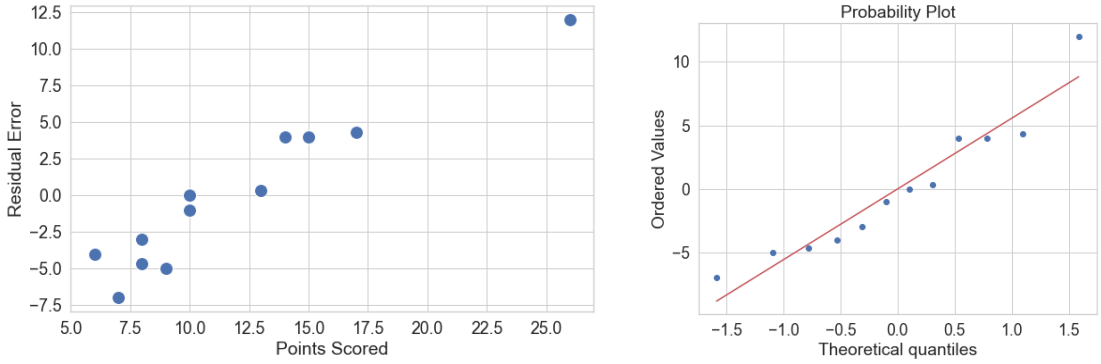
(a) *Normal quantile-quantile plot for the residuals.* **(b)** *Plot of the residuals versus predicted response.*

Figure A.14: *Visual diagnostic tests for experiment E6 in Section 11.2.*



(a) *Normal quantile-quantile plot for the residuals.* **(b)** *Plot of the residuals versus predicted response.*

Figure A.15: *Visual diagnostic tests for experiment E7 in Section 11.3.*



(a) *Normal quantile-quantile plot for the residuals.* **(b)** *Plot of the residuals versus predicted response.*

Figure A.16: *Visual diagnostic tests for experiment E8 in Section 11.4.*

Appendix B

Anytime Jason specific commands

In this chapter, we will present the commands added to the Jason framework during the implementation of `Anytime Jason`. These are the necessary commands to run a Jason agent using the proposed model and implementation.

B.1 Architecture usage

To use the anytime architecture, it is only necessary to select it in the same way as with the other available Jason variations. In the infrastructure specification, the agent developer must write $Centralised(anytime, responseTime)$. $responseTime$ is specified in milliseconds and is the only required parameter. There are two methods for controlling the quality of time allocations: The first is specifying only the upper bounds. For this, we use the command $Centralized(anytime, responseTime, perceptionLowerLimit, deliberateLowerLimit)$, where $perceptionLowerLimit$ and $deliberateLowerLimit$ are numbers between 0 and 1 that represent a minimum quality limit for each module. Thus, by specifying both values, the ϵ-constraint method will select a time allocation that obeys this constraint. The second way is to specify both upper and lower limits through the command $Centralised(anytime, responseTime, perceptionLowerLimit, deliberateLowerLimit, perceptionUpperLimit, deliberateUpperLimit)$. We present an example of this command in Algorithm 16. Also, in this case, $perceptionUpperLimit$ and $deliberateUpperLimit$ are values between 0 and 1. With these parameters, the ϵ-constraint method will select a time allocation whose quality is between the upper and lower limit.

B.2 Perception filters

We developed five internal actions to control the perception filters. We present examples of each of these functions in Algorithm 17. The perception selection mechanism evaluates first the perceptions that fit filters with lower priority values. That is, perceptions with a value of 1 will be the first to be evaluated. Next are perceptions with a value of 2 and so on. The values do not need to be sequential. If there are three filters with priority values 3,

Algorithm 16 `Anytime Jason` agent configuration file example.

```
1: MAS mars{
2:     infrastructure : Centralised(anytime, 40, 0.5, 0, 0.59, 0.04)
3:     environment : MarsEnv
4:     executionControl : jason.control.ExecutionControl
5:     agents : agent1;
6: }
```

5, and 12, the mechanism will evaluate them in that order without affecting performance by the selection mechanism. Multiple filters can not have the same priority value. Doing this will overwrite the previous filter.

Create To create a new perception filter, we use the *create* function. This function receives three parameters: The name of the filter, its priority, and a list of restrictions. The filter name is a string, the priority is an integer value, and the constraints are a list where each constraint is composed of a position indicator, a comparison operator, and a comparison value. On line 1 of Algorithm 17, the filter "name1" has a priority of 3 and accepts all perceptions whose predicate is not the word insect. That is, all perceptions not in the form "$insect(...)$". On line 2, the filter "name2" has a priority of 2 and accepts all perceptions whose first component is greater than 5. As an example, it would accept the perception "$insect(10, ...)$" but not the "$insect(3, ...)$". Lastly, on line 3, the filter "name3" has a priority of 4 and accepts all perceptions that both has a predicate starting with "position" and the second component is less or equal to 10. As an example, it would accept the perception "$positionAg(12, 10, ...)$" but not the perceptions "$positionAg(13, 13, ...)$" or "$agent\_position(1, 1, ...)$". The following operators can be used:

- **eq**: Tests for equality;
- **ne**: Tests for difference;
- **bg**: Tests if the value in the perception begins with the value in the filter;
- **gt**: Tests if the value in the perception is bigger then the value in the filter;
- **ge**: Tests if the value in the perception is bigger or equal to the value in the filter;
- **lt**: Tests if the value in the perception is smaller then the value in the filter;
- **eq**: Tests if the value in the perception is smaller or equal to the value in the filter;

Delete The *delete* internal action takes the filter name as a parameter and completely removes it. We present an example of it in line 4 of Algorithm 17.

Activate/Deactivate The internal actions *deactivate* and *activate* are responsible for temporarily suspending and resuming filter operation. Both actions take the filter name as a parameter. Deactivate suspends a filter, meaning that it is not considered in the analysis of perceptions until it is activated again by the activate action. We present an example of these internal actions in lines 5 and 6 of Algorithm 17

Change priority The *changePriority* internal action parameters are the filter name and a new priority value. This action replaces the old priority value for the one in the parameter, possibly changing the order in which the perception module will apply the filters. We present an example of the action in line 7 of Algorithm 17.

Algorithm 17 `Anytime Jason` perception filters example.

```
1: +!useFilters <- .filter.create(name1, 3, [[0, ne, insect]]);
2:                 .filter.create(name2, 2, [[1, gt, 5]]);
3:                 .filter.create(name3, 4, [[0, bg, position], [2, le, 10]]);
4:                 .filter.delete(name3).
5:                 .filter.deactivate(name2);
6:                 .filter.activate(name2);
7:                 .filter.changePriority(name1, 1).
```

B.3 Plan priority

As described in Section 7.3, when the anytime architecture finds two possible external actions to execute in the environment, the architecture automatically chooses the one with the lowest priority value. These priorities can be defined through annotations in the plans, as shown in Algorithm 18.

Algorithm 18 `Anytime Jason` perception filters example.

```
1: @label[priority(1)]
2: +!moveNorthwest <- -move(north); move(west);
```

Bibliography

[Ahlbrecht *et al.* 2021] Tobias Ahlbrecht, Jürgen Dix, Niklas Fiekas, and Tabajara Krausburg. "The 15th multi-agent programming contest". In: *The Multi-Agent Programming Contest 2021*. Ed. by Tobias Ahlbrecht, Jürgen Dix, Niklas Fiekas, and Tabajara Krausburg. Cham: Springer International Publishing, 2021, pp. 3–20. isbn: 978-3-030-88549-6 (cit. on p. 87).

[Bellifemine *et al.* 1999] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. "JADE - A FIPA-compliant agent framework". In: *Proceedings of PAAM*. 1999, pp. 97–108. isbn: 158113326X. doi: 10.1145/375735.376120 (cit. on p. 12).

[Bordini, Hübner, *et al.* 2007] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John-Wiley & Sons Ltd, 2007, p. 273. isbn: 9780470057476 (cit. on pp. 14, 15).

[Bordini, Braubach, *et al.* 2006] Rafael H Bordini, Lars Braubach, *et al.* "A Survey of Programming Languages and Platforms for Multi-Agent Systems". In: *Informatica (Slovenia)* 30.1 (2006), pp. 33–44 (cit. on pp. 12, 14).

[Bratman 1987] M. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, 1987. isbn: 9780674458185 (cit. on p. 1).

[Braubach *et al.* 2003] L Braubach, W Lamersdorf, and Alexander Pokahr. "Jadex : Implementing a BDI-Infrastructure for JADE". In: *EXP in search of innovation* 3.September (2003), pp. 76–85 (cit. on p. 12).

[Dastani *et al.* 2003] Mehdi Dastani, Frank De Boer, Frank Dignum, and John-Jules Meyer. "Programming agent deliberation: an approach illustrated using the 3APL language". In: *Proceedings of the second international joint conference on Autonomous agents and multiagent systems* (2003), pp. 97–104 (cit. on pp. 12, 13).

[Dean and Boddy 1988] Thomas Dean and Mark Boddy. "An Analysis of Time-Dependent Planning". In: *Seventh AAAI National Conference on Artificial Intelligence (AAAI)*. Ed. by Howard E. Shrobe, Tom M. Mitchell, and Reid G. Smith. 6. AAAI Press / The MIT Press, 1988, pp. 49–54. isbn: 0-262-51055-3 (cit. on p. 19).

[Ferguson 1992] Innes A. Ferguson. "Touring Machines: Autonomous Agents with Attitudes". In: *Computer* 25.5 (1992), pp. 51–55. issn: 00189162. doi: 10.1109/2.144395 (cit. on p. 2).

[Gat 1998] Erann Gat. "Three-layer Architectures". In: *Artificial Intelligence and Mobile Robots*. Ed. by David Kortenkamp, R. Peter Bonasso, and Robin Murphy. MIT Press, 1998, pp. 195–210. isbn: 0-262-61137-6 (cit. on p. 2).

[Hindriks *et al.* 1999] Koen V Hindriks, Frank S De Boer, Wiebe Van Der Hoek, and John-Jules Ch. Meyer. "Agent Programming in 3APL". In: *Autonomous Agents and Multi-Agent Systems* 2.4 (1999), pp. 357–401. issn: 1387-2532. doi: 10.1023/A:1010084620690 (cit. on p. 12).

[Jain 1991] Raj Jain. "The art of computer system performance analysis: techniques for experimental design, measurement, simulation and modeling". In: *New York: John Willey* (1991) (cit. on pp. 4, 67–70, 97).

[Kostiadis and Hu 2000] K Kostiadis and H S Hu. "A multi-threaded approach to simulated soccer agents for the RoboCup competition". In: *Robocup-99: Robot Soccer World Cup Iii* 1856 (2000), pp. 366–377. issn: 0302-9743 (cit. on pp. 1, 31, 55, 56, 58, 62, 63).

[Lorini and Piunti 2010] Emiliano Lorini and Michele Piunti. "Introducing relevance awareness in BDI agents". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 5919 LNAI. 2010, pp. 219–236. isbn: 3642148425. doi: 10.1007/978-3-642-14843-9_14 (cit. on pp. 50, 60, 61, 63).

[Miettinen 2008] Kaisa Miettinen. "Introduction to multiobjective optimization: noninteractive approaches". In: *Multiobjective Optimization: Interactive and Evolutionary Approaches*. Ed. by Jürgen Branke, Kalyanmoy Deb, Kaisa Miettinen, and Roman Słowiński. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–26. isbn: 978-3-540-88908-3. doi: 10.1007/978-3-540-88908-3_1 (cit. on pp. 25, 26, 48).

[Pantoja *et al.* 2016] Carlos Eduardo Pantoja, Marcio Fernando Stabile Jr., Nilson Mori Lazarin, and Jaime Simão Sichman. "ARGO: An Extended Jason Architecture that Facilitates Embedded Robotic Agents Programming". In: *Engineering Multi-Agent Systems - 4th International Workshop, EMAS 2016, Singapore, Singapore, May 9-10, 2016, Revised, Selected, and Invited Papers*. Ed. by Matteo Baldoni, Jörg P. Müller, Ingrid Nunes, and Rym Zalila-Wenkstern. Springer, 2016, pp. 136–155 (cit. on p. 2).

[Rao 1996] Anand S Rao. "AgentSpeak(L): BDI agents speak out in a logical computable language". In: *Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world (MAAMAW'96)*. Ed. by Walter Van de Velde and John W Perram. Vol. 1038. Lecture Notes in Artificial Intelligence. Secaucus, USA: Springer-Verlag, 1996, pp. 42–55 (cit. on p. 13).

[Rao and Georgeff 1991] Anand S. Rao and Michael P. Georgeff. "Modeling Rational Agents within a BDI-Architecture". In: *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*. KR'91. Cambridge, MA, USA: Morgan Kaufmann Publishers Inc., 1991, pp. 473–484. isbn: 1558601651 (cit. on pp. 1, 9, 31).

[Santos *et al.* 2015] Fernando R. Santos, Jomi F. Hubner, and Leandro B. Becker. "Concepção e Análise de um Modelo de Agente BDI Voltado para o Planejamento de Rota em um VANT". In: *Proceedings of WESAAC 2015 9th Software Agents, Environments and Applications School*. Ed. by Baldoino Fonseca, Viviane Torres da Silva, and Ricardo Choren. 2015, pp. 66–77 (cit. on p. 2).

[Schut *et al.* 2004] Martijn Schut, Michael Wooldridge, and Simon Parsons. *The theory and practice of intention reconsideration*. Vol. 16. 4. 2004, pp. 261–293. ISBN: 0952813041. DOI: 10.1080/09528130412331309277 (cit. on p. 37).

[Shoham 1993] Y Shoham. "Agent-oriented programming". In: *Artificial Intelligence* 60.1 (Mar. 1993), pp. 51–92. ISSN: 00043702. DOI: 10.1016/0004-3702(93)90034-9 (cit. on p. 9).

[Stabile Jr and Sichman 2015a] Márcio F. Stabile Jr and Jaime S Sichman. "Incorporando Filtros de Percepção para Aumentar o Desempenho de Agentes Jason". In: *Workshop-Escola de Sistemas de Agentes, seus Ambientes e apliCações (WESAAC)*. 2015, p. 12 (cit. on pp. 50, 61).

[Stabile Jr and Sichman 2015b] Márcio Fernando Stabile Jr and Jaime S Sichman. "Melhorando o desempenho de agentes BDI Jason através de filtros de percepção". MA thesis. Universidade de São Paulo, 2015, p. 84 (cit. on pp. 40, 49, 50, 72).

[Stabile Jr. 2022] Marcio Fernando Stabile Jr. "Using Multi-objective Optimization to Generate Timely Responsive BDI Agents". In: *AAMAS 2022 Doctoral Consortium*. 2022, pp. 1875–1877 (cit. on p. 35).

[Stabile Jr. and Sichman 2021] Marcio Fernando Stabile Jr. and Jaime S. Sichman. "The lti-usp strategy to the 2020/2021 multi-agent programming contest". In: *The Multi-Agent Programming Contest 2021*. Ed. by Tobias Ahlbrecht, Jürgen Dix, Niklas Fiekas, and Tabajara Krausburg. Cham: Springer International Publishing, 2021, pp. 108–133. ISBN: 978-3-030-88549-6 (cit. on p. 88).

[Traldi *et al.* 2022] Andrea Traldi, Francesco Bruschetti, Marco Robol, Marco Roveri, and Paolo Giorgini. *Real-Time BDI Agents: a model and its implementation*. 2022. DOI: 10.48550/ARXIV.2205.00979 (cit. on pp. 62, 63).

[Tweedale *et al.* 2007] J. Tweedale *et al.* "Innovations in multi-agent systems". In: *Journal of Network and Computer Applications* 30.3 (2007), pp. 1089–1115. ISSN: 10848045. DOI: 10.1016/j.jnca.2006.04.005 (cit. on p. 1).

[Van Oijen and Dignum 2011] Joost Van Oijen and Frank Dignum. "Scalable perception for BDI-agents embodied in virtual environments". In: *Proceedings - 2011 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, IAT 2011* 2 (Aug. 2011), pp. 46–53. DOI: 10.1109/WI-IAT.2011.176 (cit. on pp. 60, 61, 63).

[Wooldridge 2000] M Wooldridge. *Reasoning About Rational Agents*. MIT Press, 2000 (cit. on pp. 10, 11, 45, 58).

[Wooldridge 1997] Michael Wooldridge. "Agent-Based Software Engineering". In: *IEE Proceedings of Software Engineering 144*. 1997 (cit. on p. 9).

[Wooldridge 2009] Michael Wooldridge. *An Introduction to Multiagent Systems*. Second. John Wiley & Sons Ltd, 2009. ISBN: 0470519460 (cit. on p. 9).

[Yao and Logan 2016] Yuan Yao and Brian Logan. "Action-Level Intention Selection for BDI Agents". In: *Aamas 2016*. C. 2016, pp. 1227–1236. ISBN: 978-1-4503-4239-1 (cit. on pp. 59, 60, 62, 63).

[Zatelli 2017] Maicon Rafael Zatelli. "Exploiting Parallelism in the Agent Paradigm". PhD thesis. 2017 (cit. on pp. 58, 59, 62, 63).

[Zatelli *et al.* 2016] Maicon Rafael Zatelli, Alessandro Ricci, and Jomi F. Hübner. "A Concurrent Architecture for Agent Reasoning Cycle Execution in Jason". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Ed. by Michael Rovatsos, George Vouros, and Vicente Julian. Vol. 9571. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 425–440. ISBN: 978-3-319-33508-7. DOI: 10.1007/978-3-319-33509-4 (cit. on p. 31).

[Zhang and Huang 2005] Huiliang Zhang and Shell Ying Huang. "A parallel BDI agent architecture". In: *Proceedings - 2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, IAT'05* 2005.May (2005), pp. 157–160. DOI: 10.1109/IAT.2005.17 (cit. on pp. 31, 56, 58).

[Zhang and Huang 2007] Huiliang Zhang and Shell Ying Huang. "A general framework for parallel BDI agents". In: *Proceedings - 2006 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2006 Main Conference Proceedings), IAT'06* (2007), pp. 103–109. DOI: 10.1109/IAT.2006.8 (cit. on pp. 31, 56–58, 62, 63).

[Ziafati and Dastani 2013] Pouyan Ziafati and Mehdi Dastani. "Agent Programming Languages Requirements for Programming Autonomous Robots". In: (2013), pp. 35–53 (cit. on pp. 1, 2).

[Zilberstein 1993] Shlomo Zilberstein. "Operational rationality through compilation of anytime algorithms". PhD thesis. 1993, p. 164 (cit. on pp. 19, 21, 22, 36, 48).

[Zilberstein 1995] Shlomo Zilberstein. "Operational rationality through compilation of anytime algorithms". In: *AI Magazine* (1995) (cit. on p. 36).

[Zilberstein 1996] Shlomo Zilberstein. "Using Anytime Algorithms in Intelligent Systems". In: *AI Magazine* 17.3 (1996), pp. 73–83 (cit. on p. 36).