Performance Analysis of the Jason Reasoning Cycle

by

Jason Miller

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

© 2021
Jason Miller
Abstract

Jason is a popular interpreter of AgentSpeak(L) that provides an environment to develop and run autonomous agents. We are interested in determining if Jason can be used to develop autonomous robots that can be run on modern hardware. To this end, we determined the time complexity of each of the ten steps of the Jason reasoning cycle and used that complexity to help us identify what parameters could be changed to have the biggest effect on the execution time of the reasoning cycle. We also generate a model to predict future performance. We validate this model using three different case studies, comparing the predicted execution times to those we measure. We find that our developed model consistently predicted an execution time one order of magnitude higher than the measured execution time for each case study. We further find that, using modern hardware, Jason provides a suitable environment for autonomous robots.
Acknowledgements

I would like to first and foremost sincerely thank my thesis supervisor, Professor Babak Esfandiari. He has given me the latitude to pursue this research in the direction I desired, while providing extremely insightful suggestions and questions to ensure I didn’t get lost along the way.

Special thanks to Patrick Gavigan, who was instrumental in collecting data from hardware that I didn’t have access to.

I also wish to express my deep gratitude to my wife, Annie St-Amour, who has supported me every step of the way. Annie, your belief in me kept me going when I didn’t think I could.

Finally, thank you to my sponsors.

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) and Four DRobotics.

Nous remercions le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) ainsi que Four DRobotics de leur soutien.
# Contents

Abstract i  

Acknowledgements ii  

List of Tables x  

List of Figures xii  

List of Algorithms xvi  

Acronyms xvii  

1 Introduction 1  

1.1 Contributions 2  

1.2 Publications 4  

1.3 Structure 4  

2 Background 5  

2.1 BDI 5  

2.2 AgentSpeak(L) 7  

2.3 Jason 9
2.3.1 Reasoning Cycle .............................................. 9
2.4 Summary ......................................................... 11

3 Related Work ...................................................... 12
3.1 Comparison Between 2APL, GOAL, and Jason .................... 12
3.2 Other Works ...................................................... 13
3.3 Projects currently using Jason .................................... 15
3.4 Summary ......................................................... 16

4 Methodology ....................................................... 17
4.1 Analysis of Reasoning Cycle ..................................... 18
4.2 Model Development .............................................. 19
4.3 Case Study ....................................................... 21
4.4 Summary ......................................................... 22

5 Analysis ........................................................... 23
5.1 Agent Implementation ............................................ 23
  5.1.1 Agent ......................................................... 25
  5.1.2 Belief Base ................................................... 25
  5.1.3 Plan Library .................................................. 26
  5.1.4 Agent Architecture ......................................... 26
  5.1.5 Circumstance ................................................ 26
  5.1.6 Transition System .......................................... 26
  5.1.7 Selection Functions ........................................ 27
  5.1.8 Discussion ................................................... 27
5.2 Sequence Analysis of the Reasoning Cycle ......................... 28
  5.2.1 Sense ........................................................ 28
<table>
<thead>
<tr>
<th>Section</th>
<th>Subsection</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.1.1</td>
<td>Step 1 - Gather Perceptions</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>5.2.1.2</td>
<td>Step 2 - Update Belief Base</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>5.2.1.3</td>
<td>Step 3 - Check Mailbox</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>5.2.1.4</td>
<td>Throttling of Perception</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>5.2.1.5</td>
<td>Step 4 - Checking if a Message is “Socially Acceptable”</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>5.2.2</td>
<td>Deliberate</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>5.2.2.1</td>
<td>Step 5 - Selecting an Event to Deliberate Upon</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>5.2.2.2</td>
<td>Step 6 - Selecting a List of Relevant Plans</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>5.2.2.3</td>
<td>Step 7 - Selecting a List of Applicable Plans</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>5.2.2.4</td>
<td>Step 8 - Selecting an Option</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>5.2.2.5</td>
<td>Optimizing the Deliberate Phase</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>5.2.3</td>
<td>Act</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>5.2.3.1</td>
<td>Step 9 - Selecting an Intention</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>5.2.3.2</td>
<td>Step 10 - Executing an Intention</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>5.2.4</td>
<td>Discussion</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Analysis of the Time Complexity of the Reasoning Cycle</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>5.3.1</td>
<td>Step 1 - Perceive</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>5.3.2</td>
<td>Step 2 - Belief Update Function</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>5.3.3</td>
<td>Step 3 - Check Mail</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>5.3.4</td>
<td>Step 4 - Determining if a Message is “Socially Acceptable”</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>5.3.5</td>
<td>Step 5 - Selecting an Event</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>5.3.6</td>
<td>Step 6 - Collecting Relevant Plans</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>5.3.7</td>
<td>Step 7 - Collecting Applicable Plans</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>5.3.8</td>
<td>Step 8 - Selecting an Option</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>5.3.9</td>
<td>Step 9 - Selecting an Intention</td>
<td>51</td>
<td></td>
</tr>
</tbody>
</table>
5.3.10 Step 10 - Executing the Intention .......................... 51
5.3.11 Step X - Combine Step 6 through Step 8 into one action ... 53
5.3.12 Discussion ................................................. 53
5.4 Data Flow Analysis of the Reasoning Cycle ....................... 55
5.4.1 Discussion .................................................. 57
5.5 Summary ....................................................... 57

6 Experiments 59

6.1 Function: perceive() ............................................ 61
   6.1.1 Laptop Results ........................................... 63
   6.1.2 Raspberry Pi Results ..................................... 67
   6.1.3 Discussion of Results .................................... 69
6.2 Function: buf() .................................................. 70
   6.2.1 Laptop Results ........................................... 72
   6.2.2 Raspberry Pi Results ..................................... 75
   6.2.3 Discussion of Results .................................... 78
6.3 Function: applyFindOp() ......................................... 79
   6.3.1 Laptop Results ........................................... 80
      6.3.1.1 Plans ............................................... 81
      6.3.1.2 Beliefs ............................................. 81
      6.3.1.3 Belief Rules ....................................... 85
      6.3.1.4 Determining the Final Equation ................. 88
   6.3.2 Raspberry Pi Results ..................................... 90
      6.3.2.1 Plans ............................................... 90
      6.3.2.2 Beliefs ............................................. 92
      6.3.2.3 Belief Rules ....................................... 92
6.3.2.4 Determining the Final Equation .................................. 95
6.3.3 Discussion of Results .................................................. 99

6.4 Function: \texttt{applyExecInt()} ........................................... 99
  6.4.1 Laptop Results ....................................................... 100
    6.4.1.1 Beliefs .......................................................... 100
    6.4.1.2 Belief Rules .................................................. 101
  6.4.2 Raspberry Pi Results ............................................... 101
    6.4.2.1 Beliefs .......................................................... 103
    6.4.2.2 Belief Rules .................................................. 103
  6.4.3 Discussion of Results ............................................... 104

6.5 Discussion .............................................................. 106
  6.5.1 Laptop Discussion .................................................. 107
  6.5.2 Raspberry Pi Discussion ........................................... 108
  6.5.3 Discussion Summary ................................................ 109

6.6 Summary ............................................................... 110

7 Case Studies ............................................................. 112
  7.1 Domestic Robot ...................................................... 113
    7.1.1 Results of Domestic Robot Execution ............................ 115
    7.1.2 Discussion of Results ........................................... 119
  7.2 Grid World ............................................................ 120
    7.2.1 Results of Grid World Execution ............................... 121
    7.2.2 Discussion of Results ........................................... 122
  7.3 Mail Delivery Robot .................................................. 125
    7.3.1 Results of Mail Delivery Robot Execution ...................... 128
    7.3.2 Discussion of Results ........................................... 130
List of Tables

5.1 Summary of Time Complexity per Step of the Reasoning Cycle . . . . 55
6.1 Summary of Functions to Analyze and Their Associated Parameters . 59
6.2 Summary of Hardware Specifications of Test Systems . . . . . . . . 60
6.3 $R^2$ values of curves of best fit for perceive() . . . . . . . . . . 66
6.4 Percepts Modified per Reasoning Cycle . . . . . . . . . . . . . . . 71
6.5 $R^2$ values of curves of best fit for buf() . . . . . . . . . . . . . . . 73
6.6 $R^2$ values of curves of best fit for the plans of applyFindOp() . . . 81
6.7 $R^2$ values of curves of best fit for the beliefs of applyFindOp() . . 83
6.8 $R^2$ values of curves of best fit for the belief rules of applyFindOp() . 86
6.9 Additional Time Outside of Reasoning Cycle Steps . . . . . . . . . . 110
7.1 Parameters Extracted From Robot Agent . . . . . . . . . . . . . . . 114
7.2 Predicted Execution Times of Domestic Robot Agent from Model (in $\mu$s) 115
7.3 Order of Maximum Execution Time for Domestic Robot Agent (in $\mu$s) 118
7.4 Parameters Extracted from navigatorEnvAssist Agent . . . . . . . . . 121
7.5 Predicted Execution Times of Grid World Agent from Model (in $\mu$s) . 121
7.6 Order of Maximum Execution Time for Grid World Agent (in $\mu$s) . . 124
7.7 Updated Summary of Hardware Specifications of Test Systems . . . . 126
7.8 Parameters Extracted from Mail Robot Agent . . . . . . . . . . . . . 128
7.9 Predicted Execution Times of Mail Robot Agent from Model (in $\mu$s) . 128
7.10 Order of Maximum Execution Time for Mail Robot Agent (in $\mu$s) . . 130
7.11 Summary of Maximum Order of Execution Time for Case Studies . . 131
8.1 Predicted Execution Times of Domestic Robot Agent from Model (in $\mu$s)135
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Generic Belief-Desire-Intention (BDI) Architecture [1]</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Reasoning Cycle State Diagram [2]</td>
<td>10</td>
</tr>
<tr>
<td>5.1</td>
<td>Agent Class Diagram [2]</td>
<td>24</td>
</tr>
<tr>
<td>5.2</td>
<td>State Transitions Within Jason Reasoning Cycle [2]</td>
<td>27</td>
</tr>
<tr>
<td>5.3</td>
<td>The Jason Reasoning Cycle Sequence Diagram - Sense</td>
<td>29</td>
</tr>
<tr>
<td>5.4</td>
<td>The Jason Reasoning Cycle Activity Diagram - Sense</td>
<td>30</td>
</tr>
<tr>
<td>5.5</td>
<td>The Jason Reasoning Cycle Sequence Diagram - Deliberate</td>
<td>32</td>
</tr>
<tr>
<td>5.6</td>
<td>The Jason Reasoning Cycle Activity Diagram - Deliberate</td>
<td>33</td>
</tr>
<tr>
<td>5.7</td>
<td>The Jason Reasoning Cycle Sequence Diagram - Act</td>
<td>37</td>
</tr>
<tr>
<td>5.8</td>
<td>The Jason Reasoning Cycle Activity Diagram - Act</td>
<td>38</td>
</tr>
<tr>
<td>5.9</td>
<td>Jason Reasoning Cycle Data Flow</td>
<td>56</td>
</tr>
<tr>
<td>6.1</td>
<td><code>perceive</code>: Execution Time vs. Percepts - Laptop (Boxplot)</td>
<td>64</td>
</tr>
<tr>
<td>6.2</td>
<td><code>perceive</code>: Execution Time vs. Percepts - Laptop</td>
<td>65</td>
</tr>
<tr>
<td>6.3</td>
<td><code>perceive</code>: Execution Time vs. Percepts - Laptop (Extreme)</td>
<td>65</td>
</tr>
<tr>
<td>6.4</td>
<td><code>perceive</code>: Execution Time vs. Percepts - Raspberry Pi (Boxplot)</td>
<td>67</td>
</tr>
<tr>
<td>6.5</td>
<td><code>perceive</code>: Execution Time vs. Percepts - Raspberry Pi</td>
<td>68</td>
</tr>
<tr>
<td>6.6</td>
<td><code>perceive</code>: Execution Time vs. Percepts - Raspberry Pi (Extreme)</td>
<td>68</td>
</tr>
</tbody>
</table>
6.30 applyFindOp(): Execution Time vs. Belief Rules - Raspberry Pi (Boxplot) ........................................... 94
6.31 applyFindOp(): Execution Time vs. Belief Rules - Raspberry Pi ....................................................... 95
6.32 applyFindOp(): Execution Time vs. Belief Rules - Raspberry Pi (Extreme) ........................................... 96
6.33 applyFindOp(): Execution Time vs. Beliefs, Belief Rules and Plans - Raspberry Pi ........................................... 97
6.34 applyExecInt(): Execution Time vs. Beliefs - Laptop (Boxplot) ......................................................... 100
6.35 applyExecInt(): Execution Time vs. Beliefs - Laptop ............................................................................. 101
6.36 applyExecInt(): Execution Time vs. Belief Rules - Laptop (Boxplot) ................................................. 102
6.37 applyExecInt(): Execution Time vs. Belief Rules - Laptop ................................................................ 102
6.38 applyExecInt(): Execution Time vs. Belief Rules - Raspberry Pi (Boxplot) ........................................... 103
6.39 applyExecInt(): Execution Time vs. Belief Rules - Raspberry Pi ......................................................... 104
6.40 applyExecInt(): Execution Time vs. Belief Rules - Raspberry Pi (Boxplot) ........................................... 105
6.41 applyExecInt(): Execution Time vs. Belief Rules - Raspberry Pi ......................................................... 106
7.1 Domestic Robot: Execution Time vs. Function Called - Laptop ......................................................... 116
7.2 Domestic Robot: Execution Time vs. Function Called - Raspberry Pi .................................................... 117
7.3 Domestic Robot: Log Execution Time vs. Function Called - Laptop .................................................... 117
7.4 Domestic Robot: Log Execution Time vs. Function Called - Raspberry Pi ................................................ 118
7.5 Grid World: Execution Time vs. Function Called - Laptop ................................................................. 122
7.6 Grid World: Execution Time vs. Function Called - Raspberry Pi ......................................................... 123
7.7 Grid World: Log Execution Time vs. Function Called - Laptop ............................................................... 123

xiv
7.8 Grid World: Log Execution Time vs. Function Called - Raspberry Pi 124
7.9 Mail Robot: Execution Time vs. Function Called - Raspberry Pi 4 129
7.10 Mail Robot: Log Execution Time vs. Function Called - Raspberry Pi 4 129

8.1 Execution Times of Profiler Compared to Source Code Instrumentation 133
8.2 Domestic Robot: Execution Time vs. Function Called - Laptop 136
8.3 Domestic Robot: Log Execution Time vs. Function Called - Laptop 136
## List of Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Perceive</td>
<td>41</td>
</tr>
<tr>
<td>2</td>
<td>Belief Update Function</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>Check Mail</td>
<td>44</td>
</tr>
<tr>
<td>4</td>
<td>Socially Acceptable</td>
<td>44</td>
</tr>
<tr>
<td>5</td>
<td>Select Event</td>
<td>45</td>
</tr>
<tr>
<td>6</td>
<td>Relevant Plans</td>
<td>46</td>
</tr>
<tr>
<td>7</td>
<td>Unifies</td>
<td>46</td>
</tr>
<tr>
<td>8</td>
<td>Applicable Plans</td>
<td>48</td>
</tr>
<tr>
<td>9</td>
<td>Logical Consequences</td>
<td>49</td>
</tr>
<tr>
<td>10</td>
<td>Select Option</td>
<td>50</td>
</tr>
<tr>
<td>11</td>
<td>Select Intention</td>
<td>51</td>
</tr>
<tr>
<td>12</td>
<td>Apply and Execute Intention</td>
<td>52</td>
</tr>
<tr>
<td>13</td>
<td>Apply and Find Option</td>
<td>54</td>
</tr>
</tbody>
</table>
Acronyms

BDI . . . . . . . . . . . . . . . . . . . . Belief-Desire-Intention

IQR . . . . . . . . . . . . . . . . . . . . Inter-Quartile Range

ROS . . . . . . . . . . . . . . . . . . . . Robot Operating System

UAV . . . . . . . . . . . . . . . . . . . . Unmanned Aerial Vehicle
Chapter 1

Introduction

In order to be fully autonomous, an autonomous robot must pursue user-defined goals while adapting to a dynamically and rapidly changing environment. It would be potentially beneficial to have a programming language and framework that allows for easy definition of missions, translation of sensory input into internal representations of the state of the environment, simple definition of plans that act based on such states, and monitoring of which goals and plans are achievable given the current state. Such a framework would also have to be computationally efficient given the hardware limitations of the embedded systems at the heart of these mobile robots.

The belief-desire-intention (BDI) paradigm serves as a method by which a rational agent can be created that has the autonomy to act in a constantly changing environment. We see BDI as a good model to investigate, as it has a well defined and formalized language built upon it in AgentSpeak(L) [3]. We will look specifically at Jason [2, 4], which is a popular interpreter of AgentSpeak(L) [5].

Jason provides a framework for a developer to create an agent. It is declarative, meaning Agentspeak(L) allows the developer to focus on defining goals, plans and rules,
while Jason takes care of triggering relevant rules, monitoring the plans, selecting, and executing the applicable ones. It is implemented with an extendable design. Many complex operations (e.g. belief revision) have only a simple default implementation provided, leaving it up to the developer to decide if more complex versions of these operations are needed. Example agents are provided to showcase aspects of Jason’s feature set. However, we should note that these example agents are extremely simple and exist within a static environment and might not be representative of agents designed to perform real tasks.

Our main interest is to develop a model using a given configuration of hardware to predict the execution time of the reasoning cycle and use this model to determine if agents written in Jason are suitable for use in autonomous robots on that hardware. We will find that our model is mostly pessimistic about the execution time. In the best case, our model’s predictions come within the same order of magnitude as the actual execution time. In the worst case, the model predicts an execution time of one order of magnitude higher than the actual execution time. We also find that the execution time of the reasoning cycle of our “real world” case study (see section 7.3) runs in the order of 0.1 seconds. The prediction results together with the actual measured execution time signify that our pessimistic model predicts an overall satisfactory performance for these case studies.

1.1 Contributions

The main contributions of this thesis are:

1. an in-depth analysis of the BDI reasoning cycle — this includes pseudo-code abstracted away from Jason’s code, to the point where it could be potentially
valid for any interpreter based on AgentSpeak(L). We have determined the time complexity of each step so that we can determine how the execution time will scale in response to modifying parameters, and what parameters are potential performance bottlenecks and should be investigated further. We expect that this analysis is general enough for any BDI based agent language that follows the general sense-deliberate-act cycle, not just Jason. Specifically, we have found bottlenecks in the function \texttt{buf()} (called in step 2), and the function \texttt{logicalConsequence()} (called in step 7 and 10);

2. experimental validation — we design agents to vary the aspects of the reasoning cycle that we concluded have an effect on performance based on our analysis of the time complexity, which we run on two different configurations of hardware to provide contrasting data. We use this data to create two performance models that are specific for the studied hardware configuration;

3. three case studies looking at agents that exist in the world — we pick three agents that were not written by us so that our bias does not affect the development of the agent. Also, we want a sample of agents that exist outside the scope of this work. One of our case studies runs on an autonomous robot aiming to get as close to a “real world” as possible, in which we find that our model provides an execution time prediction one order of magnitude larger than the measured execution time. Given that this is a first attempt at this type of model, we find this to be a satisfactory result.
1.2 Publications

Work that has been done as part of this thesis has been presented at the 9th International Workshop on Engineering Multi-Agent Systems (EMAS 2021) [6]. It is currently waiting to be published in the proceedings.

1.3 Structure

Chapter 2 contains background relevant to understanding Jason and its implementation, mainly an explanation of AgentSpeak(L) concepts and an overview of the Jason reasoning cycle. Chapter 3 looks at work that has already been done with respect to the Jason reasoning cycle, specifically work looking at optimizing bottlenecks currently present, as well as a look at existing projects that use Jason in “real world” scenarios. Chapter 4 outlines how we will go about our performance analysis of the reasoning cycle. Chapter 5 presents our analysis, including algorithms created to relate the Jason reasoning cycle to the more generic BDI reasoning cycle along with determining the time complexity. We provide experimental data to validate our analysis in chapter 6 and use that data to generate a model to predict the execution time of other agents. We look at three different agents and use our developed model to predict the execution time of their reasoning cycles in chapter 7. An addendum is added as chapter 8 which removes the profiler in favour of instrumenting the Jason source code, highlighting the added overhead of the chosen profiler. Finally, we conclude in chapter 9 and provide a summary of our work, along with potential future work.
Chapter 2

Background

To determine if Jason is suitable for use in single board computers, we must first understand the underlying concepts of Jason and the framework it builds upon. We will use this background to understand how the Jason reasoning cycle operates and identify areas of interest for deeper investigation. We will first look at the Belief-Desire-Intention paradigm (BDI) and how its reasoning cycle is constructed. We will then look at AgentSpeak(L), a formal language that conforms to BDI, to identify and explain basic concepts that we require to understand how agents are designed and run. Finally, we will explore additional concepts added by Jason, a programming language that complies to the AgentSpeak(L) syntax and semantics, as well as how Jason implements the reasoning cycle.

2.1 BDI

BDI is a paradigm that specifies how a rational agent can reason about its environment and make decisions based on its goals and the plans that are available to it. The agent uses its beliefs about the environment combined with its desires (what it wants to
Figure 2.1: Generic BDI Architecture [1]

achieve) to generate intentions about how it will proceed towards those desires.

Figure 2.1 shows a representation of the generic BDI model upon which Jason is built. The BDI model has three phases: sense, deliberate and act. These three phases will execute sequentially for the entire life cycle of an agent. The sense phase consists of an agent gathering input, either from external sources (e.g. communication from other agents) or from internal sources (e.g. sensors built into the agent, also referred to as ‘percepts’), which is represented as ‘Sensor Input’ in figure 2.1. The phase ends with running belief revision on its belief base (‘Belief Revision’ in figure 2.1), which is the process by which an agent updates its existing beliefs based on new information. The deliberate phase begins with the agent generating a list of potential options (‘Generate Options’ in figure 2.1) that have been selected from the list of plans available to it and written by the developer. These options are generated based both on the change in beliefs and the intentions that an agent has. These options are then fed into a database of desires. The deliberation cycle concludes with a series of filters (‘Filter’ in figure 2.1), in which an agent, with knowledge of its beliefs, desires and intentions, will narrow the list of intentions down to a single one. Finally, the act phase encompasses the action the agent will take based on the result of the deliberation it has done.
2.2 AgentSpeak(L)

AgentSpeak(L) is defined in [3]. It is syntactically a prolog-like language, and its semantics are meant to conform to the BDI paradigm. Many interpreters have been created to conform to AgentSpeak(L)’s syntax and semantics, one of the most popular ones being Jason [5]. AgentSpeak(L) defines the following terms:

- **Belief**: a collection of literals, represented by predicates (P) that make up the knowledge of an agent. An atomic predicate is represented in the following format (from [2]):

\[
\text{at ::= } P(t_1 \ldots t_n) \quad (n \geq 0)
\]

For example, the belief \texttt{available(beer, fridge)} would mean that an agent believes that beer is available in the fridge.

- **Goal**: there are two types, test goals (?at) and achievement goals (!at), where at represents a predicate (P) in the same format as the above.

\[
g ::= ?at \mid !at
\]

A test goal is a way for an agent to see if a condition is true. For example, testing to see if there is beer available in the fridge. An achievement goal is the specific condition the agent wants to make true. For example, an achievement goal for an agent might be !drunk(beer), meaning an agent wants to get to a state in which is has drunk a beer.

- **Triggering event**: an event generated by a change in environment, or the addition/removal of a goal which can trigger the addition/deletion of beliefs or goals
as seen here (from [2]):

\[ te ::= +at \mid -at \mid +g \mid -g \]

For example, an agent might have a triggering event \(-available(beer, fridge)\) in the case that it believes that beer is no longer available in the fridge (i.e: the belief \(available(beer, fridge)\) has been removed).

- **Action**: something that an agent does to modify the environment, represented in the following format (from [2]):

\[ a ::= A(t_1, \ldots, t_n) \quad (n \geq 0) \]

For example, an agent may order more beer through the \(order(beer, store)\) action.

- **Plan**: a set of actions that have a triggering event as well as a context, shown here (from [2]):

\[ p ::= te : ct \leftarrow h. \]

\[ ct ::= at \mid -at \mid ct \& ct \]

\[ h ::= a \mid g \mid +at \mid -at \]

For example, an agent may have a plan to order more beer if it is out of beer that might look like \(-available(beer, fridge) : canAfford(beer) \leftarrow order(beer, store)\). In this case, the triggering event is that an agent no longer believes that beer is available in the fridge. The context is that the agent believes that it can afford beer. The action is for the agent to order some beer from the store.

AgentSpeak(L) further describes three selection functions, which allow an agent to
determine what should be done at a given time. The event selection function selects a single event out of a set of triggering events to act upon. The option selection function selects an option (also referred to as a plan) out of a set of applicable plans, which is a set of plans that match the triggering event and have contexts which evaluate to true. The intention selection function selects a single intention from a set of intentions, where an intention is a stack of partially instantiated plans that are available to be acted upon.

2.3 Jason

Jason [2, 4] implements AgentSpeak(L) and provides an environment to write agents. It also adds some additional features on top of the definition of AgentSpeak(L), specifically:

- Belief rules: a type of reasoning an agent can do based on its beliefs. For example, a belief rule could be `tooMuch(beer) :- consumed(beer) & limit(beer, B)` which would compare how much beer an agent has consumed to whether or not the limit of beer allowed to an agent has been reached. The agent would then add a belief `tooMuch(beer)` if the rule is true.

2.3.1 Reasoning Cycle

As well as these additional features, Jason also implements the reasoning cycle, which is the heart of the BDI engine. It is, in essence, how an agent decides what actions it should take and runs throughout the life of the agent. We see in figure 2.1 that a BDI architecture should take in some sort of sensor input, which we see represented
in figure 2.2 as step 1. Step 3 and 4 can also be interpreted as a type of sensor input (as it is input from external agents), which bypasses the belief revision and is stored within the belief base directly. Figure 2.1 then shows the sensor input undergoing belief revision. Jason implements a lighter weight belief revision in step 2 in the form of belief update, but will perform belief revision if required later during the reasoning cycle. The *generate options* step in figure 2.1 is implemented in Jason as the event selection function, represented in step 5. The filtering referenced in figure 2.1 is implemented in Jason as a set of functions starting in steps 6 and 7, and ending with the option selection function and intention selection function in steps 8 and 9. Finally, the action output is implemented as step 10, in which the intention actually gets executed. We will determine the time complexity of these steps as part of our performance analysis study in later chapters of this thesis.
2.4 Summary

In this chapter we have provided an overview of BDI, AgentSpeak(L) and Jason. We have looked at the generic BDI algorithm and compared it to the algorithm implemented by Jason, as well as defining the terms which are important to be familiar with in order to understand how the reasoning cycle operates. This background will allow us to look at the work done by others in an attempt to understand their performance analysis.
Chapter 3

Related Work

The popularity of Jason makes it a desirable choice, as work has already been done that looks at using Jason to drive autonomous robots (see section 3.3). Work also exists that look into the execution time of portions of the reasoning cycle (see section 3.2), although no work looks at the entirety of the reasoning cycle. We also find that in a simple comparison between Jason and two other BDI based languages (2APL and GOAL), agents written in Jason execute in significantly less time than agents written in either of the two other languages (see section 3.1). Based on the popularity, the lack of existing research and the performance compared to other languages, we feel that Jason is a good choice for study.

3.1 Comparison Between 2APL, GOAL, and Jason

To begin, [7] looks at comparing three languages, 2APL [8], GOAL [9] and Jason. One of the main components of this work is the idea of a similarity notation, which
attempts to keep the agent’s programs similar enough to be able to meaningfully compare them. Using this concept, they write three programs for the three agents that will calculate a desired Fibonacci number. This work explicitly mentions that a GOAL agent could be written that would run more efficiently if the desire to keep the runs similar was ignored. Their results are generated by calculating the Fibonacci number ranging from 50 to 500, in increments of 50 and measuring two factors; the number of steps (reasoning cycles) required to compute the number and the time it took to compute the number. The data reported showed that Jason executed significantly faster than the other two languages, with 2APL being a distant second, in terms of execution time. However, it should be noted that, in general, an agent that only calculates Fibonacci numbers is extremely simple. That said, the idea that Jason is significantly faster than the other two languages presented gives us hope that Jason will be suitable for use in the area of autonomous robotics.

3.2 Other Works

Other works that we will look at examine specific aspects of the execution time of the reasoning cycle. For instance, [10] puts forward the idea that increasing the number of percepts that are perceived by an agent will adversely affect the execution time of the reasoning cycle. In order to test this idea, this work uses JProfiler [11] to profile the execution time of the components of the reasoning cycle. Their initial results showed that the belief update function and the unification function (the process by which variables are bound to values) took up a large amount of the execution time, and would both benefit in fewer percepts being processed by the agent. Reasoning that robotic applications of the reasoning cycle could potentially require a large number
of percepts, they proposed a method to filter out unneeded percepts in an effort to artificially reduce the length of the execution time.

Their experimentation consisted of an agent in a grid world, with a limit to how far an agent can perceive within this world (2 squares, 5 squares, 7 squares and no limit). This causes the agent to potentially lower how many percepts it must process. As the number of percepts were filtered to smaller and smaller values, it became very evident that artificially lowering the number of percepts to be processed by the reasoning cycle would lower the execution time.

An alternate method to attempt to decrease the execution time of the Jason reasoning cycle is done through the use of caching. [12] looks into the idea that caching queries that are made many times during a single invocation of the reasoning cycle would lead to a decrease in the execution time. This work specifically explores queries against the belief base and goals, seeking to determine if caching these queries over a single reasoning cycle will have beneficial results. Through experimental observation of three different languages (2APL, GOAL and Jason), they found that many queries about beliefs and goals were repeated many times. They also found that through the introduction of a cache, their execution time was decreased by at least 10%. As part of this work, it was also found that some queries were repeated over multiple reasoning cycles. This conclusion lead to [13], which explores caching over multiple reasoning cycles. No experimental data was generated for multi-cycle caching using Jason, but this work speculates that significant performance gains would be seen when implementing a multi-cycle cache.

Finally, we look at [14], which, instead of looking at the execution time of the Jason reasoning cycle, modifies AgentSpeak(L) into AgentSpeak(L++). They also pair this new version of AgentSpeak(L) with a new interpreter, inspired by Jason, named
LightJason. This work theorizes that agent-oriented programming languages, such as GOAL and Jason, are not largely adopted for many reasons, including performance and scalability. As such, they put forward AgentSpeak(L++), which among other features support lambda expressions, parallel execution of actions within plans and thread-safe variables. Their main idea is not to lower the execution time of the existing Jason reasoning cycle, but to instead allow for agents to be easily distributed and execute actions in parallel, thereby theoretically resulting in a lower execution time for LightJason compared to Jason. Experimentation consisted of placing a varying number of agents in a grid world environment and giving them the goal to leave the room, recording the number of reasoning cycles it took for those agents to complete their goal. Comparison is done within LightJason, with an increasing number of agents, but, unfortunately, no comparisons are done against Jason to determine if a LightJason agent is truly faster than a Jason agent.

3.3 Projects currently using Jason

Projects exist that use Jason, in parallel to the ongoing work to optimize the reasoning cycle and to improve upon the language. We see [15], which presents an architecture to program embedded agents using Javino [16] as well as incorporating percept filtering from [10]. [17] uses Jason to control Unmanned Aerial Vehicle (UAV)s, while [18] controls unmanned ground vehicles with Jason. We also see that [19] describes using Jason to drive a robot to deliver mail, using Robot Operating System (ROS). We will look at this project in more depth in chapter 7, as we use it as a case study to compare with our theoretical model.
3.4 Summary

We have seen that work has been done with regards to analyzing the execution time of the Jason reasoning cycle. We have looked at works that validate our choice of analyzing Jason, as opposed to other languages. We have seen that how the percepts perceived by an agent affect the execution time and how limiting the number of percepts available can reduce the execution time. Also, we have seen that caching queries for beliefs and goals have an impact on reducing the execution time. We have even seen work done to circumvent the Jason reasoning cycle and attempt to implement concurrency into AgentSpeak(L), resulting in a new version of AgentSpeak. However, we have not found any works that look at the execution time of the Jason reasoning cycle as a whole. Knowing that Jason is currently being used in the realm of autonomous robotics leads us to believe that an in-depth analysis of the execution time of the Jason reasoning cycle will provide valuable insight for those currently using Jason in embedded environments, as well as those who would want to use Jason in the future. This work attempts to fill this hole, by analyzing the execution time of the reasoning cycle. In the next section, we will describe in detail how we plan to go about this analysis.
Chapter 4

Methodology

To complete a performance analysis of the Jason reasoning cycle, we will have to do three things:

1. determine the time complexity of the reasoning cycle code to determine what we should be looking for and to identify what parameters warrant the closest look;

2. use that analysis to determine how to go about designing agents and environments to exploit the parameters we want to look at and generate a model of the execution time;

3. look at case studies to validate our model.

Once we have completed these three steps, we will be able to provide a prediction of how long an iteration of the reasoning cycle will take on our testing systems (see table 6.2 for a summary of specifications of our testing systems). Following the same procedure on any other hardware configuration, developers will be able to ascertain whether Jason is an appropriate language for their development.
4.1 Analysis of Reasoning Cycle

To begin our performance analysis, we must first determine the time complexity of the Jason reasoning cycle and its data structures. We do this for two reasons:

1. To determine what parameters we should be looking at when developing our model. Having a narrower view will allow us to determine what parameters are most important, in terms of the impact they have on the scalability of the execution time of the algorithms that depend on them. A well understood time complexity will allow for us to eliminate parameters that do not have a lot of impact and focus on the ones that do.

2. To validate that our model is generally sound. Developing our model will act as a check that our analysis of the time complexity is correct, and our analysis will inform us as to whether our model is appropriate. We expect our analysis to match the results we experimentally get and, if they do not, we should go back to our analysis and either refine our analysis or revise our experiment. Our analysis and the results of our experimentation should match.

To this end, we will present pseudo-code that we have developed that has been generalized away from Jason and is expected to be valid for any interpreter based on AgentSpeak(L). Using this pseudo-code, we will present an analysis of the time complexity in chapter 5, which again, we expect to be valid for any AgentSpeak(L) based interpreter. For our analysis, we will use “Big O notation” and investigate the data structures used by Jason internally, and the algorithms that operate on them. We will use what we find here to drive our design for our experimentation when generating our model in chapter 6. This means that we will use the time complexity as a means
to determine what we should change in our agents and to give us an idea as to how the execution time will change in response to our modification of the agent.

4.2 Model Development

Using the time complexity that we will create in chapter 5, we will design agents and environments that will vary the parameters that affect the execution time of the Jason reasoning cycle. In general, our procedure will be as follows:

- generate customized agent files and any environment files needed to support the agents;

- using stand-alone Java (i.e: not running the agent in conjunction with a debugger), execute the agent;

- use JProfiler to record a snapshot every time the function responsible for starting the execution of the reasoning cycle is called;

- collect the profiler data and turn it into graph-able data by transforming the collected data into execution time of each reasoning cycle;

- graph the data and, through the use of the coefficient of determination ($R^2$) determine an appropriate curve of best fit to approximate an equation for the execution time.

In general, we will run our agent five times and take 50 snapshots, totaling 250 snapshots for each parameter value. We want the data we gather from our experimentation in chapter 6 to conform to our time complexity in chapter 5. If they do not, we take this as a sign that either our designed experiment is flawed and should
be corrected, or that our analysis is incorrect and should be studied again. In order to be confident in our curve of best fit, we will fit the following three different base equations:

1. \( f(x) = ax + b \)
2. \( f(x) = ax^2 + bx + c \)
3. \( f(x) = a^{(bx+c)} \)

We will then calculate the coefficient of determination \((R^2)\) of each of these by using the following formulas:

\[
SS_{\text{actual}} = \sum_i (y_i - y_{\text{median}})^2
\]

\[
SS_{\text{curve}} = \sum_i (y_i - f(x))^2
\]

\[
R^2 = 1 - \frac{SS_{\text{curve}}}{SS_{\text{actual}}}
\]

Where:

- \( y_i \): Measured data point.
- \( y_{\text{median}} \): Median data point for all measured data points.
- \( f(x) \): Function that represents curve of best fit.

We will expect that the coefficient of determination associated with the curve of best fit that matches the form of the time complexity will be the higher than the other two. This is due to the fact that this value represents “[...] the amount of variation in the data explained by the regression model” [20]. Note that we will only perform this calculation for results gathered from the laptop, as we expect the trend to hold for any hardware. Once the data and the analysis agree, we can look at case studies in
chapter 7 to determine overall if our model is sound. We expect that our model will perform adequately, but as this is the first iteration of this type of model, we expect that there will be areas where the model does not perform exceptionally.

4.3 Case Study

Using the model that we will develop in chapter 6, we will calculate our estimated execution time of three different agents developed by others. This will allow us to remove our own bias from the case study, as well as to get an example of “real world” agents. The three agents we will analyze are:

- domestic robot agent — included with the Jason package as an example agent. The github repository for this project can be found at [21];

- grid world agent — provided by the Network Management and Artificial Intelligence lab at Carleton University [22]. This agent will navigate through a simple grid world. The github repository for this project can be found at [23];

- mail delivery robot — provided by the Network Management and Artificial Intelligence lab at Carleton University [22]. This robot runs ROS and uses a Jason agent to reason. It represents an example of a real autonomous robot as it does not run in a simulated environment. The github repository for this project can be found at [24].

We will compare our estimated execution time with measured execution times from these agents to see how appropriate the model that we will develop in chapter 6 is.
4.4 Summary

This chapter has outlined how we will go about developing a model for the execution time of the Jason reasoning cycle. We will perform an analysis of the time complexity of the source code which will inform how we design agents to gather execution time data. We will then design agents to exploit the parameters we find during the analysis, which should verify our analysis. Using the data we gather; we can develop a model which we will use to predict execution times of agents not written by ourselves. These case studies should verify our data collection and model generation. The next chapter will present our analysis of the time complexity.
Chapter 5

Analysis

To develop an accurate model of the execution time of the reasoning cycle we will start with an analysis of the Jason source code. We will first look at the Agent class and the other classes that are important to it — this will help us understand how Jason views the agent and what its role is within the greater reasoning cycle. Second, we will investigate the reasoning cycle sequence to ensure that we properly understand what happens during an iteration of the cycle. We will then do an analysis of the time complexity on the steps that make up the reasoning cycle to determine what portions of the reasoning cycle should be looked at.

5.1 Agent Implementation

Having seen an overview of the semantics of AgentSpeak(L) in section 2.2 as well as how Jason builds upon that base in section 2.3, we can now look at how Jason implements an agent. How these classes are designed will provide us insight as to the organization of the code, the choice of data structures and why the reasoning cycle runs as it does.
Figure 5.1: Agent Class Diagram [2]
5.1.1 Agent

As shown in figure 5.1, an agent consists of a BeliefBase, a PlanLibrary, and a TransitionSystem. It contains a list of initial beliefs and initial goals (stored this way so that the belief base can be populated once the transition system is ready). This class also defines the default behaviour for the belief revision function, and the belief update function (a lightweight version of belief revision, in which beliefs are added and deleted, but no additional processing is done).

It also allows for customized agents to be created, inheriting from the generic Agent class. The selection functions \texttt{selectAction()}, \texttt{selectEvent()}, \texttt{selectIntention()}, \texttt{selectMessage()}, and \texttt{selectOption()} have a default behaviour of taking the first item in their respective queues and selecting them, with no regard for any priority. Writing a customized Agent class allows for the overriding of these functions to design more complex selection functions. The belief update function (\texttt{buf()}) and belief revision function (\texttt{brf()}) can also be overridden, in the case that specific functionality is desired above what is already provided.

5.1.2 Belief Base

The DefaultBeliefBase class inherits from the abstract BeliefBase class and is the default implementation of a belief base, providing a place to store and manipulate beliefs. The DefaultBeliefBase implements two Collection objects, a LinkedBlockingDeque (thread-safe double ended linked list) and a ConcurrentHashMap, which both contain the exact same data. Using two different Collections in this way allows the belief base to preserve the order of the beliefs, as they are added to the belief base (via the LinkedBlockingDeque) but also exploit the faster search of a ConcurrentHashMap.
5.1.3 Plan Library

The PlanLibrary class represents the set of plans that can be used by an agent. It stores the plans as a list, which allows for maintaining the order of the plans as written by the developer. It is also responsible for determining candidate plans (plans which match a triggering event).

5.1.4 Agent Architecture

This class is the interface between the agent and the environment. It can be customized and includes `perceive()`, which is how an agent interfaces with any sensors it might have and `act()`, which executes an intention. It also contains `sendMsg()`, `broadcast()`, and `checkMail()`, which allow an agent to communicate with other agents.

5.1.5 Circumstance

A place to store and coordinate a set of intentions (stored as a stack of partially instantiated plans), a set of events and a set of actions. It also contains the agent’s mailbox and a list of events that are to be processed. This class acts as an area where actions common to all agents, which should not be customized by a developer, are located. It is tightly coupled with the Agent class.

5.1.6 Transition System

This class is the implementation of the underlying states that drive the reasoning cycle. It is responsible for kicking off the reasoning cycle and transitioning states, based on the execution of each state. Figure 5.2 shows an overview of the possible
transitions within this class.

5.1.7 Selection Functions

As mentioned in section 2.2, there are three selection functions that play a major part in the reasoning cycle. The selectEvent() function from step 5 will take a set of events and pick one to deliberate upon. In step 8, the selectOption() will take a set of options and pick one to add to the intention stack. Finally, selectIntention() will take the set of intentions that exist within the intention stack and pick one to act upon. By default, all three of these functions will select the first entry. However, Jason provides the ability for developers to override these functions and write their own selection criteria. For example, one could introduce some sort of priority and only select the highest priority object. For our purposes, we will limit ourselves to the default implementation of these functions.

5.1.8 Discussion

We have seen how the Agent class is implemented, as well as how it interacts with the classes closest to it. This provides some understanding as to how the reasoning
cycle is structured. We will be able to use this to help understand the sequence of the reasoning cycle, which we will look at next.

5.2 Sequence Analysis of the Reasoning Cycle

Building upon the information presented in the previous section, we will be able to look at how the reasoning cycle is constructed. We will present two different diagrams (an activity diagram and a sequence diagram) for each of the three phases. The *sense* phase, which is responsible for gathering input from internal and external sources. The *deliberate* phase, which will select a plan to act upon based on its own beliefs, desires, and intentions. Finally, the *act* phase will select an intention to act upon and execute it. The activity diagram will focus on state transitions and highlight the idea that not all transitions through the reasoning cycle are identical. The sequence diagram will focus on the order in which functions are called and shows a lower-level view of when the functions are called.

5.2.1 Sense

The *sense* phase of the reasoning cycle is predominately concerned with gathering data from outside sources (whether that be sensors, or other agents), and consolidating that into its own belief base. Figure 5.3 shows the sequence of the *sense* phase and figure 5.4 shows the activity diagram.

5.2.1.1 Step 1 - Gather Perceptions

The first step of the reasoning cycle to be executed is the *perceive()* function. This function will return a list of perceptions (common to all agents and specific to the
Figure 5.3: The Jason Reasoning Cycle Sequence Diagram - Sense
agent being run) gathered from the environment. By default, this data is assumed to be complete and accurate. It is then provided to the buf() function.

5.2.1.2 Step 2 - Update Belief Base

Taking a list of percepts from the perceive function, the default belief base functionality (implemented in buf()) will add items that have been perceived (and do not already exist) and remove things that do exist in the belief base but have not been perceived. Once completed, the reasoning cycle will check its mailbox to see if messages have been received from other agents.

5.2.1.3 Step 3 - Check Mailbox

Next, checkMail() gets called to transfer any messages that might exist in the local mailbox to the architecture’s circumstance object.
5.2.1.4 Throttling of Perception

We should note at this point, that step 1, 2 and 3 can be throttled by setting the \textit{nrcbp} (Number of Reasoning Cycles Before Revision) variable, which will skip the gathering of percepts, updating the belief base and checking the mailbox until a specified number of reasoning cycles have been run. This means that it is possible to artificially limit the frequency in which these three aspects of the BDI engine are called, with respect to the rest of the engine.

5.2.1.5 Step 4 - Checking if a Message is “Socially Acceptable”

At this point, the transition system initializes its internal states. It sets its state to be \textit{StartRC} (Start Reasoning Cycle). Currently, the only thing that happens is the internal state always gets changed to \textit{SelEv} (Select Event), a message is selected to process (using \textit{selectMessage()}, which can be customized like other selection functions). The selected message is passed to \textit{socAcc()} to determine if it is an appropriate message to process and will either be processed or dropped.

Regardless of whether the message is processed, the \textit{sense} portion of the reasoning cycle is concluded and we move onto the \textit{deliberate} phase.

5.2.2 Deliberate

This phase will do the majority of the choosing what should be done based on what has changed during the \textit{sense} phase. Figure 5.5 shows the sequence diagram of the \textit{deliberate} phase of the reasoning cycle, figure 5.6 shows the activity diagram.

Once the \textit{deliberate} phase of the reasoning cycle starts, the state gets explicitly set to \textit{SelEv}. After that, \texttt{applySemanticRuleDeliberate()} is called to iterate over the
Figure 5.5: The Jason Reasoning Cycle Sequence Diagram - Deliberate
Figure 5.6: The Jason Reasoning Cycle Activity Diagram - Deliberate

[Diagram of the Jason Reasoning Cycle Activity Diagram - Deliberate]
steps of this phase. This function will be repeatedly called until the transition system moves to the *ProcAct* (Proceed to Act) state.

### 5.2.2.1 Step 5 - Selecting an Event to Deliberate Upon

When called, `applySelEv()` will check to see if there are any atomic intentions (which represent an intention to be run immediately). If so, the state will be set to *ProcAct* and the rest of the *deliberate* phase will be skipped. Otherwise, a check to see if an atomic event exists (which represents an event which must be run immediately). If this event exists, then the state will be set to *RelPl* (Relevant Plan) and the rest of the function will be skipped. If there are no atomic intentions or atomic events, then the event selection function (`selectEvent()`) will be called. This function can be overridden by a customized Agent class if desired, but by default will pick the first event out of a list of events provided to it.

Once an event has been selected, the reasoning cycle checks to see if a customized `selectOption()` function exists. If it doesn’t, then the reasoning cycle will execute a shortcut, designed to potentially cut down the execution time. This will be discussed in section 5.2.2.5. If the customized `selectOption()` function exists, then the state gets set to *RelPl*.

### 5.2.2.2 Step 6 - Selecting a List of Relevant Plans

After an event is selected, `applyRelPl()` will be called to gather relevant plans (plans that have a triggering event that unify with the event that was just selected) via `relevantPlans()`. Note that unification is the process by which variables are bound to values. In this case, if the triggering event unifies with the selected event, then an option (a plan and the unifier that makes it relevant) is created. If `relevantPlans()`
results in an empty list (i.e: no plans were relevant to the selected event), then one of two things will happen: the state gets set to \textit{ProcAct} (Proceed to Act), as there is nothing that can be deliberated upon, or the state gets set back to \textit{SelEv}, and \texttt{applySelEv()} will be called again to select a new event (as the original event has been deemed irrelevant). If \texttt{relevantPlans()} results in a list with options in it, then the state is set to \textit{ApplPl} (Applicable Plan) and the list is passed to the next step to see which of these options are also applicable.

**5.2.2.3 Step 7 - Selecting a List of Applicable Plans**

Having found a list of relevant plans, the transition system calls \texttt{applyApplPl()}, which will call \texttt{applicablePlans()} to determine which of the relevant plans are applicable. This is done by looking to see if the context of the plan is a logical consequence of the agent’s belief base and the unifier that made the plan relevant (i.e: the unified context, when substituted into the belief base results in a belief base that is rational). If this results in an empty list of applicable plans then, like step 6, the reasoning cycle will determine if it must be restarted, or a new event can be selected instead. Otherwise, the state will be set to \textit{SelAppl} (Select Applicable Plan) and the list of applicable plans will be passed to the next step to see which one will be selected.

**5.2.2.4 Step 8 - Selecting an Option**

After the list of applicable plans is found, \texttt{applySelAppl()} will be called, which will call \texttt{selectOption()} to determine which of these options will be selected. By default, the first option in this list will be picked, but this function can be overridden by a customized agent class. If an option is found, then the state gets set to \textit{AddIm} (Add Intended Means), otherwise the rest of the \textit{deliberate} phase is skipped and the state
gets set to \textit{ProcAct} (Proceed to Act).

Once in the \textit{AddIm} state, \texttt{applyAddIM()} is called, and adds the selected option to the intention stack of the circumstance object. The state then moves to \textit{ProcAct}.

\subsection*{5.2.2.5 Optimizing the Deliberate Phase}

As mentioned in section 5.2.2.1, if the default \texttt{selectOption()} function is used, then the reasoning cycle will employ a potential shortcut. Knowing that only the first applicable plan will be selected (regardless of how many have been found), the reasoning cycle will check to see if a given plan is relevant and, if so, it will then check to see if it is applicable. If the plan is relevant and applicable, it will be selected, and no more plans will be checked. Otherwise, the next plan will be checked to see if it is relevant and applicable. This shortcut will combine step 6, 7 and 8 into one step with the hope that an applicable plan will be determined quicker, and less processing will be required. If no relevant and applicable plan is found, the reasoning cycle will determine if the cycle must be restarted or if a new event can be selected. Otherwise, the state is set to \textit{AddIm} which, as in step 8, will call \texttt{applyAddIM()} to add the selected option to the intention stack then sets the state to \textit{ProcAct}.

Regardless of the route taken to get through the \textit{deliberate} phase, the \textit{ProcAct} state represents the beginning of the final phase of the reasoning cycle — \textit{act}.

\subsection*{5.2.3 Act}

Having completed the \textit{deliberate} phase of the reasoning cycle, the transition system explicitly sets its state to \textit{ProcAct} and will call \texttt{applySemanticRuleAct()} until the transition system reaches the beginning state again: \textit{StartRC} (Start Reasoning Cycle). This starts the \textit{act} phase of the reasoning cycle (seen in figure 5.7 as a sequence
Figure 5.7: The Jason Reasoning Cycle Sequence Diagram - Act

```
loop

all

apply(SemanticRuleAct)

app(ProcAct)

feedbackAction = hasFeedbackAction

result = removePendingAction(action.getIntent().getAct())

apply SetInt()

hasRunningIntent = hasRunningIntent

runningIntentList = getRunningIntentions

selectIntent = selectAction(runningIntentList)

if stepAct == ProcAct
    apply SetInt()
if stepAct == SetInt
    apply ExecInt

addActValue

all

app(Proc)

planBodyType = achieve

planBodyType = buffer

planBodyType = buffer

apply Change(selectedIntent)

dropRunningIntent(selectedIntent)

if selectedAction is finished
    if stepAct == Change
while stepAct != StopPC
```
Figure 5.8: The Jason Reasoning Cycle Activity Diagram - Act
diagram, and in figure 5.8 as an activity diagram).

5.2.3.1 Step 9 - Selecting an Intention

To start, as the transition system is currently in the state \textit{ProcAct}, \texttt{applyProcAct()} will be called to set the state to \textit{SelInt} (Select Intention) and perform some minimal processing that is irrelevant to our analysis is done here. After, \texttt{applySelInt()} gets called to select an intention to execute, setting the state is set to \textit{ExecInt} (Execute Intention). If an atomic intention exists, that intention is selected. Otherwise, \texttt{selectIntention()} is called to pick an intention to be executed. By default, this function picks the first intention in the list, but can be overridden by a customized agent. If no intention can be selected, the state is set to \textit{StartRC} and the reasoning cycle is restarted.

5.2.3.2 Step 10 - Executing an Intention

Finally \texttt{applyExecInt()} is called to set the state to \textit{ClrInt} (Clear Intention) and execute the intention that has been selected. It should be noted that if a belief is added or deleted as a result of this intention, \texttt{brf()} is called to revise the belief base, which by default, will just add or delete the belief from the belief base. The \textit{ClrInt} state consists of only setting the state to \textit{StartRC} and clearing out the selected intention that was just executed.

5.2.4 Discussion

In this section we have presented an in-depth look at how the reasoning cycle executes. We would like to highlight the fact that not every execution of the reasoning cycle will be identical, as we have seen that a given iteration of the reasoning cycle can be
terminated early. This means that we must be aware that the execution time will fluctuate as different combinations of functions are called and will potentially need to account for this in our model.

5.3 Analysis of the Time Complexity of the Reasoning Cycle

A large variety of Java data structures are used to store the data needed to run the reasoning cycle. This allows Jason to leverage well defined and well understood algorithms when storing and manipulating beliefs, perceptions, etc. To get an idea of the time complexity of the reasoning cycle, we will need to look in depth at the data structures used and the scalability of the operations performed on these data structures, as their size increases. The number of elements contained in these data structures will affect the parameters of the analysis of the time complexity in this chapter and of the performance models in the next chapter. As the Jason implementation of the reasoning cycle has ten steps, which we have presented in the previous section, we will also break down our analysis into the same ten steps. We should note that the Jason framework provides a lot of flexibility in the form of being able to extend classes and override default implementation of some of the steps of the reasoning cycle. For our analysis, we will stick to the default behaviour, which means we will be using a default selectOption() function. Recall that this means we will not be executing steps 6, 7, and 8, instead executing the alternate function described in section 5.2.2.5. While we are focusing on Jason for this work, we have developed generic pseudo-code that in applicable for AgentSpeak(L) interpreters in general. It is our belief that any performance or scalability bottleneck found in our analysis in this section could be
found in not only other AgentSpeak(L) interpreters but possibly in other BDI based agent languages which follow a sense - deliberate - act cycle. We also expect this pseudo-code to be useful in re-implementing AgentSpeak(L) interpreter algorithms in a different language in case Java no longer proves a suitable language.

5.3.1 Step 1 - Perceive

Algorithm 1: Perceive

\begin{algorithm}
\begin{algorithmic}
\State **Input:** \( P_{\text{global}} \) : Set of perceptions available to all Agents
\State **Input:** \( P_{\text{agent}} \) : Set of perceptions available only to an Agent
\State **Output:** \( P \) : Set of perceptions that have been perceived by an Agent
\State \( P \leftarrow P_{\text{global}} \cup P_{\text{agent}} \)
\State return \( P \)
\end{algorithmic}
\end{algorithm}

The `perceive()` function returns a list of percepts gathered from the environment which is a combination of global percepts that are perceived by all agents and percepts that are perceived by just the specific agent. Looking at algorithm 1, we see that in general, updating the perceptions in this step is a simple union of two sets.

Jason stores both of these sets as Arrays of Literals. For our case, the union of these two sets will be either \( O(|P_{\text{global}}|) \) or \( O(|P_{\text{agent}}|) \). This means that the time complexity for this function will be linear, given by \( \text{MAX}(O(|P_{\text{global}}|), O(|P_{\text{agent}}|)) \). This set will be directly provided to the belief update function, seen in step 2.

5.3.2 Step 2 - Belief Update Function

The belief update function shown in algorithm 2 does two things; deletes perceptions that are no longer perceived and adds perceptions that have been perceived, as given to it by the `perceive()` function seen in step 1.
Algorithm 2: Belief Update Function

**Input:** \( P \) : Set of perceptions that have been perceived this reasoning cycle

**Input:** \( P_{\text{belief}} \) : Set of beliefs that exist within the belief base from the previous reasoning cycle

**Input:** \( P_{\text{percept}} \) : Set of perceptions that exist within the belief base from the previous reasoning cycle

**Output:** \( n \) : The number of additions and deletions that have taken place

**Output:** \( P_{\text{belief}} \) : Set of updated beliefs that exist within the belief base as of this reasoning cycle

**Output:** \( P_{\text{percept}} \) : Set of updated perceptions that exist within the belief base from the previous reasoning cycle

foreach \( p : \text{perception} \) in \( P_{\text{percept}} \) do

if \( p \notin P \) then

\( P_{\text{belief}} \leftarrow P_{\text{belief}} \setminus \{p\} \)

\( P_{\text{percept}} \leftarrow P_{\text{percept}} \setminus \{p\} \)

\( n++ \)

end

end

foreach \( p : \text{perception} \) in \( P \) do

\( P_{\text{belief}} \leftarrow P_{\text{belief}} \cup \{p\} \)

\( P_{\text{percept}} \leftarrow P_{\text{percept}} \cup \{p\} \)

\( n++ \)

end

return \((n, P_{\text{belief}})\)
In order to understand the Jason implementation a little more clearly, we will look at the second portion of the algorithm first (the addition of percepts), which will provide some clarity as to how the percept is stored within the belief base and consequently why the deletion happens the way it does.

To add the percepts to the belief base, the list of perceptions will be iterated over, taking $O(|P|)$ time, and will be added to the belief base in two locations, once in a HashSet of perceptions and once in the belief base itself. Within the belief base, this perception is stored in two forms; a LinkedBlockingDeque (in order to maintain the order of the beliefs) as well as a HashMap (in order to utilize a faster search). The “percept” annotation will be added to the percept to represent where the belief came from.

To delete from the belief base, the Jason implementation of this algorithm will first iterate over the HashSet of perceptions, taking $O(|P_{belief}|)$ time and determine if the perception needs to be deleted by comparing it to the set of perceptions passed into the belief update function ($P$). If it needs to be deleted, then it deletes the perception from the HashSet being iterated over and from the belief base. This means that, because a belief is stored in two forms, two delete actions must happen: a deletion from the HashMap as well as a deletion from the LinkedBlockingDeque, which, because it needs to search for the specific perception to delete, takes $O(|P_{belief}|)$ time.

Overall, this function would be expected to take a worst case of $O(|P_{belief}|^2)$ time. This is due to the deletion of percepts that are no longer perceived as the set $P_{belief}$ must be iterated over once to see if the percept is still in $P$. If a given percept is found to no longer be in $P$, then it must be deleted from $P_{belief}$, which requires a second nested iteration over $P_{belief}$ as the percepts here are stored as a LinkedBlockingDeque.
5.3.3 Step 3 - Check Mail

Algorithm 3: Check Mail

| Input: | M_{agent} : Set of messages that have been sent to an Agent |
| Input: | M_{mail} : Set of messages that is stored within an Agent |
| Output: | M_{mail} : Set of messages updated with the relevant messages from M_{agent} |

\[ M_{mail} \leftarrow M_{mail} \cup \{M_{agent}\} \]

\text{return } M_{mail}

The check mail function simply moves messages from the messaging implementation to the agent implementation. By default, Jason implements this by moving messages from one ConcurrentLinkedQueue to a different Queue taking O(1) time. Any messages that exist in the agent’s queue will be examined by the socAcc() function shown in step 4.

5.3.4 Step 4 - Determining if a Message is “Socially Acceptable”

Algorithm 4: Socially Acceptable

| Input: | m : Message to be checked, which has been sent to an Agent |
| Input: | F_n : Set of n predicates, taking a message as input, such that F_j : m \rightarrow true, false |
| Output: | s : Whether or not Message m is appropriate to be acted upon |

\[ s \leftarrow \bigwedge_{j=1}^{n} F_j(m) \]

\text{return } s

To determine if a message is “socially acceptable”, (per algorithm 4), a message is selected from the queue of messages generated by the checkMail() function (step 3) and run through a set of provided predicates (represented by F_n) to determine if the
message is appropriate to be acted upon. In general, this function will have a time complexity of \( \text{MAX}(O(F_1), O(F_2), \ldots, O(F_n)) \).

Jason provides a default implementation of this and considers all messages to be acceptable and, as such, has a time complexity of \( O(1) \).

5.3.5 Step 5 - Selecting an Event

Algorithm 5: Select Event

| Input: \( E \) : Set of events that can potentially be selected this reasoning cycle |
| Input: \( f \) : Selection function, such that \( f : E \rightarrow \epsilon \) |
| Output: \( \epsilon \) : Event that has been selected to act upon this reasoning cycle |

\[
\epsilon \leftarrow f(E) \\
\text{return } \epsilon
\]

The select event function shown in algorithm 5 takes a set of events and selects one event to be acted upon. As such, the time complexity of this function is expected to be dependent on the selection function \( f \).

The Jason implementation will, by default, take the first event present in \( E \), which will take \( O(1) \) time. The selected event will be passed to step 6 and used to generate relevant plans.

5.3.6 Step 6 - Collecting Relevant Plans

Algorithm 6 takes the entire plan library and filters it down to a smaller set of plans that unify with the provided triggering event (selected in step 5). This smaller set is stored as a set of options, which represents a plan as well as the result of the unification of the plan and triggering event. This is expected to have a time complexity of \( O(|\Pi| \times O(\text{unifies}())) \).
Algorithm 6: Relevant Plans

Input: $t_e$: Triggering event
Input: $\Pi$: Set of plans that are available to an Agent
Output: $Opt_{relevant}$: Set of relevant options (Note that an option is a plan, as well as the unifier that makes it relevant and applicable)

foreach $\pi : plan$ in $\Pi$ do
  $u \leftarrow \pi.unifies(t_e)$
  if $u \notin \emptyset$ then
    $Opt_{relevant} \leftarrow Opt_{relevant} \cup \{(\pi, u)\}$
  end
end
return $Opt_{relevant}$

Algorithm 7: Unifies

Input: $t_1$: Literal to unify
Input: $t_2$: Literal to unify
Output: $u$: Unifier that unifies $t_1$ and $t_2$

if $t_1.isVariable()$ && $t_2.isVariable()$ then
  $vt_1 \leftarrow t_1.variable()$
  $vt_2 \leftarrow t_2.variable()$
  return $vt_1.unifies(vt_2)$
end
if $t_1.isVariable()$ then
  $vt_1 \leftarrow t_1.variable()$
  return $vt_1$
end
if $t_2.isVariable()$ then
  $vt_2 \leftarrow t_2.variable()$
  return $vt_2$
end
return $\emptyset$
The Jason implementation of this step happens in two parts. First, the plan library is narrowed down to a set of candidate plans by getting plans that match the arity of the triggering event. Due to the plan library being stored as a HashMap, this is expected to take $O(1)$ time, due to the entire bin being retrieved. This will result in a set $\Pi_{\text{candidate}} : |\Pi_{\text{candidate}}| \leq |\Pi|$. Once this set has been generated, it is iterated over while being unified with the trigger to determine if the plan is relevant. This will take $O(|\Pi_{\text{candidate}}|)$. It should be noted, that in the unification algorithm (algorithm 7), it is possible for the time complexity to be $O(2^n)$, where $n$ is the number of nested variables that make up a complex literal. In our case, we are limiting ourselves to only one level since our unification is limited to triggering events and beliefs. This leaves us with a constant time complexity. If the plan is relevant, then it will be placed into a second ArrayList, which is presented as the list of relevant plans. In general, this function will have a time complexity of $O(|\Pi_{\text{candidate}}| \times 2^n)$, however due to Jason limiting its unification to triggering events and beliefs (setting $n$ to 1), the time complexity of this step is $O(|\Pi_{\text{candidate}}|)$. This list of relevant plans is then provided to step 7, which will further narrow the list down.

### 5.3.7 Step 7 - Collecting Applicable Plans

The applicable plan function shown in algorithm 8 takes the set of options provided in step 6 and generates a set of options based on the logical consequences of: the plan, the unifier and the belief base of the agent. This function is expected to have a time complexity of $O(|\Pi_{\text{relevant}}| \times O(\text{logicalConsequences}))$. The logical consequences (see algorithm 9), however, ends up being more complicated. This function will iterate over the belief base and attempt to unify the provided Literal with each belief. In the case of more complicated beliefs, we see the following:
Algorithm 8: Applicable Plans

**Input:** $\text{Opt}_{\text{relevant}}$ : Set of relevant options (Note that an option is a plan, as well as the unifier that makes it relevant and applicable)

**Input:** $B$ : Set of beliefs which comprise the belief base of an Agent

**Output:** $\text{Opt}_{\text{applicable}}$ : Set of options that are logical consequences of the belief base

```
foreach $\text{opt} : \text{option in } \text{Opt}_{\text{relevant}}$ do
    $\pi \leftarrow \text{opt.plan()}$
    $c \leftarrow \pi.\text{context()}$
    if $c \in \emptyset$ then
        $\text{Opt}_{\text{applicable}} \leftarrow \text{Opt}_{\text{applicable}} \cup \{\text{opt}\}$
    else
        /* See algorithm 9 for logicalConsequences */
        $\text{LC} \leftarrow \text{logicalConsequences}(c, \text{opt.unifier()}, B)$
        foreach $u : \text{unifier in } \text{LC}$ do
            $\text{opt}_{lc} = (\pi, u)$
            $\text{Opt}_{\text{applicable}} \leftarrow \text{Opt}_{\text{applicable}} \cup \{\text{opt}_{lc}\}$
        end
    end
end
return $\text{Opt}_{\text{applicable}}$
```
Algorithm 9: Logical Consequences

Input: l : Literal to calculate the logical consequences of
Input: u : Unifier to get the logical consequences for
Input: B : Set of beliefs contained within the Belief Base of an Agent
Output: LC : Set of logical beliefs from the belief base for unifier u

foreach b : belief in B do
  if b.equals(l) then
    B_candidate ← B_candidate ∪ {b}
  end
end

foreach b : belief in B_candidate do
  if b.isRule() then
    /* See algorithm 7 for unifies */
    u_rule ← u.unifies(b)
    if u_rule /∈ ∅ then
      /* See algorithm 9 for logicalConsequences */
      LC_rule ← logicalConsequences(b, u_rule, B_candidate)
      foreach l_rule : logical belief in LC_rule do
        B_candidate ← B_candidate ∪ {l_rule}
      end
    end
  else if b.hasAnnotations() then
    B_annotations ← b.annotations()
    foreach b_annotations : belief in B_annotations do
      B_candidate ← B_candidate ∪ {b_annotations}
    end
  else
    /* See algorithm 7 for unifies */
    u_lc ← l.unifies(b)
    if u_lc /∈ ∅ then
      LC ← LC ∪ {u_lc}
    end
  end
end

return LC
• If the belief is a rule, it will get the logical consequences of that rule (this is recursive, so it can be exponential).

• If the belief has annotations, those annotations are added to the Iterator (this is linear).

In general, it is expected that this function will have a time complexity of $O(|B|^n)$, where $n$ is the number of belief rules. Note that this is potentially a major bottleneck as this is exponential time complexity. The more belief rules that exist for an agent (which are a useful tool and are expected to be commonly used), the longer the execution time is expected to be.

Overall, the Jason implementation of these functions will have a time complexity of $O(|Opt_{relevant}| \times |B|^n)$.

This set of options will be provided to step 8 so that a single option can be selected.

5.3.8 Step 8 - Selecting an Option

**Algorithm 10: Select Option**

**Input:** $O_{applicable}$ : Set of applicable options available to an Agent  
**Input:** $f$ : Selection function, such that $f : O \to o$  
**Output:** $o$ : The option that has been selected to be acted upon this reasoning cycle  

\[ o \leftarrow f(O_{\text{applicable}}) \]

\[ \text{return } o \]

The select option function takes a set of options and selects one option to be acted upon. As such, the time complexity of this function is expected to be dependent on the selection function.

By default, the first option in $Opt_{\text{applicable}}$ is selected. This takes $O(1)$ time, as no array manipulation is occurring.
5.3.9 Step 9 - Selecting an Intention

Algorithm 11: Select Intention

Input: $I$ : Set of intentions available to an Agent

Input: $f$ : Selection function, such that $f : I \rightarrow \iota$

Output: $\iota$ : The intention that has been selected to be acted upon this reasoning cycle

$\iota \leftarrow f(I)$

return $\iota$

The intention selection function takes a set of intentions and selects one intention to be acted upon. As such, the time complexity of this function is expected to be dependent on the selection function.

By default, the Jason implementation will take the first element of $I$, which has a time complexity of $O(1)$. This intention will be provided to step 10 to be executed.

5.3.10 Step 10 - Executing the Intention

The execution of the intention that was selected as part of step 9 is presented in algorithm 12. This function will take the intention that has been selected and perform different actions based on what the selected intention is. In the case of the intention being an action, the action will simply be executed and removed from the set of available intentions. The expected time complexity of this is expected to be $O(f_i)$, which represents the complexity of the action itself. If the intention is checking a constraint, then the time complexity will be similar to the complexity presented as part of the logical consequences function in algorithm 9 ($O(|B|^n)$). Finally, if beliefs are to be added or deleted, then belief revision will have to be run. By default, Jason does not implement any sort of real belief revision, leaving that up to the programmer to implement. It should be noted that, in general, belief revision is considered to be
Algorithm 12: Apply and Execute Intention

Input: \( \iota \): Current intention
Input: \( B \): Set of beliefs which comprise the belief base of an Agent
Input: \( I \): Set of intentions available to an Agent
Output: \( I \): Set of updated intentions available with either intention \( \iota \) executed and removed from \( I \), or modified and added back into \( I \)

switch \( \iota \) do
  case \emph{action} do
    execute \( \iota \).action()
    \( I \leftarrow I \setminus \{\iota\} \)
  end
  case \emph{constraint} do
    \( c \leftarrow \iota\text{\_current}\text{\_context}() \)
    \( u \leftarrow \iota\text{\_current}\text{\_unifier}() \)
    /* See algorithm 9 for logicalConsequences */
    \( LC \leftarrow \text{logicalConsequences}(c, u, B) \)
    if \( LC \notin \emptyset \) then
      \( \iota\text{\_unifier} = \{l_{c_1}\}; l_{c_1} \in LC \)
      \( I \leftarrow I \cup \{\iota\} \)
    end
  end
  case \emph{delete belief and/or add belief} do
    \( B_{\text{add}} \leftarrow \text{Beliefs to add} \)
    \( B_{\text{del}} \leftarrow \text{Beliefs to delete} \)
    beliefRevisionFunction()
    \( I \leftarrow I \setminus \{\iota\} \)
  end
end
return \( I \)
NP-complete [25]. That said, [26] proposes ways to utilize the features of Jason that could potentially result in a belief revision function that could have a time complexity of $O(n \log(n) + |B| \times |\Pi|)$, where $n$ represents the number of literals within the belief base.

### 5.3.11 Step X - Combine Step 6 through Step 8 into one action

The Jason implementation also provides a bit of a shortcut, in that step 6, 7, and 8 can be skipped if the `selectOption()` function has not been overridden by a custom Agent class. Shown in algorithm 13, instead of generating entire sets of relevant and applicable plans, the entire set of plans is converted into a set of candidate plans (plans that have the same arity as the triggering event). This set is then iterated over to find the first plan that is both relevant and applicable. We see that this operation still has a worst-case time complexity of $O(|Opt_{relevant}| \times |B|^n)$, but we expect the actual timing to be much less than this, as the first relevant and applicable plan will cause the function to end.

### 5.3.12 Discussion

We summarize our time complexities in table 5.1. We see that five of the default implementations are constant, so we will exclude them from the rest of our analysis. We will also replace steps 6, 7, and 8 with step X, as we have a default implementation of `selectOption()`. This means that we have four steps to investigate: step 1 (`perceive()`), step 2 (`buf()`), step 10 (`applyExecInt()`) and step X (`applyFindOp()`).
Algorithm 13: Apply and Find Option

Input: $t_e$: Triggering event
Input: $\Pi$: Set of plans that are available to an Agent
Input: $B$: Set of beliefs which comprise the belief base of an Agent
Output: $opt$: Selected option that unifies with triggering event $t_e$ will be selected for an Agent

foreach $\pi \in \Pi$ do
  $t_\pi \leftarrow \pi.trigger()$
  if $t_\pi.equals(t_e)$ then
    $\Pi_{\text{candidate}} \leftarrow \Pi_{\text{candidate}} \cup \{\pi\}$
  end
end

foreach $\pi \in \Pi_{\text{candidate}}$ do
  /* See algorithm 7 for unifies */
  $u \leftarrow \pi.unifies(t_e)$
  if $u \notin \emptyset$ then
    if $\pi.context() \in \emptyset$ then
      $opt \leftarrow \{(\pi, u)\}$
      return $opt$
    else
      /* See algorithm 9 for logicalConsequences */
      $LC \leftarrow \text{logicalConsequences}(\pi.context(), u, B)$
      if $LC \notin \emptyset$ then
        $opt \leftarrow \{(\pi, lc_1); lc_1 \in LC\}$
        return $opt$
      end
    end
  end
end
<table>
<thead>
<tr>
<th>Step</th>
<th>Function</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>perceive</td>
<td>$O(</td>
</tr>
<tr>
<td>Step 2</td>
<td>buf</td>
<td>$O(</td>
</tr>
<tr>
<td>Step 3</td>
<td>checkMail</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Step 4</td>
<td>socAcc</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Step 5</td>
<td>selectEvent</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Step 6</td>
<td>relevantPlans</td>
<td>$O(</td>
</tr>
<tr>
<td>Step 7</td>
<td>applicablePlans</td>
<td>$O(</td>
</tr>
<tr>
<td>Step 8</td>
<td>selectOption</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Step 9</td>
<td>selectIntention</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Step 10</td>
<td>applyExecInt</td>
<td>$O(</td>
</tr>
<tr>
<td>Step X</td>
<td>applyFindOp</td>
<td>$O(</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of Time Complexity per Step of the Reasoning Cycle

### 5.4 Data Flow Analysis of the Reasoning Cycle

Through analysis of the data flow, we will attempt to simplify time complexity by investigating how the data will flow through the reasoning cycle as a whole. In theory, we can potentially eliminate some aspects of the overall time complexity if we can show that the output of one or more steps will be a constant amount of objects regardless of what the input consists of.

Looking at figure 5.9, we see that step 1 takes a variable number of percepts as an input and outputs a variable number of percepts. The input of step 2 consists of a variable number of percepts and a variable number of beliefs, outputting a variable number of external events, as well as a variable number of beliefs. Step 3 has a number of messages as its input, but will output only one (or zero) messages into step 4, meaning that the number of inputs into step 4 will be constant. Since step 4 will take a constant input, we know that the number of messages will not significantly affect our time complexity. However, we know that from section 5.3.4, the time complexity will depend on any customized functions written by the developer and by default will have
Figure 5.9: Jason Reasoning Cycle Data Flow
a constant time complexity. Step 5 takes a number of events as input and outputs one
event into step 6. Step 6 will take a constant number of events (one) as input, but
will also take a variable number of plans from the plan library and output a variable
number of relevant plans. Step 7 will take a variable number of relevant plans, as
well as a variable number of beliefs and output a variable number of applicable plans.
Step 8 will take a variable number of applicable plans and output one plan, which will
get saved into the list of intentions. Step 9 will take a variable number of intentions
and select one intention as an output. Step 10 will take one intention and a variable
number of beliefs and output one action, which could take the form of an updated
intention, an internal action, a belief update or an external action.

5.4.1 Discussion

We were not able to simplify the time complexity through the use of data flow analysis.
Any functions that produced a constant output would have a preceding function take
that constant output along with a variable output, meaning we were not able to
perform any simplification.

5.5 Summary

In this chapter, we have taken a deep dive into how the reasoning cycle operates. We
have first looked at the Agent class and how it interacts the other classes that support
it. We then took a more in-depth look by investigating the sequence of the reasoning
cycle, which has shown us how the pieces of the cycle interact with each other and has
also taught us that not all of the functions get called every iteration of the reasoning
cycle. We will see this in action in chapter 7 when we look at some examples of agents
created for non-testing purposes. We also attempted to simplify the time complexities found in section 5.3 through the use of data flow analysis, but found that we were unable to.

Since the default reasoning cycle will execute steps 1, 2, 10, and X, we feel that analyzing these four steps and the functions contained within will give the best initial look at developing a model of the execution time of the Jason reasoning cycle. Our expectation is that the execution time of \texttt{perceive()} will scale linearly as the percepts received from the environment increase. The execution time of \texttt{buf()} should increase polynomially as the percepts received from the environment increase. The execution time of \texttt{applyExecInt()} and \texttt{applyFindOp()} should increase polynomially as the number of beliefs in the belief base increase, but should increase exponentially as the number of belief rules increase. This exponential increase can potentially be a major bottleneck for the reasoning cycle. Finally, the execution time of \texttt{applyFindOp()} should increase linearly as the number of relevant plans increase. Our next chapter will outline the specifics of how we will test our hypothesis and present our findings.
Chapter 6

Experiments

To verify our analysis of the time complexity done in section 5 as well as coming up with a way to predict the execution time in general, we will generate and measure the performance of agent programs by varying the parameters that we identified in chapter 5 as contributors to the time complexity of Jason’s execution life cycle. We present Table 6.1 as a summary of these parameters.

We will use two hardware configurations for our testing, the first being a Raspberry Pi 1 Model B \[27\] with a single-core 700 MHz ARM11 processor. We have chosen this hardware as it provides a low-end benchmark. Any modern model of Raspberry Pi (or other single-board computers) should outperform this with ease and, as such, have extremely better execution time results. Our second configuration will be a

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>perceive()</td>
<td>Percepts (P)</td>
</tr>
<tr>
<td>buf()</td>
<td>Percepts (P)</td>
</tr>
<tr>
<td>applyExecInt()</td>
<td>Beliefs (B), Belief Rules (n)</td>
</tr>
<tr>
<td>applyFindOp()</td>
<td>Relevant Plans (\text{Opt}_{\text{relevant}}), Beliefs (B), Belief Rules (n)</td>
</tr>
</tbody>
</table>

Table 6.1: Summary of Functions to Analyze and Their Associated Parameters
Windows laptop [28] with a i7-8565U processor, which is a quad-core processor running at 1.80 GHz. This will represent more modern hardware. A summary of hardware specifications can be seen in table 6.2.

We will use JProfiler to record a snapshot of the execution time every time the reasoning cycle is executed. This will provide us a measure of how long the reasoning cycle (and all the functions that comprise it) have been running for, since the start of the profiling. We will subtract the values present in one snapshot from the previous snapshot in order to get the execution time of a single reasoning cycle. We plan to run each instance of a specified parameter five times to get a decent amount of data to analyze. Since there are unknown factors running on our test system, ranging from background processes, to Java garbage collection, to even the overhead of the profiler itself, we feel that it would be beneficial to gather as much data as possible to lessen the effect of these factors.

Recall that we have determined that not all reasoning cycles are created equally. This means that not all functions are executed every reasoning cycle so, as a result, it is possible to get an execution time of zero for a specific function over a given reasoning cycle. Given that we are only interested in reasoning cycles in which these functions were executed, we will filter out instances of zero execution time. We will also remove any outliers by determining the Inter-Quartile Range (IQR) (defined as:

<table>
<thead>
<tr>
<th></th>
<th>Laptop</th>
<th>Raspberry Pi 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Speed</td>
<td>1.8 GHz</td>
<td>700 MHz</td>
</tr>
<tr>
<td>Cores</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>256 KB</td>
<td>16 KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1 MB</td>
<td>128 KB</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>6 MB</td>
<td>—</td>
</tr>
<tr>
<td>RAM (SDRAM)</td>
<td>8 GB DDR4</td>
<td>512 MB</td>
</tr>
</tbody>
</table>

Table 6.2: Summary of Hardware Specifications of Test Systems
third quartile - first quartile) and filtering out values that are beyond ± 1.5 × IQR.

To begin, we will write an agent to test the perceive() and buf() functions. We will be able to use that data to generate a curve of best fit, which we will expect to conform to our analysis of the time complexity and will be able to provide an equation which we can use to estimate the execution time of other agents based on their parameters. We will then repeat this process for applyFindOp() and finally applyExecInt().

6.1 Function: perceive()

Recall in table 5.1, we have stated that the time complexity of perceive() is $O(|P|)$, where $P$ is the number of percepts that have been perceived this reasoning cycle. This means that we should expect a linear increase in execution time as we increase the number of percepts that are processed by the agent.

In order to measure the effect of changing the number of percepts has on the execution time of the reasoning cycle, we will first need to design agents that will perceive different amounts of percepts. By measuring the execution time many times at each percept value, we will be able to see how the execution time changes as the number of percepts change. Our expectation is that it will conform to the time complexity we determined in section 5.3. We expect this to hold true for each of the functions we analyze in this manner.

We will extend the Environment class and override the executeAction() function so that it switches back and forth between providing odd numbers as percepts and even numbers as percepts. We do this so that we can simulate a specified number of percepts being perceived. This will be relevant for the belief update function
(see section 6.2), but, for now, will represent different numbers of perceptions being perceived by the agent. The relevant portions of our customized `executeAction()` function is presented in listing 6.1.

**Listing 6.1: Excerpt of `executeAction()`**

```java
String t;
int startValue = 0;

if (deletePercepts)
    startValue = 1;

deletePercepts = !deletePercepts;

clearPercepts(agName);
for (int i = startValue; i < numPercepts; i += 2)
{
    t = String.valueOf(i);
    addPercept(agName, ASSyntax.parseLiteral("percept (" + t + ")");
}
```

We will use this customized Environment class in conjunction with the agent shown in listing 6.2. Recall that we have shown the syntax of a Jason agent in section 2.2. The agent in listing 6.2 acts as follows: a plan achievement goal is added to the agent, which triggers the only matching plan the agent knows about. The context evaluates to true, so this plan is executed. The action “generateFakePercepts” is taken, which executes `executeAction()` as shown in listing 6.1 which will generate a specified number of percepts (either even or odd numbers for our purposes). The agent will sleep for 100ms, after which it will add a plan achievement goal, thus closing the infinite loop.

On the laptop, we will modify the passed in value of percepts to increase from 0 to 1000, in increments of 100. We will also add in an extreme value of 10000 percepts
to help show that the trend is not localized to a small number of percepts. However, due to lower computing power of the Raspberry Pi, we will modify the passed in value of percepts to increase from 0 to 300, in increments of 50 and add an extreme value of 3000 percepts. Since the extreme value will make any graphs presented very condensed at the lower end, we will present two graphs, one with the smaller values present and then one with the large value added in.

Listing 6.2: Excerpt of Percept Agent

```lisp
!(plan .
+!(plan : true <- generateFakePercepts; 
  . wait (100);
  !plan .
)
```

Since algorithm 1 showed a linear relationship between the execution time and the number of percepts, we expect that equation 6.1 will provide the best fit to our data. We will compare this curve to the curves outlined in section 4.2 to confirm this.

\[ t(x) = ax + b \]  \hspace{1cm} (6.1)

### 6.1.1 Laptop Results

Gathering the data and filtering out outliers as outlined earlier in this chapter, we can plot a boxplot of the execution time vs. the number of percepts that we have forced the agent to perceive and get figure 6.1. Using a boxplot, we can easily see the spread of data broken down into a minimum value, shown as the bottom line below the box. This represents the lowest in range value; any other values below this will be considered an outlier. This is repeated for the maximum value shown as the top line above the box. The blue coloured box represents the values above the first quartile
Figure 6.1: **perceive()**: Execution Time vs. Percepts - Laptop (Boxplot)

![Boxplot showing execution time vs. number of percepts](image)

and below the third quartile, with the median value being represented as the black line. Using this method of visualization shows the spread of data as well as how many outliers may be present.

We can re-plot this data as individual points and use a curve of best fit, as shown in figure 6.2. Adding in the extreme percept value, we get figure 6.3, which looks similar to figure 6.2.

We see in figure 6.1, 6.2, and 6.3 there is a low slope to our line of best fit, showing that the increasing the number of percepts do not affect the execution time of **perceive()** very much. Given that **perceive()** is simply combining two sets into one, it makes sense that an increase in the number of percepts results in a very slight increase in execution time.

Looking at figure 6.2 more closely, we can summarize the results of the R² values

64
Figure 6.2: `perceive()`: Execution Time vs. Percepts - Laptop

![Diagram showing execution time vs. percepts for Laptop]

Figure 6.3: `perceive()`: Execution Time vs. Percepts - Laptop (Extreme)

![Diagram showing execution time vs. percepts for Laptop (Extreme)]
Table 6.3: $R^2$ values of curves of best fit for `perceive()`

<table>
<thead>
<tr>
<th>Equation</th>
<th>$R^2$ value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x) = ax + b$</td>
<td>0.2</td>
</tr>
<tr>
<td>$f(x) = ax^2 + bx + c$</td>
<td>0.2</td>
</tr>
<tr>
<td>$f(x) = a^{bx+c}$</td>
<td>0.19</td>
</tr>
</tbody>
</table>

in table 6.3. We see that all three curves of best fit have an extremely low $R^2$ value, implying that none of the lines are of particularly high quality. Given how spread out the measured data is, this is to be expected. However, we do see that our linear equation has a slightly higher $R^2$ value than our exponential one, meaning that this curve of best fit is the closest to the given data. We also note that the $R^2$ value of the polynomial equation is equal to that of the linear equation, but since our time complexity states that `perceive()` should have a linear relationship between the number of percepts and execution time, we will continue to use that.

Using equation 6.1 and fitting it to the data we have graphed in figure 6.2 and 6.3, we get the equations 6.2 and 6.3, respectively.

\[
t(p) = (3 \times 10^{-02}p + 20) \quad (6.2)
\]

\[
t(p) = (5 \times 10^{-04}p + 40) \quad (6.3)
\]

where:

- $p$ is the number of percepts that have been processed by the agent.
- $t$ is the time in microseconds ($\mu$s).

While there may indeed be a linear increase as predicted by our analysis of the time complexity, the coefficient of the parameter is very small, so the observed behavior is almost constant. It would take an artificially high number of percepts to overwhelm
the constant.

6.1.2 Raspberry Pi Results

Following the same procedure as section 6.1.1, we get figure 6.4, 6.5, and 6.6.

As with our laptop results, we see in figure 6.4, 6.5, and 6.6 a low slope to our line of best fit. However, due to the lower hardware specifications on the Raspberry Pi, we see that overall the execution time is about two orders of magnitude higher than the laptop.

Using equation 6.1 and fitting it to the data we have graphed in figure 6.5 and 6.6, we get the equations 6.4 and 6.5, respectively.

\[ t(p) = (p + 400) \]  \hspace{1cm} (6.4)
Figure 6.5: perceive(): Execution Time vs. Percepts - Raspberry Pi

Figure 6.6: perceive(): Execution Time vs. Percepts - Raspberry Pi (Extreme)
\[
  t(p) = (0.1p + 800) 
\]  

(6.5)

where:

p is the number of percepts that have been processed by the agent.

t is the time in microseconds (\(\mu s\)).

Comparing these equations to the ones we gathered from the laptop, we see that on a slower device, the slope is more perceptible, but the constant is also accordingly higher, which is confirmed by our graphs.

### 6.1.3 Discussion of Results

The main takeaway from comparing the laptop results and the Raspberry Pi results comes from the large difference in scale, with the range of the Pi results being an order of magnitude larger that of the laptop. This is expected due to the difference in hardware specifications between the two systems and will be a common theme as we look at the other aspects of the reasoning cycle. We also see that while the number of percepts affects the execution time, both systems showed that the actual effect of the percepts is minimal, even when an extremely large value of percepts is processed. Also, when comparing the equations before and after adding in an extreme data point, we see that the equation changes minimally. Finally, we have confirmed this portion of our analysis presented in section 5.3.

When it comes to determining our final equation, we will use the versions of our equation that do not take into account the extreme data point (equation 6.2 for the laptop and equation 6.4 for the Raspberry Pi). We will see in chapter 7 that the values of parameters we gather from the three agents we study are extremely small,
so we would like to use equations that have been generated from a model that better reflects our case studies.

6.2 Function: \texttt{buf()} 

Recall in table 5.1, we have stated that the time complexity of \texttt{buf()} is $O(|P_{\text{belief}}|^2)$, where $P_{\text{belief}}$ is the number of percepts that are in the agent’s belief base. This means that we should expect a polynomial increase in execution time as we increase the number of percepts that are processed by the agent.

We will be able to use the same agents to analyse this portion of the reasoning cycle that we had used in section 6.1. Because we are perceiving two completely different sets of percepts each reasoning cycle, we are forcing the belief update function to remove everything it has perceived last reasoning cycle (as it is no longer being perceived) and add a new set of percepts that are currently being perceived. This will cause both the “add” functionality and “delete” functionality to be executed, although we are only interested in the “delete” portion, as that has a higher order of time complexity than the addition.

As outlined in section 5.3.2, adding percepts to the belief base is a linear action, whereas it is the deletion that causes the extra time complexity. During our initial experimentation, it was noticed that, at higher numbers of percepts, there looked to be two graphs overlaid onto each other (e.g: figure 6.7). In order to fully investigate this, we turned to the Jason Mind Inspector to track the additions and deletions being done to the belief base and found that not every invocation of \texttt{buf()} will delete and add all of the perceived percepts to our belief base. It is possible to get reasoning cycles which will only add a fraction of the desired percepts to the belief base, which
happens more frequently than reasoning cycles which will delete all the percepts from the belief base. We see in table 6.4 that, for 100 percepts, we add and delete the expected 200 percepts per cycle (100 additions plus 100 deletions), but for 500 and 1000 percepts, sometimes not all of the percepts are added or deleted as expected. This explains the results we had gotten in figure 6.7. Since we are recording each reasoning cycle, we end up graphing some cycles that do not complete all the expected additions and deletions, and thus have a shorter execution time that expected. We will manually filter out these values, as to not impact our estimates of the reasoning cycle. The “delete” portion of the function will increase faster than the “add” portion, so we should be focusing on the deletions.

It is currently unclear as to why this happens, as adding to the belief base takes much less time than deleting from it. We would have expected that deleting from the belief base could take multiple iterations of \texttt{buf()}, but it was not the case.

Since algorithm 2 showed a polynomial relationship between the execution time and the number of percepts, we expect that equation 6.6 will provide the best fit to
our data. We will compare this curve to the curves outlined in section 4.2 to confirm this.

\[ t(x) = ax^2 + bx + c \]  

(6.6)

### 6.2.1 Laptop Results

Gathering the results and filtering out the outliers (as well as removing any reasoning cycles which only added percepts to the belief base), we get the boxplot shown in figure 6.8. We also will re-plot the data and add a line of best fit (figure 6.9), using equation 6.6 as the base, which gives equation 6.7. We can simplify equation 6.7 by removing the constant term, as it is extremely small when compared to the rest of the equation, giving equation 6.8.

We will again add the extreme value of 10000 percepts to make sure that the trend
continues as we expect it to, as shown in figure 6.10. Using equation 6.6 to fit a curve gives us equation 6.9.

We see from figures 6.8, 6.9 and 6.10 what looks to be a quadratic relationship between the number of percepts and execution time. Looking at figure 6.9 closer and summarizing the $R^2$ values in table 6.5, we can confirm that the quadratic curve of best fit has the highest $R^2$ value, meaning that it is the best of the three curves we have attempted to fit. However we also notice that our extreme data point in figure 6.10 has a very large spread, which we expect will be reflected in our generated

<table>
<thead>
<tr>
<th>Equation</th>
<th>$R^2$ value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x) = ax + b$</td>
<td>0.92</td>
</tr>
<tr>
<td>$f(x) = ax^2 + bx + c$</td>
<td>0.96</td>
</tr>
<tr>
<td>$f(x) = a^{(bx+c)}$</td>
<td>0.93</td>
</tr>
</tbody>
</table>

Table 6.5: $R^2$ values of curves of best fit for buf()
Figure 6.9: \textbf{buf()}: Execution Time vs. Percepts - Laptop

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure6_9}
\caption{Execution Time vs. Percepts - Laptop}
\end{figure}

Figure 6.10: \textbf{buf()}: Execution Time vs. Percepts - Laptop (Extreme)

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure6_10}
\caption{Execution Time vs. Percepts - Laptop (Extreme)}
\end{figure}
equation.

\[
t(p) = (0.2p^2 + 100p + 9 \times 10^{-11})
\]

(6.7)

\[
t(p) = (0.2p^2 + 100p)
\]

(6.8)

\[
t(p) = (0.4p^2 + p + 10000)
\]

(6.9)

where:

p is the number of percepts that have been processed by the agent.

t is the time in microseconds (\(\mu s\)).

We see that there is a large difference between our equation that uses the extreme data point (equation 6.9) and the one that doesn’t (equation 6.8). Our data points from 0 to 1000 have been compressed to the point of insignificance and our data at 10000 percepts has an extremely large spread. Overall we see that the trend continues over a large number of percepts, but the equation gathered from it is not very helpful.

6.2.2 Raspberry Pi Results

As in section 6.2.1, we can gather the results and filter out the outliers (as well as removing any reasoning cycles which only added percepts to the belief base). This gives the boxplot shown in figure 6.11. We also will re-plot the data and add a line of best fit in figure 6.12 using equation 6.6 as the base. Finally, we will add an extreme value of 3000 percepts in figure 6.13.

Again, figure 6.11, 6.12, and 6.13 show how our predicted time complexity predicts
Figure 6.11: `buf()`: Execution Time vs. Percepts - Raspberry Pi (Boxplot)

Figure 6.12: `buf()`: Execution Time vs. Percepts - Raspberry Pi
the execution time.

Using equation 6.6 and fitting it to the data we have graphed in figure 6.12 and 6.13, we get the equations 6.10 and 6.11, respectively. We can further simplify equation 6.11 by removing the constant term, as it is extremely small when compared to the rest of the equation, giving equation 6.12.

\[
t(p) = (200p^2 + 10000p + 20000)
\] 

(6.10)

\[
t(p) = (80p^2 + 30000p + 2 \times 10^{-18})
\]

(6.11)

\[
t(p) = (80p^2 + 30000p)
\]

(6.12)
where:

- \( p \) is the number of percepts that have been processed by the agent.
- \( t \) is the time in microseconds (\( \mu s \)).

We notice the large difference in coefficients in the equations generated in this section as compared to the equations generated in section 6.2.1, again, due to the difference in hardware. We also see that, in equation 6.10, our equation takes into account a \( 20000 \mu s \) execution time when there are no percepts to process. This is an unfortunate aspect of fitting a curve to our high spread data points and we will see this in our case study in chapter 7.

### 6.2.3 Discussion of Results

We note that in figure 6.11, despite gathering data from the Raspberry Pi for 300 percepts, all the recorded invocations of the reasoning cycle were adding percepts to the belief base, meaning we did not use any of the execution time data for that number of percepts. Recall that this is due to the fact that some invocations of \texttt{buf()} will only add part of the percept list into the belief base, instead of adding and deleting the whole thing. This means that, in this case, only percepts were added, and none were deleted. Looking at figure 6.13 we see only one recorded instance of deletion at 3000 percepts. Looking at figures 6.8 and 6.11, we do notice a larger spread of data as the number of percepts increase. We can also use this spread as a reason as to why the equation changes between the versions with and without the extreme data point (although this is somewhat mitigated on the Raspberry Pi due to the lower number of data points collected). On the laptop, we see additional variance in the extreme data point causing the curve of best fit to be shifted upwards, but for the Raspberry Pi, we see the curve shift down because only one valid point was recorded.
For this portion of the final equation, we will use the equation that doesn’t take into account the extreme data point for the laptop (equation 6.8) and for the Raspberry Pi (equation 6.10).

6.3 Function: applyFindOp()

Recall in table 5.1, we have stated that the time complexity of applyFindOp() is \( O(|\text{Opt}_{\text{relevant}}| \times |B|^n) \), where \( \text{Opt}_{\text{relevant}} \) is the number of relevant options, \( B \) is the number of beliefs within the agents belief base and \( n \) is the number of belief rules the agent has. There is a complicated relationship between all of these terms and so we will look at each variable separately in the following sections, but want to point out that overall, this is an exponential time complexity (due to the \( B^n \) term) and as such is potentially a major bottleneck in the execution of the reasoning cycle.

To test the execution time of this function, we will need to generate agents that can modify the number of relevant plans, the number of beliefs and the number of belief rules. We can see an example of the generated agent file presented in listing 6.3, using syntax presented in section 2.2. We will be able to modify the number of beliefs (shown as ‘belief’ in listing 6.3), the number of belief rules (shown as ‘test’ in listing 6.3) and the number of plans (shown as ‘plan’ in listing 6.3). The agent presented in listing 6.3 will act as follows. A number of beliefs (specified by us) will be added to the belief base. The achievement goal ‘plan’ is added to the agent, which triggers a search through all the plans (specified by us) to see if any contexts will evaluate to true. Upon not finding a context that evaluates to true, the last plan is triggered, which similar to listing 6.2, will take the “generateFakePercepts” action. However, in this case, only a small, constant, number of percepts are perceived, as this is not a
parameter we are testing in this section. The agent will sleep for 100ms after which it will add the achievement goal ‘plan’ again, closing the infinite loop.

Listing 6.3: Excerpt of Percept Agent

\begin{verbatim}
belief(10000).
/* Modify number of beliefs as required */

test(N) :- N > 0 & test(N-1).
/* Modify number of belief rules as required */

!plan(5).

+!plan(N) : belief(N) <- generateFakePercects; .wait(100); !plan(N).
/* Modify number of plans as required */

+!plan(N) : test(N) <- .wait(100); generateFakePercects; !plan(N).
=!plan(N) : true <- generateFakePercects; .wait(100); !plan(N).
\end{verbatim}

We will use equation 6.13 as a base for our curve fit, as it most closely resembles the time complexity we have determined \texttt{applyFindOp()} to be.

\[ t(x, y, z) = (ax + b) \times (cy + d) \times (ey + f)^{gz+h} \]  
(6.13)

6.3.1 Laptop Results

Because this equation is fairly complex, comprised of three variables, we will have a look at each parameter individually to make sure that the execution time of \texttt{applyFindOp()} increases when each parameter is increased. We will combine our data back into one group when we attempt to fit a curve, as all three of the parameters will affect the execution time. We will not present any equations for the individual portions, as they would only take into account the varying parameter.
### Table 6.6: R² values of curves of best fit for the plans of `applyFindOp()`

<table>
<thead>
<tr>
<th>Equation</th>
<th>R² value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x) = ax + b$</td>
<td>0.99</td>
</tr>
<tr>
<td>$f(x) = ax^2 + bx + c$</td>
<td>0.99</td>
</tr>
<tr>
<td>$f(x) = a^{bx+c}$</td>
<td>0.96</td>
</tr>
</tbody>
</table>

6.3.1.1 Plans

First, we will graph a sample of the plan data. We will set the number of beliefs to be 100 and the number of belief rules to be 10, varying the number of plans only. Looking at figure 6.14, 6.15, and 6.16, we see that the execution time of `applyFindOp()` increases as we increase the number of plans available to the agent.

Summarizing the R² values in table 6.6, we see that the linear and polynomial equations have the highest R² value. Since we have determined this parameter of `applyFindOp()` to have a linear relationship to the execution time, we will use that for our overall equation.

6.3.1.2 Beliefs

Secondly, we will graph a sample of the belief data. We will set the number of plans to be 100 and the number of belief rules to be 10, varying the number of beliefs only. Looking at figures 6.17, 6.18, and 6.19, we see that the execution time of `applyFindOp()` increases as the number of beliefs increase.

Summarizing the R² values in table 6.7, we see that the linear and polynomial equations have the highest R² value. Since we have determined this parameter of `applyFindOp()` to have a polynomial relationship to the execution time, we will use that for our overall equation.
Figure 6.14: applyFindOp(): Execution Time vs. Plans - Laptop (Boxplot)

Figure 6.15: applyFindOp(): Execution Time vs. Plans - Laptop
Figure 6.16: applyFindOp(): Execution Time vs. Plans - Laptop (Extreme)

Table 6.7: $R^2$ values of curves of best fit for the beliefs of applyFindOp()
Figure 6.17: `applyFindOp()`: Execution Time vs. Beliefs - Laptop (Boxplot)

Figure 6.18: `applyFindOp()`: Execution Time vs. Beliefs - Laptop
6.3.1.3 Belief Rules

Finally, we will graph a sample of the belief rule data. We will set the number of plans to be 100, and the number of beliefs to be 100 varying the number of beliefs rules only. Looking at figures 6.20, 6.21, and 6.22, we can see that the execution time of applyFindOp() increases as the number of belief rules increase.

Summarizing the $R^2$ values in table 6.8, we see that the polynomial and exponential equations have the highest $R^2$ value. Since we have determined this parameter of applyFindOp() to have an exponential relationship to the execution time, we will use that for our overall equation.
Table 6.8: $R^2$ values of curves of best fit for the belief rules of applyFindOp()
Figure 6.21: `applyFindOp()`: Execution Time vs. Belief Rules - Laptop

Figure 6.22: `applyFindOp()`: Execution Time vs. Belief Rules - Laptop (Extreme)
6.3.1.4 Determining the Final Equation

While we can confirm individually that increasing each parameter will increase the execution time, we still need to create a curve of best fit for the function as a whole. Graphing the plans, beliefs and belief rules vs the execution time gives the graph shown in figure 6.23. Note that this figure is provided more as a general idea of the trend that we have seen in the last three sections. We will also not plot this graph a second time with the extreme data points added.

Using equation 6.13 and fitting it to the data we have graphed in figure 6.23 we get equation 6.14. We can further simplify this equation by setting the $2 \times 10^{-13}b$ term to zero, as it is extremely small when compared to the constant 40 that is added
to it. We can also multiply the terms together to get equation 6.15.

Similarly, we will use equation 6.13 and fit it to the combination of what is graphed in figure 6.23 as well as the extreme data points to get equation 6.16. We can further simplify this equation by setting the $6 \times 10^{-19}b$ term to zero, as it is extremely small when compared to the constant 40 that is added to it. We can also multiply the terms together to get equation 6.17.

\[
t(b, l, r) = (0.6l + 100)(0.7b + 100)((2 \times 10^{-13}b + 40)^{0.004r+0.5} \tag{6.14}
\]

\[
\boxed{t(b, l, r) = (0.42bl + 70b + 60l + 10000)(40)^{0.004r+0.5}} \tag{6.15}
\]

\[
t(b, l, r) = (0.5l + 100)(0.5b + 100)((6 \times 10^{-19}b + 40)^{0.006r+0.7} \tag{6.16}
\]

\[
\boxed{t(b, l, r) = (0.25bl + 50b + 50l + 10000)(40)^{0.006r+0.7} \tag{6.17}}
\]

where:

b is the number of beliefs that exist within the belief base of the agent.

l is the number of relevant plans that are available to the agent.

r is the number of belief rules that are available to the agent.

t is the time in microseconds ($\mu$s).

We can see from equation 6.15 and 6.17 that each of the parameters are dependent on each other, making this a more complicated equation. We also note a large similarity between the equation with the extreme data point (equation 6.17) and no extreme data point (equation 6.15). It is possible that this is because the belief rules were not
increased as much as the other two parameters (or as much as the percepts were in section 6.2), thus adding less spread to the measured execution times.

### 6.3.2 Raspberry Pi Results

We will follow the same procedure as before, in section 6.3.1. We expect to get a similar shape, just larger in magnitude.

#### 6.3.2.1 Plans

Again, we will first graph a sample of the plan data, setting the number of beliefs to be 100 and the number of belief rules to be 10. Looking at figures 6.24, 6.25, and 6.26, we can see that increasing the number of plans available to the agent increases the execution time of `applyFindOp()`.
Figure 6.25: `applyFindOp()`: Execution Time vs. Plans - Raspberry Pi

Figure 6.26: `applyFindOp()`: Execution Time vs. Plans - Raspberry Pi (Extreme)
6.3.2.2 Beliefs

Graphing the belief data by setting the number of plans to 100 and the number of belief rules to be 10 and varying the number of beliefs gives figures 6.27, 6.28, and 6.29. We see that the increase of beliefs increases the execution time of applyFindOp().

6.3.2.3 Belief Rules

Finally, we graph a sample of the belief rule data. We again set the number of plans to 100 and the number of beliefs to 100. Varying the number of belief rules, we get figures 6.30, 6.31, and 6.32, which show that increasing the number of belief rules will increase the execution time of applyFindOp().
Figure 6.28: `applyFindOp()`: Execution Time vs. Beliefs - Raspberry Pi

Figure 6.29: `applyFindOp()`: Execution Time vs. Beliefs - Raspberry Pi (Extreme)
Figure 6.30: `applyFindOp()`: Execution Time vs. Belief Rules - Raspberry Pi (Boxplot)
6.3.2.4 Determining the Final Equation

We see again that, like section 6.3.1, increasing each individual parameter will increase the execution time of `applyFindOp()`, we again need to create a curve of best fit for the function as a whole, as each portion is not independent. Graphing the plans, beliefs and belief rules vs the execution time gives the graph shown in figure 6.33. Note that, as in section 6.3.1.4, this figure is provided more as a general idea of the trend that we have seen in the last three sections.

Using equation 6.13 and fitting it to the data we have graphed in figure 6.33, we get equation 6.18. We can further simplify this equation by setting the $5 \times 10^{-12}b$ term to zero, as it is extremely small compared to the 200 that is added to it. We can also multiply the terms together to get equation 6.19.

Similarly, we will use equation 6.13 and fit it to the data we have graphed in figure
Figure 6.32: applyFindOp(): Execution Time vs. Belief Rules - Raspberry Pi (Extreme)
Figure 6.33: `applyFindOp()`: Execution Time vs. Beliefs, Belief Rules and Plans - Raspberry Pi
6.33 as well as the extreme data points to get equation 6.20. We can further simplify this equation by setting the $4 \times 10^{-13}b$ term to zero, as it is extremely small compared to the 30 that is added to it. We can also multiply the terms together to get equation 6.21.

$$t(b, l, r) = (5l + 900)(5b + 900)((5 \times 10^{-12}b + 200)^{0.005r+0.4}$$

(6.18)

$$t(b, l, r) = (25bl + 4500b + 4500l + 810000)(200)^{0.005r+0.4}$$

(6.19)

$$t(b, l, r) = (0.4l + 100)(0.4b + 100)((4 \times 10^{-13}b + 30)^{0.009r+2}$$

(6.20)

$$t(b, l, r) = (0.16bl + 40b + 40l + 10000)(30)^{0.009r+2}$$

(6.21)

where:

b is the number of beliefs that exist within the belief base of the agent.

l is the number of relevant plans that are available to the agent.

r is the number of belief rules that are available to the agent.

t is the time in microseconds ($\mu$s).

Unlike the equations presented in section 6.3.1.4, we see a large difference between the equation with the extreme data point (equation 6.21) and with no extreme data point (equation 6.19). This is due to the hardware limitations of the Raspberry Pi when processing these extreme values. However, we see that equation 6.19 and equation 6.15 follow a similar format.
6.3.3 Discussion of Results

As with the other two functions, we find that we get two equations that are similar between the laptop and the Raspberry Pi. The main difference between them being a matter of scale, which is inline with the fact that the Raspberry Pi has much worse hardware specifications when compared to the laptop. We also see that adding the extreme data point introduces more variance, giving us a less accurate model. Because of this, we will be using the equations that do not take into account the extreme data point (equation 6.15 and 6.19 for the laptop and Raspberry Pi, respectively).

6.4 Function: applyExecInt()

Recall in table 5.1, we have stated that the time complexity of \texttt{applyExecInt()} is \(O(|B|^n)\), where \(B\) is the number of beliefs within the agents belief base and \(n\) is the number of belief rules the agent has. As with \texttt{applyFindOp()}, there is a complicated relationship between these terms, so we will look at them separately in the following sections. This is also an exponential time complexity and as such is potentially a major bottleneck in the execution of the reasoning cycle.

We will use the same agent files as presented in listing 6.3.

We will use equation 6.22 as a base for our curve fit, as it most closely resembles the time complexity we have determined \texttt{applyExecInt()} to be.

\[
 t(x, y) = ((ax + b)^cy + d)
\]  
(6.22)
6.4.1 Laptop Results

As with section 6.3.1, we will break our analysis of the time complexity down into two parts: the beliefs and the belief rules. We expect the execution time of the reasoning cycle to increase when we increase either of these parameters.

6.4.1.1 Beliefs

First, we will graph a sample of the belief data. We will set the number of belief rules to be 10 and vary the number of beliefs only. Looking at figures 6.34 and 6.35, we see that the execution time of `applyExecInt()` does not increase when the number of beliefs increase.
6.4.1.2 Belief Rules

Next, we will graph a sample of the belief rule data. We will set the number of beliefs to be 100 and vary the number of beliefs rules only. Looking at figures 6.36 and 6.37 we see that, as with the beliefs, the execution time of \texttt{applyExecInt()} does not increase as the number of belief rules increase.

6.4.2 Raspberry Pi Results

Since we did not get the expected results from the laptop, we will look at the data retrieved from the Raspberry Pi to see if we get similar results to the laptop or if we are closer to our expected result.
Figure 6.36: \texttt{applyExecInt()}: Execution Time vs. Belief Rules - Laptop (Boxplot)

Figure 6.37: \texttt{applyExecInt()}: Execution Time vs. Belief Rules - Laptop
6.4.2.1 Beliefs

We will graph the belief data, keeping the number of belief rules fixed at 10 and vary the number of beliefs. Looking at figures 6.38 and 6.39, we see that while there looks to be a very slight relationship, if we take into account our laptop results, we can conclude that this is also evidence that the beliefs do not have a significant impact on the execution time.

6.4.2.2 Belief Rules

Fixing the number of beliefs at 100 and varying the belief rules, we see that, again in figures 6.40 and 6.41, there looks to be no evidence to support the idea that the number of belief rules affect the execution time of `applyExecInt()`.
6.4.3 Discussion of Results

Given that we did not get the expected results, we need to look back to our analysis of the time complexity to understand why. Having a second look highlights the idea that there are many different types of plan bodies, including constraints, and internal actions. Internal actions are actions that have been defined apart from the reasoning cycle but can be run as part of a plan. These fall outside the scope of our analysis, as we cannot control the execution time of those functions (like our original exclusion of overridden functions, opting to use the default ones provided by the Jason framework). We had originally thought that the constraints, which triggered the finding of logical consequences, would make up much of our execution time. Given the results we see in figures 6.35, 6.37, 6.39, and 6.41, we see this is not the case. This plan type seem to be rarely used, looking through [2] seems to not make any reference to this plan type.
Figure 6.40: `applyExecInt()`: Execution Time vs. Belief Rules - Raspberry Pi (Boxplot)
It is our impression that due to the rarity of the logical consequences being called in this function, and the prevalence of internal actions, of which we can have no control over, that it would be best to exclude `applyExecInt()` from the model at this time.

### 6.5 Discussion

The last portion that we require is the miscellaneous execution time that has not been captured by our analysis. This represents code that runs outside the ten steps we have defined in chapter 5.3. We can simply add up the execution time of each of the steps and subtract it from the recorded time of the reasoning cycle. We will discuss the specifics for the laptop and Raspberry Pi in sections 6.5.1 and 6.5.2, respectively,
as they will be dependent on the hardware.

6.5.1 Laptop Discussion

Adding up the individual steps of the reasoning cycle and comparing it to the recorded execution time, we see that there is about 800\(\mu s\) unaccounted for that happens outside of the ten analyzed steps. This means that, once we combine all of the individual pieces of the execution of the reasoning cycle, we will get equation 6.23. Throughout this chapter, we have looked at two sets of data, one that does not include an extremely large data point, and one that does. In all cases of our laptop testing, the additional parameter has had a large amount of spread, which changes our equation. It is our expectation that through careful experimentation of more parameters between 1000 and 10000, we will be able to refine the model and remove some of the added uncertainty. For our purposes, however, we will use the equations generated without the extremely large data point, as it gives a more accurate reflection of the range of parameters we expect to see in our case studies. Substituting the equations developed in this chapter (equation 6.2, 6.8, and 6.15), we get equation 6.24. We will further simplify this equation by setting the \(3 \times 10^{-2}p\) term to zero, as it is small when compared to the \(100p\) term, giving us equation 6.25.

\[
\begin{align*}
t_{\text{reasoning\_cycle}} &= t_{\text{perceive}} + t_{\text{buf}} + t_{\text{apply\_FindOp}} + t_{\text{external}} \\
t(b,l,p,r) &= (3 \times 10^{-2}p + 20) + (0.2p^2 + 100p) \\
&\quad + (0.42bl + 70b + 60l + 10000)(40)^{0.004r+0.5} + 800
\end{align*}
\]
\[
t(b, l, p, r) = (0.2p^2 + 100p) + (0.42bl + 70b + 60l + 10000)(40)^{0.004r+0.5} + 820
\]

(6.25)

where:

b is the number of beliefs that exist within the belief base of the agent.
l is the number of relevant plans that are available to the agent.
p is the number of percepts that have been processed by the agent.
r is the number of belief rules that are available to the agent.
t is the time in microseconds (\(\mu s\)).

The equation that represents our model of the execution time of the Jason reasoning cycle for the laptop test system is equation 6.25. We notice that when compared to the rest of the parameters, the time represented by the `perceive()` function was negligible. We see that the perceptions, belief, belief rules and the plans all contribute to the overall execution time as well as there being execution outside of our analysis, represented by a constant time.

### 6.5.2 Raspberry Pi Discussion

Similarly for the Raspberry Pi, we see that there is about 30000\(\mu s\) unaccounted for outside of the ten analyzed steps. We will add this unaccounted time to our combined calculation of the execution time. Using equation 6.23 as our guide and substituting in (equation 6.4, 6.10, and 6.19), we get equation 6.26. We will simplify this equation by setting the \(p\) term to zero, as it is extremely small when compared to the 10000\(p\) term, and will remove the constant 400, as it is small when compared to 20000, which
gives equation 6.27.

\[
t(b, l, p, r) = (p + 400) + (200p^2 + 10000p + 20000) \\
+ (25bl + 4500b + 4500l + 810000)(200)^{0.005r+0.4} + 30000 
\] (6.26)

\[
t(b, l, p, r) = (200p^2 + 10000p) + (25bl + 4500b + 4500l + 810000)(200)^{0.005r+0.4} + 50000 
\] (6.27)

where:
b is the number of beliefs that exist within the belief base of the agent.
l is the number of relevant plans that are available to the agent.
p is the number of percepts that have been processed by the agent.
r is the number of belief rules that are available to the agent.
t is the time in microseconds (µs).

The equation that represents our model of the execution time of the Jason reasoning cycle for the Raspberry Pi test system is equation 6.27. Again, we notice that even on this system, the time represented by the perceive() function was negligible. We also note how it looks extremely similar to the equation generated for the laptop (equation 6.25). In fact, the only difference are the larger coefficients, which represent the difference between hardware specifications between the two systems.

6.5.3 Discussion Summary

We should reiterate that not every reasoning cycle is created equally. This means that not all steps get called every time the reasoning cycle is executed, so the equations presented in section 6.5.1 and section 6.5.2 represent the theoretical maximum execution
time if all functions are invoked on a given execution of the reasoning cycle. We can come up with a theoretical maximum for each combination of functions in the reasoning cycle in the case that that combination gets run. We also have seen that there is a constant amount of time that is not captured as part of our analysis. This represents execution time outside of the ten steps and should be added to all calculations. The values for this extra parameter is presented in table 6.9.

6.6 Summary

In this section, we have presented our experimental design to determine the execution time of the Jason reasoning cycle. We also used this experiment, through the calculation of $R^2$ values as validation of our analysis of the time complexity done in chapter 5. For the most part, our experimentation validates our analysis and we have come up with a possible explanation in section 6.4.3 as to why we did not get the expected result for `applyExecInt()`. We have looked at two different test systems, a Windows laptop and a Raspberry Pi (see table 6.2 for specifications), and shown that the hardware affects the equations that describe the execution time. We have also come up with a general idea of what the equations for these two systems would look like, giving an upper bound of the execution time based on the percepts, beliefs and belief rules. We have seen in both developed equations that the effect of the percepts on the execution time of `perceive()` is negligible. However, the effect of percepts on the execution time of `buf()` is polynomial and could potentially be a bottleneck. We also see that the

<table>
<thead>
<tr>
<th>Laptop</th>
<th>Raspberry Pi</th>
</tr>
</thead>
<tbody>
<tr>
<td>800µs</td>
<td>30000µs</td>
</tr>
</tbody>
</table>

Table 6.9: Additional Time Outside of Reasoning Cycle Steps
effect of belief rules has an exponential effect on the execution time of `applyFindOp()` and could potentially be a major bottleneck.

We will use these equations to estimate the execution time of three case studies in chapter 7 to get an idea of how well our model will perform with agents that have not been specifically designed by us. Given that we have designed our own agents in this chapter, we feel it would be beneficial to look at agents that already exist as to not have our bias influence the results, as well as use this as an opportunity to see how many parameters are used in "real world" agents.
Chapter 7

Case Studies

To get an idea of whether our model is appropriate, we will look at three programs that have been written by others. The first agent we will look at comes bundled with Jason as an example agent: The Domestic Robot [21]. We will then look at a grid world agent [23], and finally, a mail delivery robot [19, 24] which uses the ROS software package to drive the hardware, and a Jason agent as the brain. We will first analyze the Jason agents to determine how many beliefs, belief rules, percepts and plans are present. We will run the domestic robot and the grid world agent on both the Windows system and the Raspberry Pi, but the mail delivery robot will be run on a separate system which will be described in section 7.3. Based on the system we are using; we will predict the execution time using the appropriate equations. We will then profile the agent and validate our predicted times.

Recall that our equations generated in section 6.5 describe the maximum execution time if all functions we have analyzed run. These three functions can run in any combination, meaning that while we always expect our execution time to be less than what we have calculated, we can provide a little bit more granularity. Instead of
always calculating the absolute maximum execution time of the reasoning cycle as a whole, we can look at the following eight cases:

1. none of the three functions run;

2. perceive() only runs;

3. buf() only runs;

4. applyFindOp() only runs;

5. both perceive() and buf() run;

6. both buf() and applyFindOp() run;

7. both perceive() and applyFindOp() run;

8. all three functions run.

We can calculate each of these and then categorize our snapshots into executions of the reasoning cycle in which each of these scenarios happen. We can then look at the max value of each of these and see how they relate to our modelled values.

7.1 Domestic Robot

The domestic robot [21] is a set of three agents: an owner agent, a robot agent, and a supermarket agent. The owner agent will request beer from a robot and then when received, will drink that beer. It will also randomly request the time from the robot. We will modify this to request the time every 13 seconds, instead of being random, as we would like to make the running of the agent deterministic, for a more consistent study. The robot agent will travel between the refrigerator and owner, delivering beer
as requested. If the refrigerator is running low, the robot will request more beer from the supermarket agent. The robot agent is also responsible for keeping track of how many beers the owner has had per day and refusing to provide more beer if the limit has been reached. Normally this limit is ten, but for our purposes we will change that limit to be 1000 so that we can ensure our robot will always be busy for the entire time we profile it. Finally, the supermarket agent will send beer to the robot agent when requested.

Our profiling will be limited to the robot agent, as that is the most complex of the three agents. Our first step will be to determine how many plans, beliefs and belief rules exist within the agent. We will also need to see what percepts are available for the robot to perceive. Table 7.1 summarizes the analysis done on the robot agent. See appendix A.1 for the source code of the robot agent and appendix A.2 for the list of possible percepts extracted from the domestic robot environment class.

For the laptop system, we will substitute the values from table 7.1 into equations 6.2, 6.8, and 6.15, giving us the estimated execution time of the each of our analyzed functions of the robot agent. We can then calculate the other combinations from there. For the Raspberry Pi, we will substitute those same values into equations 6.4, 6.10, and 6.19 and get estimated values appropriate for the Raspberry Pi. Table 7.2 shows the approximate time that has been calculated using our models, in $\mu s$.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beliefs (b)</td>
<td>2</td>
</tr>
<tr>
<td>Belief Rules (r)</td>
<td>1</td>
</tr>
<tr>
<td>Percepts (p)</td>
<td>8</td>
</tr>
<tr>
<td>Relevant Plans (l)</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 7.1: Parameters Extracted From Robot Agent
### Table 7.2: Predicted Execution Times of Domestic Robot Agent from Model (in µs)

<table>
<thead>
<tr>
<th>Function Combination</th>
<th>Laptop</th>
<th>Raspberry Pi</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>800</td>
<td>30000</td>
</tr>
<tr>
<td>perceive</td>
<td>820</td>
<td>30400</td>
</tr>
<tr>
<td>buf</td>
<td>1600</td>
<td>130000</td>
</tr>
<tr>
<td>applyFindOp</td>
<td>70800</td>
<td>9030000</td>
</tr>
<tr>
<td>perceive,buf</td>
<td>1620</td>
<td>130400</td>
</tr>
<tr>
<td>perceive/applyFindOp</td>
<td>70820</td>
<td>9030400</td>
</tr>
<tr>
<td>buf/applyFindOp</td>
<td>71600</td>
<td>9130000</td>
</tr>
<tr>
<td>perceive/buf/applyFindOp</td>
<td>71620</td>
<td>9130400</td>
</tr>
</tbody>
</table>

#### 7.1.1 Results of Domestic Robot Execution

Collecting the data we gathered from the domestic robot agent, we can categorize it into one of eight categories, as summarized as a boxplot in figure 7.1 for the laptop and 7.2 for the Raspberry Pi. Looking at these boxplots, we see that our predicted values for reasoning cycles that have executed `applyFindOp()` end up being extremely far off the top of the graph. In order to present our predicted execution times in a meaningful way, we can re-plot the graphs shown in figure 7.1 and 7.2 with a logarithmic y-axis. Since this is the first attempt at creating this type of model, we can expect a large discrepancy between values from the model and measured values and, as such, relating these two values in terms of orders of magnitude provides a good comparison. We present our re-plotted graphs in figure 7.3 and 7.4.

Of note, we see that in figure 7.2 and 7.4 there is no execution time for `buf()`. This is due to the fact that no reasoning cycle executed that called only `buf()`. Recall that not all reasoning cycles will execute the same way (as we have discussed in section 5.2.4) and we expect to see a varying number of invocations for each combination. This also has the unfortunate side effect of providing us with some boxplots that have many data points and other boxplots that have very few data points, which
somewhat diminish the information that can be gleaned from them. For example, looking at figure 7.3, we see that the execution time of the combination of `perceive() / applyFindOp()` (with six recorded invocations) looks to be higher than that of the combination of `perceive() / buf() / applyFindOp()` (based on the average line being higher and the box portion of the plot extending higher), with 16 recorded invocations. We would expect that with a larger number of collected data points, we would see the boxplot trend towards a more expected configuration (i.e: the execution time of the `perceive() / buf() / applyFindOp()` combination to be higher than that of the `buf() / applyFindOp()` combination, which would in turn be higher than that of the `perceive() / applyFindOp()` combination).
Figure 7.2: Domestic Robot: Execution Time vs. Function Called - Raspberry Pi

Figure 7.3: Domestic Robot: Log Execution Time vs. Function Called - Laptop
Figure 7.4: Domestic Robot: Log Execution Time vs. Function Called - Raspberry Pi

![Graph showing log execution time vs. function called for Raspberry Pi study.]

Table 7.3: Order of Maximum Execution Time for Domestic Robot Agent (in µs)

<table>
<thead>
<tr>
<th>Study</th>
<th>Laptop</th>
<th>Raspberry Pi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domestic Robot (Model)</td>
<td>$10^4$</td>
<td>$10^6$</td>
</tr>
<tr>
<td>Domestic Robot (Measured)</td>
<td>$10^3$</td>
<td>$10^5$</td>
</tr>
</tbody>
</table>

Table 7.3: Order of Maximum Execution Time for Domestic Robot Agent (in µs)
7.1.2 Discussion of Results

We summarize the results of figure 7.3 and 7.4 into table 7.3. Since we are looking at log values in these figures, we will present orders of magnitude and use this for our basis of comparison. We also do it because this is our first attempt to create a model for the execution time of the reasoning cycle and large discrepancies between model predictions and actual values are to be expected. Also, many parameters are outside of the scope of the model, for example the use of internal functions, which can take an arbitrarily large amount of time. First, we will take our largest predicted execution time and convert that to a magnitude. Then we take the magnitude of the longest reasoning cycle (as this should be our basis for comparison) and compare that to the predicted execution time. By using the longest reasoning cycle, we can provide an upper bound and reasonably expect all other reasoning cycles run under the same conditions to be shorter.

We see that our predicted execution times for each function are higher than the actual execution time. If we look at figure 7.3 and 7.4, we see that the predicted values can be up to an order of magnitude above the maximum measured execution time. This is especially true for cycles that have executed \texttt{applyFindOp()} and to a lesser extent, cycles that have executed \texttt{buf()}. We can attribute a lot of our inaccuracy to the low resolution of our model. When selecting a range to investigate, we chose to look at a relatively large one. Our percepts, beliefs, and plans went from 0 to 1000 on the laptop, and from 0 to 300 on the Raspberry Pi. Our belief rules went from 0 to 50 on both the laptop and the Raspberry Pi. While this is good for a look at the trend in general, none of these parameters for the domestic robot agent were larger than ten. This leads us to believe that perhaps a more micro look at these parameters might be beneficial as future work.
We also notice that the predictions and measured values for the execution time on the Raspberry Pi are two orders of magnitude higher than that of the laptop, highlighting the difference between the two hardware configurations. This also provides justification for the fact that we needed to produce a separate model for each hardware configuration.

### 7.2 Grid World

The grid world agent [23] is an agent that navigates through a small grid environment. It simulates a battery powered car that travels to a destination that it must find. It must avoid obstacles, recharge its battery if needed and not hit pedestrians (honking its horn to move them out of the simulated environment). We will not make any modifications to this agent, opting to do our analysis on the agent as created. We will profile the navigatorEnvAssist agent as implemented in this project. This agent specifically will be assisted by the environment for movement. We use this over the navigator agent because that agent uses a customized `selectOption()` class, which causes `applyFindOp()` to not be called, instead, `relevantPlans()`, `applicablePlans()`, and `selectOption()` will be called. This should not be an issue in general, but at this point, we feel it would be best to keep the reasoning cycle execution as close to the experimental execution as possible wherever possible.

As with the domestic robot agent, we will need to determine how many plans, beliefs and belief rules exist within the agent, and how many percepts are available for the agent to perceive. Refer to appendix B.1 for the agent source code and to appendix B.2 for the list of possible percepts the agent can perceive.

Again, as with the domestic robot agent, we will substitute the values in table...
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beliefs (b)</td>
<td>48</td>
</tr>
<tr>
<td>Belief Rules (r)</td>
<td>4</td>
</tr>
<tr>
<td>Percepts (p)</td>
<td>15</td>
</tr>
<tr>
<td>Relevant Plans (l)</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 7.4: Parameters Extracted from navigatorEnvAssist Agent

<table>
<thead>
<tr>
<th>Function Combination</th>
<th>Laptop</th>
<th>Raspberry Pi</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>800</td>
<td>30000</td>
</tr>
<tr>
<td>perceive</td>
<td>820</td>
<td>30400</td>
</tr>
<tr>
<td>buf</td>
<td>2800</td>
<td>230000</td>
</tr>
<tr>
<td>applyFindOp</td>
<td>90800</td>
<td>10030000</td>
</tr>
<tr>
<td>perceive/buf</td>
<td>2820</td>
<td>230400</td>
</tr>
<tr>
<td>perceive/applyFindOp</td>
<td>90820</td>
<td>10030400</td>
</tr>
<tr>
<td>buf/applyFindOp</td>
<td>92800</td>
<td>10230000</td>
</tr>
<tr>
<td>perceive/buf/applyFindOp</td>
<td>92820</td>
<td>10230400</td>
</tr>
</tbody>
</table>

Table 7.5: Predicted Execution Times of Grid World Agent from Model (in µs)

7.4 into equations 6.2, 6.8, and 6.15 for the laptop, and 6.4, 6.10, and 6.19 for the Raspberry Pi. This will give us an estimate of how long the reasoning cycle will take for this hardware.

### 7.2.1 Results of Grid World Execution

Again, collecting the data we gathered from the grid world agent and categorizing it into the same eight categories, which we summarize in figure 7.5 for the laptop and 7.6 for the Raspberry Pi. We will also re-plot these graphs with a logarithmic y-axis and include our predicted values in Figure 7.7 and 7.8.

Again, looking at figure 7.5, we see no data points for the combination of `perceive() / buf()` and `buf() / applyFindOp()` and on figure 7.8, no data points for the combination of `buf()` only, `buf() / applyFindOp()`, or the case where none of these functions are run. Similar to section 7.1.1, this means that these combinations
were never executed. As well, we see that in figure 7.5, the combination of `perceive()` / `applyFindOp()` (with four data points) is larger than the combination of `perceive()` / `buf()` / `applyFindOp()` (with 20 data points). We would also expect to see that with more data points a more expected configuration of boxplots (again, the execution time of the `perceive()` / `buf()` / `applyFindOp()` combination to be higher than that of the `buf()` / `applyFindOp()` combination, which would in turn be higher than that of the `perceive()` / `applyFindOp()` combination).

### 7.2.2 Discussion of Results

Looking at table 7.6, we see that the predicted execution time of the reasoning cycle on the laptop is within the same order of magnitude as the measured values. We also notice that, similar to the domestic robot, the predicted results of the Raspberry Pi
Figure 7.6: Grid World: Execution Time vs. Function Called - Raspberry Pi

Figure 7.7: Grid World: Log Execution Time vs. Function Called - Laptop
Figure 7.8: Grid World: Log Execution Time vs. Function Called - Raspberry Pi

Table 7.6: Order of Maximum Execution Time for Grid World Agent (in µs)

<table>
<thead>
<tr>
<th>Study</th>
<th>Laptop</th>
<th>Raspberry Pi 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid World (Model)</td>
<td>$10^4$</td>
<td>$10^7$</td>
</tr>
<tr>
<td>Grid World (Measured)</td>
<td>$10^4$</td>
<td>$10^6$</td>
</tr>
</tbody>
</table>
are one order of magnitude higher than the measured results. However, looking at figure 7.8, we must note that extremely few reasoning cycles were executed in the order of seconds (more executed in the order of 0.1 seconds and 0.01 seconds). That said, we see that, again, our prediction of the execution time of the `perceive()` method is extremely close to the recorded execution times. Similarly, the predicted execution times of cycles that executed `applyFindOp()` are not very close (although within an order of magnitude).

Again, we also note that the predicted and measured execution time of the reasoning cycle on the Raspberry Pi are two orders of magnitude higher than that of the laptop. The fact that both of these case studies are so similar shows that this is a good first step towards an accurate model of the execution time of the Jason reasoning cycle. We can also derive the same conclusions from this case study as the previous. Namely that testing more agents with a parameters in-between already testing points would provide a better resolution to the model and potentially increase the accuracy.

### 7.3 Mail Delivery Robot

The last case study we will look at is an existing prototype of a mail delivery robot [24, 19]. This device uses an iRobot Create2 to move around and runs an agent written in Jason on a Raspberry Pi 4 [29]. This version of Raspberry Pi uses a Cortex-A72 processor, which is a quad-core processor running at 1.5 GHz. An updated comparison of the three systems used can be found in table 7.7. This case study is particularly interesting as it represents an agent that runs in a non-simulated environment. Because the Raspberry Pi 4 has not been used as part of the testing, we must determine what its execution time equations are by running the same tests that
we have run on the laptop and Raspberry Pi 1. We should also note that this agent has a customised `selectOption()` function, which means that `applyFindOp()` will not be called. Instead `relevantPlans()`, `applicablePlans()` and `selectOption()` will be called. Recall that the time complexity of `applicablePlans()` is also \(O(|Opt_{relevant}| \times |B|^n)\), so we will be able to fit the same curve and generate an equation to predict the execution time. After fitting curves to the collected data, we get equation 7.1 for the `perceive()` method, equation 7.2 for the `buf()` method, and 7.3 for the `applicablePlans()` method.

We can simplify equation 7.3 to get equation 7.4.

We also manually sum the execution times of the ten steps that make up the reasoning cycle and subtract that from the actual measured execution time of the reasoning cycle and find that there is about 1900\(\mu\)s unaccounted for by the steps of the reasoning cycle. We will add that to our equation, giving a final equation shown in equation 7.5. We will also simplify equation 7.5 by removing the 0.1\(p\) term as it is extremely small when compared to 2000\(p\), shown in equation 7.6.

\[
t(p) = (0.1p + 200) \quad (7.1)
\]
\[ t(p) = (5p^2 + 2000p) \] (7.2)

\[ t(b, l, p, r) = (l + 100)(50b + 5000)(50b + 500)^{0.001r} \] (7.3)

\[ t(b, l, p, r) = (50bl + 5000l + 5000b + 500000)(50b + 500)^{0.001r} \] (7.4)

\[ t(b, l, p, r) = (0.1p + 200) + (5p^2 + 2000p) + (50bl + 5000l + 5000b + 500000)(50b + 500)^{0.001r} + 1900 \] (7.5)

\[ t(b, l, p, r) = (5p^2 + 2000p) + (50bl + 5000l + 5000b + 500000)(50b + 500)^{0.001r} + 2100 \] (7.6)

where:

- \( b \) is the number of beliefs that exist within the belief base of the agent.
- \( l \) is the number of relevant plans that are available to the agent.
- \( p \) is the number of percepts that have been processed by the agent.
- \( r \) is the number of belief rules that are available to the agent.
- \( t \) is the time in microseconds (\( \mu s \)).

Now that the calibration of our model has been completed, we will determine how many of each parameters exist in the agent. Appendix C.1 contains the .asl files that comprise the agent source code we analyse and Appendix C.2 contains the list of possible percepts the agent can perceive. Our findings are summarized in table 7.8

Using the values in this table, we can substitute into equation 7.5 using the same
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beliefs (b)</td>
<td>30</td>
</tr>
<tr>
<td>Belief Rules (r)</td>
<td>18</td>
</tr>
<tr>
<td>Percepts (p)</td>
<td>9</td>
</tr>
<tr>
<td>Relevant Plans (l)</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 7.8: Parameters Extracted from Mail Robot Agent

<table>
<thead>
<tr>
<th>Function Combination</th>
<th>Predicted Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>2000</td>
</tr>
<tr>
<td>perceive</td>
<td>2200</td>
</tr>
<tr>
<td>buf</td>
<td>22000</td>
</tr>
<tr>
<td>applicablePlans</td>
<td>802000</td>
</tr>
<tr>
<td>perceive/buf</td>
<td>22200</td>
</tr>
<tr>
<td>perceive/applicablePlans</td>
<td>802200</td>
</tr>
<tr>
<td>buf/applicablePlans</td>
<td>822000</td>
</tr>
<tr>
<td>perceive/buf/applicablePlans</td>
<td>822200</td>
</tr>
</tbody>
</table>

Table 7.9: Predicted Execution Times of Mail Robot Agent from Model (in µs)

procedure as the previous two case studies (looking at each of the eight combinations of functions that could possibly be called as part of the reasoning cycle) to get table 7.9.

### 7.3.1 Results of Mail Delivery Robot Execution

Running the mail robot and collecting the profiling data on the reasoning cycle, then categorizing that data into the eight categories outlined earlier in this chapter, we get figure 7.9. Plotting the predicted maximum values and changing the execution time scale to be logarithmic, we get figure 7.10.

As with the other two case studies, we see combinations that do not get run, which is as expected. We do see that the execution time of `perceive() / buf()` is lower than that of `perceive() / buf() / applyFindOp()`, which is as was hypothesized in the last two case studies.
Figure 7.9: Mail Robot: Execution Time vs. Function Called - Raspberry Pi 4

Figure 7.10: Mail Robot: Log Execution Time vs. Function Called - Raspberry Pi 4
Table 7.10: Order of Maximum Execution Time for Mail Robot Agent (in $\mu$s)

<table>
<thead>
<tr>
<th>Study</th>
<th>Raspberry Pi 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mail Robot (Model)</td>
<td>$10^5$</td>
</tr>
<tr>
<td>Mail Robot (Measured)</td>
<td>$10^4$</td>
</tr>
</tbody>
</table>

7.3.2 Discussion of Results

Looking at figure 7.10, we see that the predicted maximum execution time of reasoning cycles that executed `perceive()` and `buf()` was actually slightly below the recorded execution time. It is speculated that perhaps the interaction with ROS might have caused the execution time to increase in a way that could not be predicted by our model. However, we see that reasoning cycles that executed all three functions (`perceive()`, `buf()`, and `applicablePlans()`) executed within an order of magnitude below our predicted values, the same as the previous two case studies.

We also must mention the parameters we extracted from this agent (summarized in table 7.8). The fact that our example of a “real world” agent has such low values for each of its parameters make us consider that perhaps the range we had chosen to generate our model was not appropriate and that a smaller range (perhaps 0 - 100 for each parameter) would have been a better choice.

7.4 Summary

We have looked at three different case studies in this chapter. Two case studies involved an agent running in a simulated environment, where the third had an agent run in a non-simulated environment and interfaced with an external operating system (ROS). The two simulated environment agents were run on both our laptop test environment and a Raspberry Pi 1. Due to the configuration of the mail robot, we
had to run this agent on the hardware that the system was designed with. This meant that we needed to calibrate our model to support a Raspberry Pi 4 before we could generate predictions.

Looking at the three different case studies, we can summarize the order of their execution time both predicted and measured in table 7.11. We see that for all but one, the predicted execution time is one order of magnitude higher than the measured execution time (excluding the laptop grid world study, which had the same order of magnitude for both). Since our model will always take into account the worst case (as we have designed agents to always pick the last plan, run the maximum number of belief rules, etc.), we expect to have a model that predicts values somewhat higher than what is measured. In reality however, an agent will not always execute the worst case, and thus will on average, execute faster than predicted. The fact that the predicted execution time is consistent across three different studies and across three different hardware configurations show that our model is a good first step for understanding the execution time of the reasoning cycle when agents have a known number of beliefs, belief rules, percepts, and plans, but that more work remains to be done to refine the model and its predictive ability.

Table 7.11: Summary of Maximum Order of Execution Time for Case Studies
Chapter 8

Addendum

8.1 Removing the Influence of the Profiler

It has come to our attention that the profiler used has added a large amount of overhead to our model and that a large source of inaccuracy can be attributed to this. We see in figure 8.1 an example of the execution time of a function being profiled vs the Jason source code being instrumented to dump the execution time. In order to eliminate this overhead, we will re-run our experiment on the laptop system with the following modifications:

- We will no longer use the profiler to measure the execution time of the agent
- We will instrument the source code to record the execution time of \texttt{perceive()}, \texttt{buf()}, and \texttt{applyFindOp()}

It is our expectation that we will see a slight increase in the accuracy of our model, as our experimentation would have been affected more by the overhead of the profiler than the case study would be. This is due to the large number of parameters used
8.1.1 Creating an Updated Model

We will follow the same procedure as found in section 6. We will run our instrumented agents and use curves of best fit to get equations 8.1 and 8.3, which can be simplified to equations 8.2 and 8.4. Note that the execution time of \texttt{perceive()} is not measurable at the number of percepts we use as the function completes too quickly. Also, we notice that \texttt{buf()} has no \texttt{p}^2 value any more. We believe that when the profiler is taken out of the system, the execution time of \texttt{buf()} increases slowly enough as to look linear from zero to 1000 percepts. While this might be problematic for extremely complicated agents, we know that the agents we used for our case studies were small.

Figure 8.1: Execution Times of Profiler Compared to Source Code Instrumentation
enough as to fall into this range and so will accept it at this time. Finally, we notice
all of the constants in equation 8.4 have lower values than that of equation 6.15, again
showing the effect the profiler overhead had on the overall execution time.

\[ t(p) = (1 \times 10^{-24}p^2 + 30p + 6000) \] \hspace{1cm} (8.1)

\[ t(p) = (30p + 6000) \] \hspace{1cm} (8.2)

\[ t(b, l, r) = (0.2l + 20)(b + 200)(b + 200)^{0.001r + 3 \times 10^{-08}} \] \hspace{1cm} (8.3)

\[ t(b, l, r) = (0.2bl + 20b + 40l + 4000)(b + 200)^{0.001r} \] \hspace{1cm} (8.4)

\[ t(b, l, p, r) = 30p + (0.2bl + 20b + 40l + 4000)(b + 20)^{0.001r} + 6800 \] \hspace{1cm} (8.5)

where:
b is the number of beliefs that exist within the belief base of the agent.
l is the number of relevant plans that are available to the agent.
p is the number of percepts that have been processed by the agent.
r is the number of belief rules that are available to the agent.
t is the time in microseconds (µs).

8.1.2 Rerun Domestic Robot Case Study

As a proof of concept, we will rerun only the domestic robot case study, originally seen
in section 7.1. We expect our model to be slightly more accurate this time around. As
<table>
<thead>
<tr>
<th>Function Combination</th>
<th>Laptop</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>800</td>
</tr>
<tr>
<td>perceive</td>
<td>800</td>
</tr>
<tr>
<td>buf</td>
<td>1600</td>
</tr>
<tr>
<td>applyFindOp</td>
<td>70800</td>
</tr>
<tr>
<td>perceive/buf</td>
<td>1620</td>
</tr>
<tr>
<td>perceive/applyFindOp</td>
<td>70820</td>
</tr>
<tr>
<td>buf/applyFindOp</td>
<td>71600</td>
</tr>
<tr>
<td>perceive/buf/applyFindOp</td>
<td>71620</td>
</tr>
</tbody>
</table>

Table 8.1: Predicted Execution Times of Domestic Robot Agent from Model (in µs)

done before, we can take the values from table 7.1 and substitute them into equations 8.2 and 8.4 to generate the expected execution time of each combination of executed function, summarized in table 8.1.

Looking at figure 8.2 and figure 8.3, we notice that overall, the predicted execution time is close the the actual measured execution time, even within an order of magnitude for some combinations. When comparing figure 8.3 to figure 7.3, we see that the overall execution time of the two are similar. Since the number of parameters used in the domestic robot are small, we expect that the overhead of the profiler to not affect our measurements much, so both executions of this case study would be expected to be similar.

### 8.2 Summary

We have seen that using a profiler to measure the execution time of the Jason reasoning cycle can potentially add a large amount of overhead and adversely skew our measurements. By instrumenting our code we are able to eliminate that additional overhead and produce a slightly more accurate model.
Figure 8.2: Domestic Robot: Execution Time vs. Function Called - Laptop

![Graph showing execution time vs. function called for domestic robots on laptop, with x-axis labeled "Function Combination (Number of Invocations)" and y-axis labeled "Execution Time (us)".]

Figure 8.3: Domestic Robot: Log Execution Time vs. Function Called - Laptop

![Graph showing log execution time vs. function called for domestic robots on laptop, with x-axis labeled "Function Combination (Number of Invocations)" and y-axis labeled "Execution Time (us)". The graph includes a scatter plot with red dots indicating the predicted maximum execution time.]
Chapter 9

Conclusion and Future Work

9.1 Conclusion

In this work, we have presented an in-depth look at the Jason reasoning cycle. We have started with analyzing the reasoning cycle, as well as breaking it down into its component ten steps and extracting generic algorithms for each step. We have used those algorithms to identify the time complexity of each step, which we have used to determine what aspects of the reasoning cycle most affect its execution time. Through the design of agents that would vary the aspects we identified as having the biggest effect on the reasoning cycle, we experimentally validated our analysis of the reasoning cycle and generated a mathematical model to predict the execution time of the reasoning cycle for a given set of parameters. Finally, we used this model to predict the execution time of three different agents to validate our model.

By using the two test systems we chose (laptop and Raspberry Pi 1), we were able to provide a glimpse at an extremely low-end and a middle of the road systems. We also highlight some other issues that can negatively affect our profiling.
• Other processes: An attempt was made to disable some of the more intensive background processes on the laptop, but it is expected that some of these processes could have potentially introduced additional delay into our profiling. While we expect that an autonomous agent to be running on dedicated hardware, we must acknowledge that our model can be affected by this factor.

• Single-Thread: Our Raspberry Pi is a single-core processor. As such no processes can run in parallel, which means that each process must rely on the scheduler to determine when it is able to run. This can introduce a large amount of variance into our collected data, which in turn makes our model that much less accurate.

• Profiler: We have seen in chapter 8 that profiling the reasoning cycle changes the execution time of the reasoning cycle. If we want to use a profiler, we must explore different options that will allow us to keep this overhead minimized.

• Java: We know that Java periodically runs garbage collection, which can potentially interfere with the execution of the Jason reasoning cycle. Ideally we should disable garbage collection while profiling the reasoning cycle.

During our domestic robot case study, we saw that the laptop has a measured worst case execution of the reasoning cycle in the order of milliseconds ($10^3\mu s$), where the Raspberry Pi executes in the order of 0.1 seconds ($10^5\mu s$). The grid world case study shows the laptop having a worst case execution of the reasoning cycle in the order of 10s of milliseconds ($10^4\mu s$), and the Raspberry Pi executes in the order of seconds ($10^6\mu s$). Adding a third system as part of our mail robot case study allows us to get an idea of how an autonomous robot actually performs. We see that for the system studied, the reasoning cycle executes in the order of 10s of milliseconds ($10^4\mu s$). This is despite having to interface with ROS, having slightly worse hardware
specifications than the laptop, and the mail robot being a more complicated agent design.

Our model has consistently predicted a maximum execution time that was at most one order of magnitude above the measured execution time. This includes three different agents running a combination of three different hardware configurations (domestic robot - laptop and Raspberry Pi 1, grid world - laptop and Raspberry Pi 1, mail robot - Raspberry Pi 4). We should note that in general, a model that trends towards being pessimistic can be considered a desirable trait. This allows for some additional future-proofing if this model is being used to determine if a specific hardware architecture is powerful enough to run a given agent. By removing the profiler and instrumenting the source code, we were able to create a model that was slightly more accurate due to the fact that the overhead of the profiler can be significant when compared to the execution time.

We should also note that it is completely appropriate for our model to predict an execution time that is above what is measured, as our experimental agents were designed to take into account only the worst case. In general, an agent will not always execute the worst case and will, as a result, execute faster than predicted by our model.

Looking at the range of parameters that were extracted from the three case study agents, we have also come to the realization that perhaps the scope of the model is too large. A study of the time complexity of existing agents would be warranted to make sure that future iterations of this model take into account the agent landscape.

We have seen that while the Raspberry Pi 1 might not be a valid system to run an autonomous robot, more modern versions of the Raspberry Pi might be viable for this purpose.
9.2 Limitations

We acknowledge that there are some limitations to the work that has been presented, as we will expand upon.

- The reasoning as to why `buf()` takes multiple reasoning cycles to add large numbers of percepts to the belief base is unknown. It is expected this might be a bug, but more investigation is warranted as it could potentially cause reasoning cycles to be wasted.

- When we tried to model `applyExecInt()`, our experiments showed that there was extremely low impact between the parameters we changed and the execution time of the function. This function does many different things based on the body of the plans that are available to the agent and should be looked at in more depth to see if it has a real impact on the execution time of the reasoning cycle.

- While we ran our experimentation on two different hardware configurations, we do not take into account cache size, memory size, memory speed or any other aspects of their architecture. This is for two reasons:

  1. We view this as the first iteration of this model and felt that the parameters used in the reasoning cycle were the most important factor to focus on.

  2. We are interested in answering the question as to whether Jason is appropriate to be run on a given hardware configuration, not provide performance metrics for multiple hardware configurations.
9.3 Future Work

As part of the research done into analysing the execution time of the Jason reasoning cycle, we have identified other areas that should be covered, to get a better understanding of the reasoning cycle.

- While we manually ran calibration to determine the equations governing the execution time of the reasoning cycle for different hardware configurations, it would be beneficial to generalize this model to consider some of the more relevant hardware factors (for example: processor speed, number of cores, amount of memory). This would allow for an easier determination as to whether Jason is an appropriate solution for a given problem.

- Comparing the execution time of the Jason reasoning cycle to other languages would give a clearer picture as to how Jason fits in with other agent-based programming languages. We looked at a comparison between 2APL, GOAL and Jason in chapter 3, but that comparison was based on a simple agent. Looking at a set of more complex agents, as well as modelling the execution time of the reasoning cycle of other languages would be beneficial in helping determine what language would be best for a circumstance.

- One major problem we have seen with our data gathering has been the large amount of spread present in the gathered data, especially at larger values of the parameters we were interested in. While this might be the result of the Raspberry Pi having only one core or the laptop having other background processes that were interfering with the Jason process, it would be a benefit to investigate this more closely to see if there are ways to limit the spread of data and, therefore, improving the accuracy of the model.
• This work has identified that the number of percepts can be a potential bottleneck when running `buf()` and the number of belief rules an agent has can potentially be a major bottleneck when running `applyFindOp()` / `applicablePlans()`, but has not provided any recommendations on how to mitigate or minimize them. While it is useful to know how long the reasoning cycle is expected to take, knowing how to decrease that value would also be useful.
Bibliography


(Last accessed: 2021-07-22).
Appendix A

Appendix: Domestic Robot

A.1 Agent Source

/* Initial beliefs and rules */

// initially, I believe that there is some beer in the fridge
available(beer,fridge).

// my owner should not consume more than 10 beers a day :-)
limit(beer,1000).

too_much(B) :-
   .date(YY,MM,DD) &
   .count(consumed(YY,MM,DD,_,_,_,B),QtdB) &
   limit(B,Limit) &
   QtdB > Limit.

/* Plans */

+!has(owner,beer)
   : available(beer,fridge) & not too_much(beer)
   <- !at(robot,fridge);
       open(fridge);
get(beer);
close(fridge);
!at(robot,owner);
hand_in(beer);
?has(owner,beer);
// remember that another beer has been consumed
.date(YY,MM,DD); .time(HH,NN,SS).
//+consumed(YY,MM,DD,HH,NN,SS,beer).

+!has(owner,beer)
  : not available(beer,fridge)
  <- .send(supermarket, achieve, order(beer,5));
  !at(robot,fridge). // go to fridge and wait there.

+!has(owner,beer)
  : too_much(beer) & limit(beer,L)
  <- .concat("The Department of Health does not allow me to give you more than ", L,
            " beers a day! I am very sorry about that!",M);
            .send(owner,tell,msg(M)).

-!has(_,_)
  : true
  <- .current_intention(I);
  .print("Failed to achieve goal '+!has(_,_)'. Current intention is: ",I).

+!at(robot,P) : at(robot,P) <- true.
+!at(robot,P) : not at(robot,P)
  <- move_towards(P);
  !at(robot,P).

// when the supermarket makes a delivery, try the 'has' goal again
+delivered(beer,_,Qtd,OrderId)[source(supermarket)]
  : true
  <- +available(beer,fridge);
  !has(owner,beer);
  .abolish(delivered(_,_,_,_)).
// when the fridge is opened, the beer stock is perceived
// and thus the available belief is updated
+stock(beer,0)
    : available(beer,fridge)
    <- ~available(beer,fridge).
+stock(beer,N)
    : N > 0 & not available(beer,fridge)
    <- ~+available(beer,fridge).

+?time(T) : true
    <- time.check(T).

A.2 Percept List

    public static final Literal of = Literal.parseLiteral("open(fridge)");
    public static final Literal clf = Literal.parseLiteral("close(fridge)");
    public static final Literal gb = Literal.parseLiteral("get(beer)");
    public static final Literal hb = Literal.parseLiteral("hand_in(beer)");
    public static final Literal sb = Literal.parseLiteral("sip(beer)");
    public static final Literal hob = Literal.parseLiteral("has(owner,beer)");
    public static final Literal af = Literal.parseLiteral("at(robot,fridge)");
    public static final Literal ao = Literal.parseLiteral("at(robot,owner)");
Appendix B

Appendix: Grid World

B.1 Agent Source

B.1.1 Navigator Environment Assist

// Demo program of Jason based navigation using A*

ramTo(d).
s.start.

/*
 // Benchmark version
 +path(Path)
 : startTime(StartTime)
 <-// Print the results
 .print("solution A* =", Path, " in ", (system.time - StartTime), " ms.");
 +done.

 +navigate(Destination)
 : (not done)
 <--startTime(system.time); // Get initial time stamp, for benchmarking performance
 getPath(a, Destination);
 !navigate(Destination).
+!navigate(_,_) <- .print("Done").
*/

{ include("obstacleHandler.asl")}
{ include("batteryManager.asl") }

+!start
<- .wait(10000);
!missionTo(d).

+!missionTo(Destination)
<+-+missionTo(Destination)
!navigate(Destination).//;
//.stopMAS.

// Perception of a path provided by the environment based navigation support
+path(Path)
: startTime(Start)
<- .print((system.time - Start), " ms route("Path")");
.broadcast(tell, path(elapsed(system.time - Start), route(Path)));
-route(_); // Get rid of any old routes.
+route(Path).

// Case where we are already at the destination
+!navigate(Destination)
: position(X,Y) & locationName(Destination,[X,Y]) & startTime(Start)
<- .broadcast(tell, navigate(elapsed(system.time - Start), arrived(Destination)));
-destination(Destination).

// We have a route path, set the waypoints.
+!navigate(Destination)
: route(Path)
& startTime(Start)
<- .broadcast(tell, navigate(elapsed(system.time - Start), route(Path)));
for (.member(NextPosition, Path)) {
!waypoint(NextPosition);
-route(Path);

!navigate(Destination).

// We don’t have a route plan, get one.
+!navigate(Destination)
: position(X,Y) & locationName(Current,[X,Y])
<--Start = system.time;
+startTime(Start); // Get initial time stamp, for benchmarking performance
.broadcast(tell, navigate(elapsed(system.time - Start), getPath(Destination)));
.broadcast(tell, navigate(elapsed(system.time - Start), current(Current)));
+destination(Destination);
getPath(Current,Destination);
!navigate(Destination).

+!navigate(Destination)
: startTime(Start)
<-.broadcast(tell, navigate(elapsed(system.time - Start), default));
!navigate(Destination).

// Move through the map, if possible.
+!waypoint(NextPosition)
: position(X,Y) & locationName(Current,[X,Y])
& possible(Current,NextPosition)
& direction(Current,NextPosition,Direction)
& map(Direction)
& not obstacle(Direction)
& startTime(Start)
<-.broadcast(tell, waypoint(elapsed(system.time - Start), move(Destination,NextPosition)));
move(Direction).

// Move through the map, if possible.
+!waypoint(NextPosition)
: position(X,Y) & locationName(Current,[X,Y])
//& possible(Current,NextPosition)
& direction(Current,NextPosition,Direction)
& map(Direction)
& obstacle(Direction)
& startTime(Start)
<-.broadcast(tell, waypoint(elapsed(system.time - Start), obstacle(NextPosition)));
!updateMap(NextPosition).

// Deal with case where Direction is not a valid way to go.
+!waypoint(Next)
  : startTime(Start)
<-.broadcast(tell, waypoint(elapsed(system.time - Start), default)).

// Revisit map update later.

+!updateMap(NextName)
  : position(X,Y) & locationName(PositionName, [X,Y])
  & missionTo(Destination)
  & startTime(Start)
<-.broadcast(tell, updateMap(elapsed(system.time - Start), obstacle(NextName)));
-possible(PositionName,NextName);
setObstacle(PositionName,NextName);
.drop_all_intentions;
!missionTo(Destination).

//+!updateMap(NextName)
// : startTime(Start)
// <-.broadcast(tell, updateMap(elapsed(system.time - Start), default)).
// !updateMap(NextName).

// Get the direction of the next movement
direction(Current,Next,up)
:-possible(Current,Next)
& locationName(Current,[X,Y])
& locationName(Next,[X,Y-1]).

// Get the direction of the next movement
direction(Current,Next,down)
:-possible(Current,Next)
& locationName(Current,[X,Y])
& locationName(Next,[X,Y+1]).

// Get the direction of the next movement
direction(Current,Next,left)
:-possible(Current,Next)
& locationName(Current,[X,Y])
& locationName(Next,[X-1,Y]).

// Get the direction of the next movement
direction(Current,Next,right)
:-possible(Current,Next)
& locationName(Current,[X,Y])
& locationName(Next,[X+1,Y]).

// Map of locations that the agent can visit.
{ include("map.asl") }
B.1.2 Obstacle Handler

safety(pedestrian).
@pedestrianAvoidance [atomic]
+pedestrian(_)
<-.broadcast(tell, pedestrian(honk(horn)));
.wait(1000);
honk(horn).

map(obstacle).
+obstacle(Direction)
: position(X,Y)
& locationName(Current, [X,Y])
& possible(Current,Next)
& direction(Current,Next,Direction)
& mission(Goal,Parameters)
<--possible(Current,Next);
//setObstacle(Current,Next);
//.broadcast(tell, obstacle(Direction));
.drop_all_intentions;
!mission(Goal,Parameters).
B.1.3 Battery Manager Source

batteryMin(2).
batteryMax(20).

/**
 * When the battery perception is below the minimum, we need to pickup the goal
 * of !chargeBattery, if we haven’t already.
 */

health(battery).

// Check if I’m on a mission, if so, I’ll need to launch the mission again
// once the battery is charged.
// Also need to confirm that I’m not already charging the battery.
+battery(State)
  : charging(false)
& lowBattery(State)
& mission(Goal,Parameters)
& (not managingBattery)
<-.drop_all_intentions;
.broadcast(tell, battery(chargingNeeded));
!chargeBattery;
.broadcast(tell, battery(chargingFinished));
!mission(Goal,Parameters).

// No mission on the go, just need to charge the battery if I’m not already
// dealing with it.
+battery(State)
  : charging(false)
& lowBattery(State)
& (not managingBattery)
<-.drop_all_intentions;
.broadcast(tell, battery(chargingNeeded));
!chargeBattery;
.broadcast(tell, battery(chargingFinished)).

/**
 * !chargeBattery
The plan for getting the battery to battery(full) if needed.

\[
\text{health(chargeBattery)}.
\]

// I'm not docked. Go to the docking station and dock.
// This is recursive as we need to wait for the battery to charge.
+!chargeBattery
  : lowBattery(_)
  & charging(false)
  & chargerLocation(ChargeStation)
  <- .broadcast(tell, chargeBattery(chargingNeeded));
+managingBattery;
!navigate(ChargeStation);
.broadcast(tell, chargeBattery(atDock));
station(dock);
.broadcast(tell, chargeBattery(docked));
!chargeBattery.

// Battery is full, undock the robot
+!chargeBattery
  : fullBattery(_)
  & charging(true)
  <- .broadcast(tell, chargeBattery(charged));
station(undock);
-managingBattery;
.broadcast(tell, chargeBattery(unDocked)).

+!chargeBattery
  <- .broadcast(tell, chargeBattery(waiting));
!chargeBattery.

lowBattery(State)
  :- battery(State)
  & batteryMin(Min)
  & State \leq Min.

fullBattery(State)
:- battery(State)
& batteryMax(State).

atStation
:- position(X,Y)
& locationName(ChargeStation,[X,Y])
& chargerLocation(ChargeStation).
B.1.4 Movement

movement(waypoint).

// Move through the map, if possible.
+!waypoint(NextPosition)
 : position(X,Y) & locationName(Current,[X,Y])
 & possible(Current,NextPosition)
 & direction(Current,NextPosition,Direction)
 & map(Direction)
 & not obstacle(Direction)
 & startTime(Start)
 <-.broadcast(tell, waypoint(elapsed(system.time - Start), move(Destination,NextPosition)));
 .wait(1000);
 move(Direction).

// Deal with case where Direction is not a valid way to go.
+!waypoint(Next)
 : startTime(Start)
 <-.broadcast(tell, waypoint(elapsed(system.time - Start), default)).

// Get the direction of the next movement
direction(Current,Next,up)
 :-possible(Current,Next)
 & locationName(Current,[X,Y])
 & locationName(Next,[X,Y-1]).

// Get the direction of the next movement
direction(Current,Next,down)
 :-possible(Current,Next)
 & locationName(Current,[X,Y])
 & locationName(Next,[X,Y+1]).

// Get the direction of the next movement
direction(Current,Next,left)
 :-possible(Current,Next)
 & locationName(Current,[X,Y])
 & locationName(Next,[X-1,Y]).
// Get the direction of the next movement
direction(Current,Next,right)
:-possible(Current,Next)
& locationName(Current,[X,Y])
& locationName(Next,[X+1,Y]).
B.1.5 Map

/**
 * Definition of the map:
 * The map is a 4 x 4 grid world. The position of the grid location is
 * encoded as [X,Y], where the bottom left corner is [0,0] and the top right
 * corner is [3,3].
 *
 * The map has accessible and inaccessible squares. A visual view of the map is
 * below. The inaccessible square have an X. (start without that)
 *
 * |-------------------------------|
 * | m | n | o | p |
 * |   |   |   | X |
 * | [0,0] | [1,0] | [2,0] | [3,0] |
 * |-------------------------------|
 * | i | j | k | l |
 * |   |   |   | X |
 * | [0,1] | [1,1] | [2,1] | [3,1] |
 * |-------------------------------|
 * | e | f | g | h |
 * |   |   |   | X |
 * | [0,2] | [1,2] | [2,2] | [3,2] |
 * |-------------------------------|
 * | a | b | c | d |
 * | C |   |   | X |
 * | [0,3] | [1,3] | [2,3] | [3,3] |
 * |-------------------------------|
 */

locationName(a,[0,3]).
locationName(b,[1,3]).
locationName(c,[2,3]).
locationName(d,[3,3]).
locationName(e,[0,2]).
locationName(f,[1,2]).
locationName(g,[2,2]).
locationName(h,[3,2]).
locationName(i,[0,1]).
locationName(j,[1,1]).
locationName(k,[2,1]).
locationName(l,[3,1]).
locationName(m,[0,0]).
locationName(n,[1,0]).
locationName(o,[2,0]).
locationName(p,[3,0]).

// Location of the charger
chargerLocation(a).

// Possible map transitions.
// possible(StartingPosition, PossibleNewPosition)
possible(a,b).
possible(a,e).
possible(b,a).
possible(d,h).
possible(e,a).
possible(e,i).
possible(g,h).
possible(h,d).
possible(h,l).
possible(i,e).
possible(i,m).
possible(k,g).
possible(k,l).
possible(k,o).
possible(1,k).
possible(1,h).
possible(m,i).
possible(m,n).
possible(n,m).
possible(n,o).
possible(o,n).
possible(o,k).

// Mess with the map, tell that C is a possible place to go
possible(b,c).
possible(c,d).
B.2 Grid Agent Percept List

```java
public static final Term moveUp = Literal.parseLiteral("move(up)" userProfile);
public static final Term moveDown = Literal.parseLiteral("move(down)" userProfile);
public static final Term moveLeft = Literal.parseLiteral("move(left)" userProfile);
public static final Term moveRight = Literal.parseLiteral("move(right)" userProfile);
public static final Term dock = Literal.parseLiteral("station(dock)" userProfile);
public static final Term undock = Literal.parseLiteral("station(undock)" userProfile);
public static final Term honkHorn = Literal.parseLiteral("honk(horn)" userProfile);
```
Appendix C

Appendix: Mail Robot

C.1 Agent Source

C.1.1 Map

/**
 * Map
 * 
 * a
 * | 
 * | 
 * d -- b
 * | 
 * | 
 * c
 */

// Possible routes between locations
possible(a,b).
possible(b,a).
possible(b,c).
possible(c,b).
possible(b,d).
possible(d,b).

// Location definitions
locationName(a,[1,1]).
locationName(b,[1,0]).
locationName(c,[1,-1]).
locationName(d,[0,0]). // Robot has to start here, facing b.

// Successor state
suc(Current,Next,Range,drive)
:-possible(Current,Next)
& locationName(Current,[X1,Y1])
& locationName(Next,[X2,Y2])
& range(X1,Y1,X2,Y2,Range).

// Heuristic definition: h(CurrentState,Goal,H)
h(Current,Goal,Range)
:-locationName(Current,[X1,Y1])
& locationName(Goal,[X2,Y2])
& range(X1,Y1,X2,Y2,Range).

// Range
range(X1,Y1,X2,Y2,Range)
:-Range = math.sqrt(((X2-X1) * (X2-X1)) + ((Y2-Y1) * (Y2-Y1))).

atLocation(Location,0)
:-locationName(Location,[X,Y])
& position(X,Y).

// Initial beliefs for position and direction faced.
position(0,0). // Initial location is [0,0]
direction(e). // See below

// Directions
// n
// |
// w -- + -- e
// l
// s
C.1.2 Obstacle Avoidance

safety(bumper).
@Bumper [atomic]
+bumper(pressed)
<- .broadcast(tell, avoid(obstacle));
!turn(left);
!move(0.5,-3,10);
!turn(left).

C.1.3 Mission

/**
 * Primary mission: collect and deliver mail
 * Plan is simple:
 * (1) Make mental note that I have a mail mission
 * (2) Collect the mail
 * (3) Deliver the mail
 * (4) Delete mental note about mail mission
 * Updating the user at every step.
 */

mission(mission).
+!mission(Goal,Parameters)
  : Goal = 'collectAndDeliverMail'
  & Parameters = [Sender,Receiver]
<- +mission(Goal,Parameters);
  !collectAndDeliverMail(Sender,Receiver);
-!mission(Goal,Parameters).

mission(collectAndDeliverMail).
+!collectAndDeliverMail(Sender,Receiver)
<- .broadcast(tell, mailUpdate(receivedMission,Sender,Receiver));
!collectMail(Sender);
.broadcast(tell, mailUpdate(gotMail,Sender,Receiver));
!deliverMail(Receiver);
.broadcast(tell, mailUpdate(delivered,Receiver)).
/**
 * !collectMail(Sender)
 * Go get the mail from the sender
 */
mission(collectMail).

// I do not yet have the mail, go get it.
+!collectMail(Sender)
: not haveMail
<-.broadcast(tell, collectMail(goToSender,Sender));
!navigate(Sender);
.broadcat(tell, collectMail(atSender,Sender));
+haveMail;
.broadcat(tell, collectMail(haveMail,Sender)).

// I already have the mail.
+!collectMail(Sender)
: haveMail
<-.broadcast(tell, collectMail(alreadyHaveMail,Sender)).

/**
 * !deliverMail(Receiver)
 * Give the mail to the receiver
 */
mission(deliverMail).

// Case where I have the mail.
+!deliverMail(Receiver)
: haveMail
<-.broadcast(tell, deliverMail(goToReceiver,Receiver));
!navigate(Receiver);
.broadcat(tell, deliverMail(arrivedAtReceiver,Receiver));
-haveMail;
.broadcat(tell, deliverMail(delivered,Receiver)).

// Case where there is no mail to deliver.
+!deliverMail(Receiver)
: not haveMail
<-broadcast(tell, deliverMail(noMail,Receiver)).

/**
 * Default plans.
 * These can run when unrelated perceptions are received, resulting in a
 * reasoning cycle where no plan context is applicable for that reasoning cycle.
 */
// Ensure recursion and remember the parameters.
// Note: This plan should never run as there are no context guards in the
// main plan for this goal above. If this plan runs, something strange
// happened.
+!collectAndDeliverMail(Sender,Receiver)
<-.broadcast(tell, collectAndDeliverMail(default));
+!collectAndDeliverMail(Sender,Receiver).

// Default plan for collect mail. Should not be possible to get here.
+!collectMail(Sender)
<-.broadcast(tell, collectMail(error,Sender)).

// Default plan for deliver mail. Should not be possible to get here.
+!deliverMail(Receiver)
<-.broadcast(tell, deliverMail(error,Receiver)).

C.1.4 Movement

// Component behaviours (think in terms of state)

movement(waypoint).
+!waypoint(Location)
  :
  atLocation(Location, _)
  <- .broadcast(tell, waypoint(atLocation(Location)));
drive(stop).

+!waypoint(Location)
  : locationName(Location, [X, Y])
  <- .broadcast(tell, waypoint(Location));
!faceNext(Location);
!drive(forward);
-\text{position}(\text{_,}_); \\
+\text{position}(X,Y); \\
!\text{waypoint}(\text{Location}).

+!\text{waypoint}(\text{Location}) \\
<- .\text{broadcast}(\text{tell, waypoint}(\text{Location, default})).

\text{movement}(\text{faceNext}). \\
+!\text{faceNext}(\text{Next}) \\
: \text{direction}(\text{CurrentDirection}) \\
& \text{directionToNext}(\text{Next, NewDirection}) \\
& \text{face}(\text{CurrentDirection, NewDirection, TurnAction}) \\
<- .\text{broadcast}(\text{tell, faceNext}(\text{Next})); \\
!\text{turn}(\text{TurnAction}); \\
-\text{direction}(\text{_}); \\
+\text{direction}(\text{NewDirection}).

+!\text{faceNext}(\text{Next}) \\
<- .\text{broadcast}(\text{tell, faceNext}(\text{Next, default})).

directionToNext(\text{Next, e}) \\
: -\text{position}(X1, _) \\
& \text{locationName}(\text{Next, [X2, _]}) \\
& \text{X1 < X2}.

directionToNext(\text{Next, w}) \\
: -\text{position}(X1, _) \\
& \text{locationName}(\text{Next, [X2, _]}) \\
& \text{X1 > X2}.

directionToNext(\text{Next, n}) \\
: -\text{position}(\text{_, Y1}) \\
& \text{locationName}(\text{Next, [_, Y2]}) \\
& \text{Y1 < Y2}.

directionToNext(\text{Next, s})
position(_,Y1)
& locationName(Next,[_Y2])
& Y1 > Y2.

face(OldDirection, NewDirection, around)
:- ((OldDirection == s) & (NewDirection == n))
| ((OldDirection == n) & (NewDirection == s))
| ((OldDirection == e) & (NewDirection == w))
| ((OldDirection == w) & (NewDirection == e)).

face(OldDirection, NewDirection, left)
:- ((OldDirection == s) & (NewDirection == e))
| ((OldDirection == e) & (NewDirection == n))
| ((OldDirection == n) & (NewDirection == w))
| ((OldDirection == w) & (NewDirection == s)).

face(OldDirection, NewDirection, right)
:- ((OldDirection == s) & (NewDirection == w))
| ((OldDirection == w) & (NewDirection == n))
| ((OldDirection == n) & (NewDirection == e))
| ((OldDirection == e) & (NewDirection == s)).

face(OldDirection, NewDirection, ok)
:- OldDirection == NewDirection.

movement(turn).
+!turn(left)
  : turnRate(X,Y)
  <- broadcast(tell, turn(left));
  !move(X,Y,2).

+!turn(right)
  : turnRate(X,Y)
  <- broadcast(tell, turn(right));
  !move(0-X,0-Y,2).

+!turn(around)
: turnRate(X,Y)
<-.broadcast(tell, turn(around));
!move(X,Y,4).

+:!turn(_)
<-.print(turn(default)).

movement(drive).
+:!drive(forward)
  : driveRate(X,Y)
  <-.broadcast(tell, drive(forward));
  !move(X,Y,5).

+:!drive(backward)
  : driveRate(X,Y)
  <-.broadcast(tell, drive(backward));
  !move(0-X,0-Y,5).

+:!drive(_)
<-.print(drive(default)).

movement(move).
+:!move(X,Y,Count)
  : Count > 0
  <-.broadcast(tell, move(X,Y,Count));
  drivexy(X,Y);
  !move(X,Y,Count-1).

+:!move(_,_,_)
<-.print(move(default)).

/**
 * Turning and driving rates
 */
turnRate(0,4.1).
driveRate(0.5,0).
C.1.5 Navigation

+!mission(Goal,Parameters)
: Goal = navigate
& Parameters = [Destination]
<-+mission(Goal,Parameters);
!navigate(Destination);
-mission(Goal,Parameters).

/**
 * !navigate(Destination)
 * Used for setting up the navigation path to get from the current location to
 * the destination.
 * Beliefs: Relevant map definitions in map.asl
 * Actions: None
 * Goals Adopted: !driveToward(Location)
 */

navigation(navigate).

// Case where we are already at the destination
+!navigate(Destination)
: atLocation(Destination, Range)
<- .broadcast(tell, navigate(arrived(Destination, Range))).

// We don’t have a route plan, get one and set the waypoints.
+!navigate(Destination)
: (not atLocation(Destination, _))
& atLocation(Current, Range)
<- .broadcast(tell, navigate(gettingRoute(Destination), Range));
.broadcast(tell, navigate(current(Current), Range));
?a_star(Current, Destination, Solution, Cost);
.broadcast(tell, navigate(route(Solution, Cost), Destination, Range));
for (.member( op(drive, NextPosition), Solution)) {
    !waypoint(NextPosition);
}
!navigate(Destination).

// !navigate(Destination) - should be impossible
+!navigate(Destination)
  <-.broadcast(tell, navigate(default, Destination)).

// A* Nav Rules
{ include("/home/pi/create_ws/src/saviRoomba/asl/a_star.asl") }

C.1.6 Battery Manager

batteryMin(0.50).
batteryMax(0.95).
charging(false).

/**
 * When the battery perception is below the minimum, we need to pickup the goal
 * of !chargeBattery, if we haven’t already.
 */

health(battery).

// Check if I’m on a mission, if so, I’ll need to launch the mission again
// once the battery is charged.
// Also need to confirm that I’m not already charging the battery.
+battery(State)
  : charging(false)
  & lowBattery(State)
  & mission(Goal,Parameters)
  & (not managingBattery)
  <-.drop_all_intentions;
  .broadcast(tell, battery(chargingNeeded));
  !chargeBattery;
  .broadcast(tell, battery(chargingFinished));
  !mission(Goal,Parameters).

// No mission on the go, just need to charge the battery if I’m not already
// dealing with it.
+battery(State)
  : charging(false)
  & lowBattery(State)
& (not managingBattery)
<-.drop_all_intentions;
.broadcast(tell, battery(chargingNeeded));
!chargeBattery;
.broadcast(tell, battery(chargingFinished)).

/**
 * !chargeBattery
 * The plan for getting the battery to battery(full) if needed.
 */

health(chargeBattery).

// I'm not docked. Go to the docking station and dock.
// This is recursive as we need to wait for the battery to charge.
+!chargeBattery
: lowBattery(_)
& charging(false)
& chargerLocation(ChargeStation)
<-.broadcast(tell, chargeBattery(chargingNeeded));
+managingBattery;
!navigate(ChargeStation);
.broadcast(tell, chargeBattery(atDock));
station(dock);
-charging(false);
+charging(true);
.broadcast(tell, chargeBattery(docked));
!chargeBattery.

// Battery is full, undock the robot
+!chargeBattery
: fullBattery(_)
& charging(true)
<-.broadcast(tell, chargeBattery(charged));
station(undock);
-charging(true);
+charging(false);
-chargeBattery;
-managingBattery;
.broadcast(tell, chargeBattery(unDocked)).

!chargeBattery
<- .broadcast(tell, chargeBattery(waiting));
!chargeBattery.

lowBattery(State)
:- battery(State)
& batteryMin(Min)
& State <= Min.

fullBattery(State)
:- battery(State)
& batteryMax(State).

atStation
:- position(X,Y)
& locationName(ChargeStation,[X,Y])
& chargerLocation(ChargeStation).

C.1.7 A Star

/**
 * A* implementation - From Jason project website
 */

{ register_function("search.h",2,"h") }

a_star( InitialState, Goal, Solution, Cost) :-
  .set.add_all(Closed,[InitialState]) &
  .queue.create(Open,priority) &
  .queue.add(Open,s(0,0,[op(initial,InitialState)]) &
  a_star_l( Open, Goal, s(_,Cost,SolutionR), Closed) &
  .reverse(SolutionR,Solution).

a_star_l( Open, GoalState, s(F,G,[op(Op,GoalState)|Path]), Closed) :-
  .queue.head(Open,s(F,G,[op(Op,GoalState)|Path])).
a_star_l(Open, GoalState, Solution, Closed) :- 
.queue.remove(Open, s(F, G, [op(Op, State)|Path])) & 
//.print("exploring ",State," to be explored ", .length(Open)) & 
State \== GoalState & 
.set.add(Closed, State) & // do not explore State again 
.findall(
   s(NF, NG,[ op(NOp, NewState), op(Op, State)|Path]), // new paths 
   ( suc(State, NewState, Cost, NOp) & 
     not .member(NewState, Closed) & 
     NG = G + Cost & // cost to achieve NewState 
     NF = search.h(NewState, GoalState) + NG // cost + heuristic for new state 
   ), 
   Suc 
) & 
//.print(" ",Suc) & 
.queue.add_all(Open, Suc) & 
a_star_l(Open, GoalState, Solution, Closed).
C.2 Mail Robot Agent Percept List

batteryPerception = ""
bumperPerception = ""
odomPositionPerception = ""
odomOrientationPerception = ""
updateReady = [False, False, False, False]
batteryIndex = 0
bumperIndex = 1
odomPositionIndex = 2
odomOrientationIndex = 3