Jess to JADE Toolkit (J2J)
A Rule-Based Solution Supporting Intelligent and Adaptive Agents

Master Thesis

Joël Vogt
August 2008

Thesis supervisors:
Prof. Dr. Jacques Pasquier-Rocha
and
Dr. Patrik Fuhrer
Software Engineering Group
Marco Savini
Information Systems Research Group
“May fortune favor the foolish.”

Admiral James T. Kirk

“Never send a human to do a machine’s job.”

Agent Smith
I shall use this first Section of my thesis to convey my profound gratitude to those who have supported me during this thesis and during my studies. First my loving family: without your encouragement and patients I would never had the privilege to study at the University of Fribourg.

I shall continue with my excellent supervisors Dr. Patrik Fuhrer and Marco Savini. They have greatly helped to define my thesis and offered most valuable input. I also thank Professor Dr. Jacques Pasquier and Professor Dr. Andreas Meier for letting me write this thesis in their research groups. Next is Minh Tuan Guyen at the Software Engineering Group. He was always open to help and gave valuable assistance. I shall continue with my fellow colleagues at the Information Systems Research Group, who were ever there to help when I needed advice and I thank again Professor Dr. Andreas Meier for offering me an employment in his Research Group.

A special thanks to my father Markus, Daniel Fasel and Darius Zumstein for proof-reading my thesis. And also many thanks to Andreas Ruppen, the developer of the MediMAS project. He was always ready to offer help and share his knowledge.

I shall not forget those who, three and a half years ago, have persuaded me to attend university. I express my thankfulness to Professor Dr. Knut Hinkelmann for his tireless support during my bachelor thesis. I shall furthermore express my great appreciation to Sybille Bühler, Professor Sabine Künzi, Professor Dr. Trung Dinh from the Fachhochschule Nordwestschweiz and Dr. Serge Reichlin and Dr. Tony Dyson from Medgate. And of course, last but not least, my utmost thanks to my friends!
Abstract

The motivation for this thesis came with the aim to develop a flexible and extensible workflow automation tool kit for the healthcare domain by combining multi-agent and rule-based technologies. The rule-based paradigm offers a natural way to codify knowledge and promotes reusability. A multi-agent system provides an extensible and scalable environment for distributed systems. These technologies combined, provide a flexible, extensible and scalable distributed system.

The frameworks chosen are the rule-based system Jess and the open source multi-agent framework JADE. Both are implemented in Java. Jess is a fast rule-based system shell than can contain large numbers of rules. Because rules are interpreted, rules can be updated, deleted and added during run-time. JADE is fully FIPA-standard compliant and has support for platforms with limited resources, such as mobile devices. Since both frameworks are written in Java, almost any modern platform is supported.

The tool kit developed for this thesis is called the Jess to JADE (J2J) tool kit. It is not restricted to process automation but can be applied to variety of applications that decide to use the rule-based paradigm to codify the decision making of JADE agents.

The J2J tool kit is managed by a single class. Agents that use Jess to reason do not extend that class, instead they are each service users of their own J2J object. This permits to add the J2J tool kit to existing JADE multi-agent environments, without “breaking” the existing class hierarchies.

Furthermore, a generic representation of the JADE domain specific ontology is created by the J2J tool kit. This generic ontology representation is decoupled from the logic to create concrete representation of the ontology, such as Jess templates. Instead, the mapping to a specific format is conducted by a given Visitor object. This approach has the advantage that the domain specific ontology is not limited to a single application and it assures interoperability, since the knowledge artifacts of the subsystems are based on the same semantics.

In addition, the J2J tool kit offers a command line interface to remotely access the Jess interface of an agent. Through this interface, the rules and the working memory of the intelligent agent’s Jess instance can be altered and monitored during run time.

**Keywords:** Multi-Agent Systems, Rule-Based Systems, Jess, JADE, Java, Process Management
# Table of Contents

1 Introduction
   1.1 Motivation and Goals ........................................ 2
   1.2 Schedule .................................................. 3
   1.3 Organization ............................................... 4
   1.4 Notations and Conventions ................................. 4

I Motivation ......................................................... 6

2 Business Process Modeling ....................................... 7
   2.1 From Functional to Process-Oriented Organization ....... 7
   2.2 Business Process Modeling .................................. 7
      2.2.1 Event-Driven Process Chains ......................... 8
      2.2.2 Activity Diagrams ................................... 10
      2.2.3 Comparing Event-driven Process Chains and Activity Diagrams 12

3 Case Study: Hospital and Laboratory ........................... 15
   3.1 Regular Case ............................................. 17
   3.2 Physician Calling the Laboratory Assistant .............. 17
   3.3 Laboratory Assistant Calling Physician .................. 19
   3.4 Laboratory Assistant has Critical Laboratory Results .... 20

II Background ...................................................... 22

4 Software Agents .................................................. 23
   4.1 What is an Agent? What is an Intelligent Agent? ......... 24
      4.1.1 Properties of an Agent ............................... 24
      4.1.2 Properties of an Agent Environment ................... 24
   4.2 Architectures .............................................. 25
### Table of Contents

4.2.1 Logic Based Agents Architectures ........................................... 26  
4.2.2 Reactive Agent Architectures ................................................. 26  
4.2.3 Hybrid Agent Architectures ................................................... 26  
4.2.4 BDI Architectures ............................................................. 27  
4.3 Communication and Coordination ............................................... 27  
4.3.1 FIPA Message Specifications ................................................ 28  
4.3.2 FIPA-Protocols ................................................................. 28  
4.4 Ontologies in a Multi-Agent Environment ..................................... 30  
4.5 Applications .............................................................................. 31  
4.5.1 Multi-Agent System for Electronic Health Record Management ........ 31  
4.5.2 Multi-Agent System for Process Automation ............................... 31  
4.5.3 Multi-Agent System for Sensor Web Management ........................ 31  

5 JADE: An Agent Framework .......................................................... 33  
5.1 The JADE Architecture ............................................................ 34  
5.1.1 Important Platform Management Services ................................ 34  
5.1.2 Administration and Debugging Tools ....................................... 35  
5.2 Agent Communication Implementation ....................................... 36  
5.2.1 Sending and Receiving Messages ............................................ 36  
5.2.2 Implementational Details of the JADE Messaging Service ............ 36  
5.2.3 Behaviors ........................................................................... 38  
5.3 Use of Ontologies in JADE ........................................................ 39  
5.3.1 Protégé .............................................................................. 41  

6 Rule-Based Systems ..................................................................... 42  
6.1 Rules ....................................................................................... 43  
6.2 The Architecture of Rule-Based Systems .................................... 44  
6.2.1 Inference Engine .................................................................. 44  
6.2.2 Working Memory .................................................................. 45  
6.2.3 Rule Base ........................................................................... 45  
6.2.4 Pattern Matcher ................................................................... 46  
6.2.5 Agenda .............................................................................. 47  
6.2.6 Execution Engine ................................................................. 47  
6.3 Applications .............................................................................. 47  
6.3.1 Rule-Based Expert System in the Medical Domain .................... 47  
6.3.2 Workflow Automation .......................................................... 47  
6.3.3 Rule-Based System for Monitoring .......................................... 48  

7 The Jess Rule-Based System .......................................................... 49  
7.1 Running Jess ............................................................................. 49  
7.2 The Jess Rule Language ............................................................. 50
# Table of Contents

## 7 Working with Java Classes and Objects

- **7.3 Working with Java Classes and Objects** ........................................ 51
  - **7.3.1 Java Objects** ................................................................. 51
  - **7.3.2 Java Class Members** .................................................... 52

## 8 Working Memory

- **7.4 Working Memory** ........................................................................ 52
  - **7.4.1 Unordered Facts** ............................................................. 52
  - **7.4.2 Shadow Facts** .................................................................. 52
  - **7.4.3 Ordered Facts** .................................................................. 53

## 9 Rules

- **7.5 Rules** ...................................................................................... 53
  - **7.5.1 Slot Constraints** .............................................................. 54
  - **7.5.2 Pattern bindings** .............................................................. 55
  - **7.5.3 Conditional Elements** ....................................................... 55

### III Implementation

- **8 Jess and JADE Integration: The Matrix Example** .......................... 59
  - **8.1 Introduction** ........................................................................ 59
    - **8.1.1 Agent Organization** ...................................................... 61
    - **8.1.2 Agents Environment** ................................................... 61
  - **8.2 The Matrix Workflow** .......................................................... 63
    - **8.2.1 State of a Matrix Agent in Jess** ...................................... 63
    - **8.2.2 Implementation of The Matrix Workflow With Jess** ............ 64
  - **8.3 Agents Communication** ......................................................... 69
    - **8.3.1 Agent Interaction** .......................................................... 70
    - **8.3.2 The Matrix Ontology** ..................................................... 73
    - **8.3.3 Using the Communication Configuration Factory** ............... 75
  - **8.4 Agent Behaviors** ................................................................... 76
    - **8.4.1 Internal Behaviors** ....................................................... 77
    - **8.4.2 Communicational Behaviors** .......................................... 77
  - **8.5 Software Agents** ................................................................... 80
    - **8.5.1 TheMatrixAgent** ............................................................ 80
    - **8.5.2 Slave Agents** ............................................................... 82
    - **8.5.3 Master Agent** ............................................................... 82
    - **8.5.4 Jess Administration Agent** ............................................ 83

### 9 Remote Jess Administration

- **9 Remote Jess Administration** ......................................................... 84
  - **9.1 Introduction** ........................................................................ 84
  - **9.2 Using the Jess Administration Agent** ...................................... 84
  - **9.3 Agents Communication** ......................................................... 85
    - **9.3.1 Jess Administration Ontology** .......................................... 85
  - **9.4 Agent Behaviors and related Classes** ...................................... 86
# Table of Contents

9.4.1 Communicational Behaviors ........................................ 87
9.4.2 Implementation of the Agent Conversations ...................... 87
9.5 Jess Administration Agents ........................................... 92

10 Jess and JADE Integration: The Jess 2 JADE Tool Kit ............... 94
  10.1 Introduction .......................................................... 95
  10.2 Agent Communication Configuration .............................. 95
  10.3 Agent Behaviors ....................................................... 97
    10.3.1 Communicational Behaviors .................................. 97
    10.3.2 Message Templates ............................................ 100
  10.4 Software Agents ..................................................... 102
    10.4.1 JessableAgent .................................................. 102
  10.5 Jess and JADE Integration Object ................................. 103
    10.5.1 Jess Integration ............................................... 104
    10.5.2 Jess Engine Manager ......................................... 105
    10.5.3 Jess Output Routing .......................................... 106
  10.6 Ontology Management ............................................... 108
    10.6.1 Ontology Tree Manager ...................................... 110
    10.6.2 Ontology Visitors ............................................ 114
    10.6.3 Ontology Dependant Managers ............................... 116
  10.7 Jess Functions for JADE and Function Management ............... 117
    10.7.1 Jess Integrated Behaviors .................................. 121
  10.8 A Brief Discussion of an Oracle Agent Without Rule Engine .... 125

11 Using the J2J Tool Kit to Add Jess to MediMAS .................... 128
  11.1 Introduction .......................................................... 128
  11.2 Creating a Jess Administration Agent ............................ 129
  11.3 Modifying the MediMAS Ontology .................................. 130
    11.3.1 Create a Communication Configuration Factory for the MediMAS ontology ........................................... 131
  11.4 Adding Jess Support to MediMAS ................................... 131
    11.4.1 Extending JessCNC ............................................. 131
    11.4.2 Making the MediMAS Alert Manager Agent Jessable ....... 131
    11.4.3 Creating a Jessable Behavior to Handle the Alert Notification .................................................. 132
    11.4.4 Implementing the Alert Notification Logic with Jess Rules .......... 135
    11.4.5 Adding a JADE Aware Jess User Function ................... 136
  11.5 Changing the Behavior at Run Time ................................ 137
    11.5.1 Sending Alerts To the Head of Laboratory .................. 137
    11.5.2 Reacting to Laboratory Results with H5N1 .................. 137
  11.6 Conclusion .................................................................. 139
## Table of Contents

### 12 Conclusion
- 12.1 Achievements ................................................................. 140
- 12.2 Future Work ................................................................. 142

### IV Appendix

**A** Healthcare Business Processes at the HCF and LCF .................. 147

**B** Messaging Service UML Sequence Diagrams .......................... 152

**C** Installing and Running the Matrix ....................................... 157

**D** Algorithms
  - D.1 Creating the JTree of the Ontology .................................... 158
  - D.2 Sorting the Ontology Classes ........................................... 159
  - D.3 Creating the Ontology Composite ...................................... 160

**E** Common Acronyms .................................................................. 162

**F** License of the Documentation .............................................. 164

**G** Website of the Project ....................................................... 165

**H** CD-ROM .............................................................................. 167

References ................................................................................ 169

Referenced Web Resources .......................................................... 172
# List of Figures

2.1 Splits possible with Event-driven Process Chains (EPC) logical connectors [13, p. 33] ................. 8

3.1 The context in which actors communicate (obtained from [19]) ................. 15
3.2 Layers of the current communication infrastructure (obtained from [19]) . 17
3.3 Physician requests the laboratory analysis results ............................. 18
3.4 Physician tries to reach the Laboratory Assistant by phone ................. 19
3.5 Laboratory assistant tries to reach the physician by phone ................. 20
3.6 The laboratory assistant has critical laboratory results ....................... 21

4.1 An agent senses information from its environment and acts upon it. Its action can affect the environment [Woo02, p. 16] ................................. 24
4.2 Foundation for Intelligent Physical Agents (FIPA) Request Interaction Protocol (obtained from [8]) ................................. 29
4.3 Ontology-Based Communication Model [5] ................................. 30

5.1 The Graphical User Interface (GUI) of the Remote Monitoring Agent (RMA) 35
5.2 High-level overview of message delivery with JADE (obtained from [BCG07, p. 134], red demarcation added) ................................. 37
5.3 A simple JADE ontology with three additional classes ................................. 40
5.4 Screenshot of the Protégé ontology editor ................................. 41

6.1 The architecture of a typical rule-based system [Hil03, p. 20] ................................. 44
6.2 The Rete network generated for the rules detect-klingon and raise-shields ................................. 46

8.1 The results of a combat between one Chosen One agent and two Smith agents ................................. 60
8.2 The multi-agent environment ................................. 61
8.3 Phases in a slave agent’s life ................................. 63
8.4 Initialization Workflow ................................. 64
8.5 Main Workflow ................................. 66
<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.6</td>
<td>Slave agent registration process</td>
<td>70</td>
</tr>
<tr>
<td>8.7</td>
<td>Main workflow</td>
<td>72</td>
</tr>
<tr>
<td>8.8</td>
<td>Matrix Ontology</td>
<td>73</td>
</tr>
<tr>
<td>8.9</td>
<td>Behaviors to find the master agent and conduct a set of setup tasks</td>
<td>77</td>
</tr>
<tr>
<td>8.10</td>
<td>CommandMeResponder jessable behavior</td>
<td>80</td>
</tr>
<tr>
<td>8.11</td>
<td>Class hierarchy for agents in the Matrix environment</td>
<td>81</td>
</tr>
<tr>
<td>9.1</td>
<td>Remote Jess administration interface</td>
<td>85</td>
</tr>
<tr>
<td>9.2</td>
<td>Jess Administration Ontology</td>
<td>86</td>
</tr>
<tr>
<td>9.3</td>
<td>&quot;Register Me&quot; and &quot;Update Buffer&quot; based on the FIPA-Request-Protocol</td>
<td>89</td>
</tr>
<tr>
<td>9.4</td>
<td>Steps performed to create a JTree representation of the ontology used by Jess</td>
<td>91</td>
</tr>
<tr>
<td>9.5</td>
<td>Class hierarchy for agents in the Jess Administration environment</td>
<td>92</td>
</tr>
<tr>
<td>10.1</td>
<td>Abstract Factory to create agent communication related objects</td>
<td>95</td>
</tr>
<tr>
<td>10.2</td>
<td>Classes involved in the message template creation process</td>
<td>101</td>
</tr>
<tr>
<td>10.3</td>
<td>Object composition for the message template created by the createProtocol-InitialisationWithAction method</td>
<td>102</td>
</tr>
<tr>
<td>10.4</td>
<td>High-level overview of the JADE and Jess integration</td>
<td>103</td>
</tr>
<tr>
<td>10.5</td>
<td>JessCNC and managers</td>
<td>104</td>
</tr>
<tr>
<td>10.6</td>
<td>JessEngineManager class</td>
<td>106</td>
</tr>
<tr>
<td>10.7</td>
<td>AgentOutputWriter and Strategies</td>
<td>107</td>
</tr>
<tr>
<td>10.8</td>
<td>Jess output routing with the AgentOutputWriter</td>
<td>108</td>
</tr>
<tr>
<td>10.9</td>
<td>Composite for the ontology managed by the Ontology Tree Manager</td>
<td>109</td>
</tr>
<tr>
<td>10.10</td>
<td>Ontology Management Classes</td>
<td>110</td>
</tr>
<tr>
<td>10.11</td>
<td>A new Observer registering itself with the concrete Subject</td>
<td>111</td>
</tr>
<tr>
<td>10.12</td>
<td>Simple class hierarchy for the sorting example</td>
<td>113</td>
</tr>
<tr>
<td>10.13</td>
<td>Steps performed to construct a composite structure of the ontology of Figure 10.12</td>
<td>114</td>
</tr>
<tr>
<td>10.14</td>
<td>Ontology Composite Visitor Class Diagram</td>
<td>114</td>
</tr>
<tr>
<td>10.15</td>
<td>A Visitor visiting a ConceptElement subtree</td>
<td>115</td>
</tr>
<tr>
<td>10.16</td>
<td>JADE aware Jess function and Jess Function Manager</td>
<td>118</td>
</tr>
<tr>
<td>10.17</td>
<td>Jessable behavior class hierarchy used for the Matrix</td>
<td>121</td>
</tr>
<tr>
<td>10.18</td>
<td>Beginning and end of prepareResponse</td>
<td>124</td>
</tr>
<tr>
<td>10.19</td>
<td>Object interaction for command-agent</td>
<td>125</td>
</tr>
<tr>
<td>10.20</td>
<td>Object interaction for return-fact-to-current-behaviour</td>
<td>125</td>
</tr>
<tr>
<td>11.1</td>
<td>The JessMedicalAdmin class</td>
<td>129</td>
</tr>
<tr>
<td>11.2</td>
<td>Extract of the MediMAS ontology</td>
<td>130</td>
</tr>
<tr>
<td>11.3</td>
<td>The class hierarchy of the JessableAlertManagerAgent class</td>
<td>132</td>
</tr>
<tr>
<td>11.4</td>
<td>The class hierarchy of the AgentHandleAlertActionsBehaviour class</td>
<td>133</td>
</tr>
<tr>
<td>11.5</td>
<td>The class hierarchy of the ChangeAlertCheckTime class</td>
<td>136</td>
</tr>
</tbody>
</table>
12.1 Eliminating the single point of failure by providing redundancy with backup agents .................................................. 144

B.1 Step 1 ........................................................................ 153
B.2 Step 2 ........................................................................ 154
B.3 Step 3 ........................................................................ 155
B.4 Step 4 ........................................................................ 156

G.1 Screenshot of the project’s official web-page ....................... 166

H.1 Tree view of the content of the CD-ROM ............................. 168
H.2 The CD-ROM of this project ............................................. 168
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Elements of the EPC notation [SW01, p. 11–14]</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Elements of the Unified Modeling Language (UML) Activity Diagram notation</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>Mapping EPC elements to UML 2.0 Activity Diagram elements [KL06, p. 5]</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Additional EPC process elements to UML 2.0 Activity Diagram elements [KL06, p. 6]</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Mapping EPC control flow to UML 2.0 Activity Diagram elements [KL06, p. 6]</td>
<td>13</td>
</tr>
<tr>
<td>2.6</td>
<td>Mapping EPC splits and joins UML 2.0 Activity Diagram elements [KL06, p. 7]</td>
<td>14</td>
</tr>
<tr>
<td>10.1</td>
<td>Steps performed by the <code>buildSimpleTree</code> of Code extract D.2 for the class hierarchy in Figure 10.12</td>
<td>113</td>
</tr>
</tbody>
</table>
**List of Code extracts**

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Composing and sending a message</td>
<td>36</td>
</tr>
<tr>
<td>5.2</td>
<td>Receiving a message</td>
<td>36</td>
</tr>
<tr>
<td>5.3</td>
<td>ExampleOntology class source code</td>
<td>39</td>
</tr>
<tr>
<td>5.4</td>
<td>Sell class source code</td>
<td>40</td>
</tr>
<tr>
<td>6.1</td>
<td>Defining a template and asserting a fact</td>
<td>45</td>
</tr>
<tr>
<td>6.2</td>
<td>Writing rules</td>
<td>46</td>
</tr>
<tr>
<td>7.1</td>
<td>Using the Jess Java API</td>
<td>49</td>
</tr>
<tr>
<td>8.1</td>
<td>Train new agent</td>
<td>64</td>
</tr>
<tr>
<td>8.2</td>
<td>Assign opponents</td>
<td>65</td>
</tr>
<tr>
<td>8.3</td>
<td>Check agent’s health</td>
<td>65</td>
</tr>
<tr>
<td>8.4</td>
<td>Remove opponent relations for killed agent</td>
<td>65</td>
</tr>
<tr>
<td>8.5</td>
<td>Add new key to the working memory</td>
<td>66</td>
</tr>
<tr>
<td>8.6</td>
<td>Wait</td>
<td>67</td>
</tr>
<tr>
<td>8.7</td>
<td>Die</td>
<td>67</td>
</tr>
<tr>
<td>8.8</td>
<td>Assert an attack fact and select new attacker</td>
<td>68</td>
</tr>
<tr>
<td>8.9</td>
<td>Attack!</td>
<td>69</td>
</tr>
<tr>
<td>8.10</td>
<td>Reduce opponents life</td>
<td>69</td>
</tr>
<tr>
<td>8.11</td>
<td>Opponent killed</td>
<td>69</td>
</tr>
<tr>
<td>8.12</td>
<td>Opponent still alive</td>
<td>69</td>
</tr>
<tr>
<td>8.13</td>
<td>Content of a FindMe message</td>
<td>71</td>
</tr>
<tr>
<td>8.14</td>
<td>Content of a CommandMe message</td>
<td>73</td>
</tr>
<tr>
<td>8.15</td>
<td>Extending the CommunicationConfigurationFactory to set the ontology, content language and codec</td>
<td>75</td>
</tr>
<tr>
<td>8.16</td>
<td>Creating a request performative message for the FIPA-Request-Protocol</td>
<td>76</td>
</tr>
<tr>
<td>8.17</td>
<td>Creating an Initiator behavior by extending RequestInitiatorTemplate</td>
<td>79</td>
</tr>
<tr>
<td>8.18</td>
<td>Creating a Responder behavior by extending RequestResponderTemplate</td>
<td>79</td>
</tr>
<tr>
<td>8.19</td>
<td>Implementation of the CommandMe responder class</td>
<td>81</td>
</tr>
<tr>
<td>8.20</td>
<td>Setup method of the TheMatrixAgent class</td>
<td>82</td>
</tr>
<tr>
<td>8.21</td>
<td>Setup of a slave agent</td>
<td>82</td>
</tr>
</tbody>
</table>
8.22 Setup the Oracle agent

9.1 Adding a JadeMessageOutputStrategy

9.2 Updating a Jess Administration agent’s Jess output console

9.4 Implementation of the ExecuteJessCommandInitiator behavior

9.3 Implementation of the ExecuteJessCommandResponder behavior

9.5 Parenthesized representation of the ontology

9.6 A simple tree that is parsed according to Figures 9.4(a) to 9.4(h)

10.1 createFIPARequestMessage instance method

10.2 createAgentConfigObject instance method of the abstract class Communication-ConfigurationFactory

10.3 RequestInitiatorTemplate

10.4 RequestResponderTemplate

10.5 createProtocolInitialisationWithAction instance method

10.6 Initializing JessCNC

10.7 Jess template for FindMe

10.8 treeChanged instance method of the Jade2JessVisitor class

10.9 Registering JADE aware Jess functions

10.10 Algorithm for the command-agent Jess user function

10.11 Algorithm for the return-fact-to-current-behaviour Jess user function

10.12 FIPA-Request abstract Responder behavior with Jess support

10.13 ChangeMeResponder without Jess rule-engine

11.1 The handleAlert method of the AgentHandleAlertActionsBehaviour class

11.2 The instructions to set the address of the master and load the behaviors needed to communicate with the master agent

11.3 Extending JessCNC to add a new JADE aware Jess user function

11.4 Making the AlertManagerAgent Jessable

11.5 Configuring the OntologyTreeManager class for the MediMAS ontology

11.6 Making the AlertManagerAgent Jessable

11.7 The call method of the ChangeAlertCheckTime class

11.8 Handle urgent and critical cases

11.9 Notify the Swiss Ministry of Health if a H5N1 case appears

11.10 A test fact for the h5n1-found rule of Code extract 11.9

11.11 The SendEmail class

D.1 Algorithm to create a JTree that contains the ontology

D.2 Sorting the classes of the ontology

D.3 Algorithm that creates the composite representing the ontology
E-health has become a driving factor in the healthcare sector. It enables the automation of clinical processes and increases efficiency. This is critical in order to fulfill the medical requirements of our society. Wireless mobile technology in particular can be rolled out relatively cheaply and quickly. E-health is not simply about digitalizing existing processes but to discover new ways to increase efficiency and effectiveness of the healthcare system. Thus it aims at driving down cost, reducing complexity and improving service for patients [3]. E-health processes seldom reside within a single organizational unit. Rather, they span several organizations, are partly automated and include different actors such as patients, nurses, laboratory personnel or physicians. The ICT infrastructure that supports these processes is likely to be heterogeneous. Furthermore, processes in the healthcare domain are dynamic in order to meet the changing needs of the patients [ASC*03, p. 83–84]. A number of solutions have been proposed to support processes in different areas of the healthcare sector. [HS06] present a rule-based approach to provide more flexible information system infrastructure and thus make business processes more agile. They explain that business requirements for health assurances change rapidly. They argue that an approach to increase the flexibility of business process is the automation of certain tasks based on business rules. By specifying the processes in a business-rule layer that accesses the ICT infrastructure required for the given process through a web-service layer, changes in the business environment can be met more efficiently. Another interesting approach is presented by [PJDH03]. They have developed a framework to support patient scheduling processes in a hospital environment. The authors explain that these processes are very dynamic and require an solution that can quickly adapt to the patients’ need. They propose a competitive, multi-agent environment to support the entire treatment process of a patient. Every patient and relevant resource in the hospital is represented by an agent. The patents’ agents compete for the scarce resources and negotiate to find an overall optimal solution.

A third example that is the initiator of this thesis, is MediMAS. MediMAS stands for Medical Multi-Agent Systems and is being developed by the Software Engineering Research Group at the University of Fribourg, Switzerland. The project was initiated to
improve laboratory results exchange processes between the Laboratoire Cantonal de Fribourg and the Hôpital Cantonal de Fribourg by adding a multi-agent system layer on top of the existing legacy ICT infrastructure. Each actor taking part in the process (i.e. physician, laboratory assistant) is represented by a software agent. MediMAS is being further developed through students projects at bachelor and master level [18]. This master thesis developed a framework for MediMAS to integrate JADE and Jess to make MediMAS more flexible and adaptive to changing process requirements.

1.1 Motivation and Goals

While a multi-agent environment developed with JADE provides a scalable, fail-safe distributed environment, it does not provide the means to change an agent behavior at run-time. Rather, if an agent’s business logic needs to be changed, the agent has to be stopped, the new classes loaded and the agent restarted. This limits its suitability to support flexible processes that require high availability, such as the processes supported by MediMAS. The two following examples shall clarify this:

1. The head of the laboratory has decided that she wants to be notified of every result that is classified as critical and urgent.

2. A new strain of H5N1 was discovered in Switzerland. The Swiss Ministry of Health requires immediate notification if this strain is found in a human.

These seemingly simple requirements would make it necessary to take down MediMAS for the update. While the first requirement is probably not that urgent and could be performed during a night shift, the second requirement would likely require to take the system down while it is being actively used. A solution to this problem is to separate the business logic execution from the other parts of the system, i.e. underlying multi-agent framework, and have the source code that represents the business logic interpreted and executed directly. Additionally, it would also be advantageous if the activities could be written in declarative way, instead of describing the process with procedural instructions, and let the rule-based system decide on the order in which the activities are executed. Such an activity for for the H5N1 example could be: "If there is a laboratory result with H5N1 then inform the Swiss Ministry of Health". If a laboratory result is submitted to the system that contains information on a H5N1 case, this activity is performed. if-then statements are the typical characteristic of rule-based systems. As mentioned in the introduction, a rule-based approach is indeed suitable to control processes. The rule-based system shell chosen for this tool kit is Jess. Jess is fully integrated in Java and can thus interact seamlessly with JADE. The Jess execution engine is an interpreter, the source code can be changed during run-time.

This leads to the goals of this thesis:

- Refinement of the study and of the description of the application domain and its different workflows. This part has to be cleanly and extensively described in a standard way, under the supervision of Marco Savini who is a member of Prof. Andreas Meier’s group.

- Extension of the MediMAS application by integrating JADE agents and the Jess rule engine: i.e. develop a richer version of the existing application, MediMAS 2.0.
This would allow users to declaratively and dynamically configure the behaviors of the agents.

These objectives were accomplished. The tool kit developed is generic and can be used for any JADE environment, not only MediMAS. In fact, it was developed independently of MediMAS with the prototype called "the Matrix".

1.2 Schedule

October - December 2007 (approx. 100h of work):

- Reading of the bachelor thesis report of [Rup07].
- Study and lecture of following topics: agents, ontology, rule engines, ...
- First contact with JADE, installation and configuration of the working environment, first tests.
- Installation of the MediMAS application and playing with it.
- Study of the application domain of the laboratory/hospital, reformulating/documenting the various workflows (cf. Marco Savini).
- Documentation of the first theoretical part of the master thesis (agents, ontology, business rules, rule engines, ...) and of the application domain description (laboratory/hospital).

January – February 2008 (approx. 290h of work):

- End of the documentation of the first theoretical part of the master thesis and of the application domain description.
- First integration tests JADE/Jess.
- Integration of Jess with one of the MediMAS agents.
- Thinking about the integration and description of the business rules in MediMAS, and integration with the "simulator".
- Roughly define the software architecture of MediMAS 2.0.

March – Avril 2008 (approx. 290h of work):

- Refactoring of the MediMAS application.
- Developing MediMAS 2.0: deployment of Jess integration, interaction with the "simulator".
- Tests, debugging, and technical documentation (javadoc, UML diagrams ...).
- Drafting the structure of the final report.
May - August 2008 (approx. 220h of work):

- Writing of the final documentation.
- Final presentation to the research group and demo of the application.

1.3 Organization

This thesis is structured as follows:

- Chapter 1 gives an introduction to this thesis and poses the motivation and goals that initiated this project.
- Part I focuses on business process modeling and discusses the case study that lead to the creation of MediMAS. It contains the following chapters:
  - Chapter 2 discusses and compares UML Activity Diagrams and Event-driven Process Chains used to model business processes and workflows.
  - Chapter 3 presents the case study that led to the creation of MediMAS.
- Part II elaborates on the theoretical foundations needed to develop the Jess and JADE integration.
  - Chapter 4 discusses software agents in general.
  - Chapter 5 introduces the multi-agent framework JADE, which is used for this thesis.
  - Chapter 6 provides an introduction to rule-based systems.
  - Chapter 7 elaborates on the rule-based system shell Jess. It is used in this thesis to handle the business logic on an agent.
- Part III discusses in details the integration approach chosen in this thesis. It is subdivided in:
  - Chapter 10 presents "the Matrix". It is a prototype developed to demonstrate how the Jess and JADE integration tool kit developed can be used.
  - Chapter 9 comments on the remote Jess Administration module of the Jess and JADE integration tool kit.
  - Chapter 10 delves into the details of the integration tool kit.
  - Chapter 11 explains the application of the Jess and JADE integration tool kit for MediMAS.
  - Chapter 12 closes this thesis with a conclusion.

1.4 Notations and Conventions

- Formatting conventions:
  - **Bold** and *italic* are used for emphasis and to signify the first use of a term.
  - **SansSerif** is used for web addresses.
  - **Code** is used in all Java code and generally for anything that would be typed literally when programming, including keywords, constants, method names, and variables, class names, and interface names.
• The present report is divided in Chapters. Chapters are broken down into Sections. Where necessary, sections are further broken down into Subsections, and Subsections may contain some Paragraphs.

• Figures, Tables and Code extracts are numbered inside a Chapter. For example, a reference to Figure $j$ of Chapter $i$ will be noted $Figure \ i.j$.

• As far as gender is concerned, I systematically select the feminine when possible.

• Source code is displayed as follows:

```java
CommunicationConfigFactory cf = JessCommunication.getInstance();
    cf.setAgent(this.myAgent);
RegisterMe rm = new RegisterMe();
```
Part I

Motivation
### 2.1 From Functional to Process-Oriented Organization

Changes in the economical situation have forced companies to restructure their organization from a functional to a process orientation. Before this paradigm shift, organizations focused on the execution and improvement of individual functions. Services and products however, are rarely created by a single functional unit. Thus business processes cut across several functional business units and will likely also include partners that are outside of an organization. The coordination between different functional units is often costly [All05, p. 12–14],[BKR03, p. 2]. The use of Information and Communications Technology (ICT) helped to increase process cycle time but they could not eliminate the structural issues caused by a company’s internal structure [BKR03, p. 2]. To reduce existing interfaces, organizations started to implement cross-functional processes by changing the organizational structure from a functional oriented organization to a process oriented one [All05, p. 12–14],[BKR03, p. 2].

Rummler and Brache define "business process" as follows: "A business process is a series of steps designed to produce a product or service. Some processes [...] may be contained wholly within a function. However, most processes [...] are cross-functional, spanning the "while space" between the boxes on the organization chart."[RB90, p. 45].

### 2.2 Business Process Modeling

Several business process modeling languages have been developed. Common examples are Petri-Nets, UML Activity Diagram, data flow diagrams and EPC [RvdAtHW06, p.
2.2. Business Process Modeling

It was opted to use two process modeling languages in this thesis: EPC and Activity Diagram.

EPC have found wide acceptance in the industry to model business processes in organizations [NR02, p. 64], [vdA99, p. 1], [Krc04, p. 122]. EPC are targeted at business people. They are easy to understand and to use. EPC serve to describe the business logic of processes and the requirements of a business process but are not necessarily used for precise, formal specifications [vdA99, p. 4]. Thus readers not familiar with ICT will likely be able to interpret EPC diagrams than the more IT-oriented Activity Diagram.

The Activity Diagram notation is a high-level tool to model the sequence of steps that an information system will execute to achieve a goal. The Activity Diagram modeling language is part of UML, which has established itself as the de-facto standard for software modeling and design [RvdAtHW06, p. 1], [Kec06, p. 18].

Both modeling languages are described in the following Sections.

2.2.1 Event-Driven Process Chains

![EPC logical connectors](image)

Figure 2.1: Splits possible with EPC logical connectors [13, p. 33]

The EPC notation has been developed by Scheer, Keller and Nüttgens and is part of the Architecture of Integrated Information System (ARIS) framework. The ARIS framework integrates different views of an organization. It is centered around business processes to provide a coherent and holistic model of an organization [SW01, p. 6]. The EPC notation contains the elements shown in Table 2.1:

<table>
<thead>
<tr>
<th>ELEMENT</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Function" /></td>
<td><em>Functions</em> (i.e. activities or process steps) that take an input state and transform it into one or more output states. A function is depicted as a rounded rectangle.</td>
</tr>
<tr>
<td><img src="image" alt="Event" /></td>
<td><em>Events</em> describe the occurrence of a condition. They are pre- and/or post conditions of functions. An event is depicted as a hexagon.</td>
</tr>
</tbody>
</table>
2.2. Business Process Modeling

Logical connectors in EPC are used to model non-linear process paths. They connect functions and events. The EPC notation offers the following logical connectors:

**AND**: Lets a process execute tasks in parallel. It is depicted with \( \land \).

**OR**: Chooses between one or more alternatives. It is depicted with \( \lor \).

**XOR**: Chooses either one of two possibilities. It is depicted with \( \otimes \) (or simply XOR).

EPC supports the splits and joins shown in Figure 2.1. As shown in Figure 2.1, the connectors OR and XOR cannot be preceded directly by an event. This is due to the fact that events do not have decisional competencies. Hence, it cannot be decided which path to follow. The different process paths after a split can, but not have to, be rejoined. If they are, the join logical connector has to be of the same type as the split logical connector [13, p. 35].

Control flow elements connect elements of the types stated above to create a chronological-logical sequence of functions and events. A control flow element is depicted as a dashed arrow.

<table>
<thead>
<tr>
<th>Organization Unit</th>
<th>An <em>Organizational unit</em> element describes an organization or person in an organization.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information Object</td>
<td>A <em>Information object</em> element describes an object of the real world.</td>
</tr>
<tr>
<td>Information Flow</td>
<td>A <em>Information flow</em> element describes the interaction type with data (i.e. read, write, update).</td>
</tr>
<tr>
<td>Association</td>
<td>An <em>Association</em> element associates resources to organizational units.</td>
</tr>
<tr>
<td>Process Path</td>
<td>A <em>Process path</em> element sets a pointer to another process.</td>
</tr>
</tbody>
</table>

Table 2.1: Elements of the EPC notation [SW01, p. 11–14]

**Criticism**

The EPC specifications have been criticized for their lack of proper semantics and syntax, hence resulting in possibly ambiguous process descriptions. This is unfortunate, since business processes are at least partly embedded in information systems that demand the use of formalism to run the modeled business processes [NR02],[vdA99]. Different approaches to resolve this issue have been proposed. [vdA99] suggests to map EPC to Petri-Nets (due to the nature of Petri-Nets, the OR \( \lor \) connector has to be omitted). Petri-Net notation is based on well defined mathematical foundations and is provided with a large set of analysis tools.

[NR02] argue that mapping EPC to Petri-Nets only works when accepting certain restric-
2.2. Business Process Modeling

The issues surrounding the formalism of EPC will not be discussed any further. It would go beyond the scope of this thesis. Additionally, the technical part of this thesis will use Activity Diagram. Activity Diagram have precise semantics [Kec06, p. 16].

2.2.2 Activity Diagrams

UML is being continuously developed. The last major update was from UML version 1.5 to version 2.0 in 2005. Several diagrams were updated. For example the concepts on which Activity Diagram are based were completely revised [Kec06, p. 17–19].

Activity Diagrams are used to model processes at different stages in a software project. In the analysis and definition phase, they serve as a tool to model and document business processes. During the design phase, they are used to model internal system processes, which is their main area of use. These processes then are implemented during the implementation phase [Kec06, p. 216].

An Activity Diagram can contain three types of ActivityNodes:

**ExecutableNode:** An ExecutableNode is a node that can conduct a task, for example an Action.

**ObjectNode:** An ObjectNode can store data in an Activity Diagram, for example a Datastore.

**ControlNode:** A ControlNode coordinates the control flow between nodes in an Activity. Examples are start and stop nodes or logical connectors.

**ActivityEdges** are directed edges that connect ActivityNodes in sequential order. An Activity starts with an InitialNode and terminates with a FinalNode. An Activity may contain several InitialNodes and FinalNodes [Kec06, p. 250–251]. The Activity Diagram modeling language is very expressive. A comprehensive discussion of every element would go beyond the scope of this thesis. The rest of this Section describes the elements that are of relevance for this thesis.

<table>
<thead>
<tr>
<th>ELEMENT</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Action Icon" /></td>
<td>An <em>Action</em> is an ExecutableNode. It is a function or task that represents the smallest unit of execution in the given model and will not be subdivided further. Most of the remaining elements in an Activity serve for creating constructs such as sequences of Actions or group or structure actions. Actions can be provided with LocalPreconditions/Postconditions that have to be true prior or after the execution [Kec06, p. 217].</td>
</tr>
<tr>
<td><img src="image" alt="ControlFlow Icon" /></td>
<td>A <em>ControlFlow</em> is a directed edge that connects ActivityNodes. It represents the order in which ActivityNodes are executed [Kec06, p. 218].</td>
</tr>
<tr>
<td><img src="image" alt="Organization Unit" /></td>
<td>An <em>ActivityPartition</em> (swim lanes) associates ActivityNodes with a specific resource (i.e. actors) [Kec06, p. 219–220].</td>
</tr>
</tbody>
</table>
An ObjectNode represents an object. Objects are created and consumed by Actions. The ObjectFlow transports an object from an Action to another. Objects are denoted as squares. Alternatively, the object can be depicted using the pin notation. The state of the object may be added in square brackets.

Signal sending and receiving is used for asynchronous communication. Signals are sent by a SendSignalAction node, which is a type of Action node that sends messages asynchronously. Signals are received by AcceptEventAction node, which is a type of Action node that waits for signals. A SendSignalAction node has exactly one AcceptEventAction associated to it. The third type of asynchronous signals supported by UML are TimeEvents: A signal is sent at given time intervals [Kec06, p. 235].

The starting point of control flow within an Activity is the InitialNode. An Activity may contain two types of termination nodes: An ActivityFinalNode terminates the Activity when it is reached, a FlowFinalNode only ends its control flow [Kec06, p. 250–251].

A DecisionNode splits one incoming control flow in several disjunct alternatives (i.e. XOR). Each outgoing control flow has a guard associated to it (in square bracket) that defines the condition that must be met for it to be selected [Kec06, p. 252–253].

A MergeNode merges several incoming control flows to a single control flow. If a MergeNode is directly followed by another DecisionNode, both notation elements can be depicted as a single element [Kec06, p. 253].

A ForkNode splits a control flow in several parallel control flows. Control flows can be synchronized again with the JoinNode [Kec06, p. 258].

Table 2.2: Elements of the UML Activity Diagram notation

Criticism

[RvdAtHW06] have analyzed the UML 2.0 Activity Diagram notation in order to evaluate their strengths and weaknesses as a language for business process modeling. Their eval-
evaluation is based on the Workflow Patterns framework (see [17] for more details). In their paper they conclude that the Activity Diagram modeling language offers comprehensive support for control flow and data perspective (creating and using data, interaction with data, etc. [RvdAtHW06, p. 4]). Its support for organizational or resource related aspects however is limited [RvdAtHW06, p. 8].

2.2.3 Comparing Event-driven Process Chains and Activity Diagrams

Business people tend view information systems from a different perspective than Information Technology (IT) people. They are more interested in high-level description of cross-functional and cross organizational processes. Their generated value, costs, changes to the organization, etc. IT people on the other hand tend to view business processes from a more formal and implementation focused perspective. As mentioned in Chapter 2.2, this may also have an influence on the choice of the business process modeling language. To create a bridge between the worlds, [KL06] developed a “UML 2 profile for EPCs based on a 1 - 1 mapping with UML 2 Activity Diagrams” [KL06, p. 1]. The following tables contain the mappings between EPC and Activity Diagram identified by [KL06]. Only those of interest for this thesis are given.

<table>
<thead>
<tr>
<th>EPC ELEMENT</th>
<th>EPC NOTATION</th>
<th>UML 2 BASE CLASS</th>
<th>UML PROFILE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elementary Function</td>
<td>Function</td>
<td>Action</td>
<td>&lt;&lt;e.Function&gt;&gt;</td>
</tr>
<tr>
<td>Event</td>
<td><img src="image" alt="Event Diagram" /></td>
<td>Control Flow</td>
<td><img src="image" alt="Control Flow Diagram" /></td>
</tr>
<tr>
<td>Start Event</td>
<td><img src="image" alt="Start Event Diagram" /></td>
<td>Initial Node</td>
<td><img src="image" alt="Initial Node Diagram" /></td>
</tr>
</tbody>
</table>
2.2. Business Process Modeling

Table 2.3: Mapping **EPC** elements to **UML 2.0** Activity Diagram elements [KL06, p. 5]

<table>
<thead>
<tr>
<th>EPC Element</th>
<th>EPC Notation</th>
<th>UML 2 Base Class</th>
<th>UML Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organization Unit</td>
<td><img src="image" alt="Organization Unit" /></td>
<td>Activity Partition</td>
<td>&lt;&lt;Organization Unit&gt;&gt;</td>
</tr>
</tbody>
</table>

Table 2.4: Additional **EPC** process elements to **UML 2.0** Activity Diagram elements [KL06, p. 6]

<table>
<thead>
<tr>
<th>EPC Element</th>
<th>EPC Notation</th>
<th>UML 2 Base Class</th>
<th>UML Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organization flow control</td>
<td><img src="image" alt="Organization flow control" /></td>
<td>Activity Partition</td>
<td>&lt;&lt;Organization Unit&gt;&gt;</td>
</tr>
</tbody>
</table>

Table 2.5: Mapping **EPC** control flow to **UML 2.0** Activity Diagram elements [KL06, p. 6]

<table>
<thead>
<tr>
<th>EPC Element</th>
<th>EPC Notation</th>
<th>UML 2 Base Class</th>
<th>UML Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td><img src="image" alt="AND" /></td>
<td>ForkNode</td>
<td>&lt;&lt;Function&gt;&gt;</td>
</tr>
</tbody>
</table>
### Table 2.6: Mapping EPC splits and joins UML 2.0 Activity Diagram elements [KL06, p. 7]

<table>
<thead>
<tr>
<th>Operator</th>
<th>Diagram</th>
<th>Corresponding Element</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AND</strong></td>
<td><img src="image" alt="AND Diagram" /></td>
<td>JoinNode</td>
</tr>
<tr>
<td><strong>OR</strong></td>
<td><img src="image" alt="OR Diagram" /></td>
<td>MergeNode</td>
</tr>
<tr>
<td><strong>OR</strong></td>
<td><img src="image" alt="OR Diagram" /></td>
<td>ForkNode with guards</td>
</tr>
<tr>
<td><strong>XOR</strong></td>
<td><img src="image" alt="XOR Diagram" /></td>
<td>DecisionNode with guards</td>
</tr>
<tr>
<td><strong>XOR</strong></td>
<td><img src="image" alt="XOR Diagram" /></td>
<td>MergeNode</td>
</tr>
</tbody>
</table>
The LCF (the French acronym for Laboratoire Cantonal de Fribourg) is commissioned by the HCF (the French acronym for Hôpital Cantonal de Fribourg) to run analyses on certain types of samples submitted by a physician from the cantonal hospital of Fribourg. The cantonal laboratory of Fribourg consists of several departments such as hematology, chemistry and microbiology. Once the analysis results are available, they are communicated to the physician who made the request [Rup07, p. 33]. Each department conducts
and communicates their analyses independently. The current organization of the cantonal laboratory of Fribourg is shown in Figure 3.1 [Rup07, p. 33]. [Rup07, p. 34] identified four stakeholders directly or indirectly involved in the process: (i) the physician, (ii) the laboratory assistant, (iii) the head of laboratory (iv) and the patient.

They use the following communication patterns:

- **physician → patient**: The physician informs the patient of his / her disease state.
- **laboratory assistant → physician**: The laboratory assistant communicates the laboratory results to the physician.
- **physician → laboratory assistant**: The physician makes a request to the laboratory assistant.
- **→ head of laboratory**: The head of laboratory is informed of any anomalies.
- **laboratory assistant → laboratory assistant**: Laboratory assistants communicate internally.

Obviously other actors are involved in running a hospital or a laboratory. They are, however, not of interest in the (to be) examined context and are, therefore, not considered. [Rup07, p. 34] notes that the system shown in Figure 3.1 can be viewed as a distributed system. The actors can communicate using (i) an information system and (ii) the phone, thus synchronization of the communication flow between different stakeholders cannot be guaranteed. The shared information of interest in this context is the analysis results from the laboratory. [Rup07, p. 34] explains that the system, in its current state, cannot guarantee information confidentiality, integrity and availability. This is due to the fact that stakeholders may choose to communicate information over the phone. Under normal circumstances, physician and laboratory assistants exchange laboratory results over the medical legacy application WinDMLAB [2]. In this case, transactions are logged and data integrity and security is guaranteed by the application. This, however, is not the case if actors decide to transmit information verbally over the phone. Reasons for doing so include that the physician cannot find the information she needs in the system or does, for some reason, not wish to use WinDMLAB. The phone is also used when the laboratory assistant realizes that the laboratory results (or parts of them) just obtained are critical and must immediately be brought to the physician’s attention. Simply submitting the information in WinDMLAB will not guarantee the delivery in time. Communication of laboratory analysis over the phone is error-prone: Besides the ideal case, in which the analysis results are communicated correctly the first time, the least worrying case is that a physician receives the laboratory results more than once. But, in the worst case, one cannot exclude that the information will never reach the physician. Because communication over the phone is not logged, it is impossible to determine, at a later point, when and where the information was lost or modified. In the medical domain this could have fatal consequences. Thus it is this communication channel that holds the most potential for improvement.

Figure 3.2 depicts the communication platform currently used to store and transmit the laboratory results between the different stakeholders. As indicated above, the communication system consists of two layers: the phone infrastructure layer and the information system layer. The information system layer is divided in two sub layers: the ICT infrastructure and WinDMLAB. The phone infrastructure is placed at the highest layer. The mindful reader might have noted that the patient, while an important stakeholder, does
not have any direct interaction with the communication platform and hence fully depends on the physician to inform her or him of the laboratory results. The rest of this Chapter will elaborate on four different use cases. The business processes in the remaining of this Chapter are viewed from the business perspective and independent of the implementation. Therefore, the EPC modeling language is used with two slight adoptions for the sake of clarity: swim lanes are used instead of organizational unit elements and each step is marked with a letter to which will be referred to in the text. The UML Activity Diagrams of these business processes can be found in Appendix A.

### 3.1 Regular Case

This workflow describes the procedure normally followed by physicians and laboratory assistants to exchange data. The workflow is depicted in Figure 3.3. The following steps are executed: A physician needs to lookup the patient’s laboratory results and medical record. 

1. She uses WinDMLAB to access the information she needs. If the information is available,
2. she studies the laboratory results and
3. treats the patient. If the laboratory results are not yet in the system
4. she has to decide whether or not (she wants) to try to reach the laboratory assistant by phone again. 
5. If she has tried too many times,
6. she tries to call the laboratory assistant (Section 3.2),
7. Otherwise, she waits for a given time period and tries again to access the data using WinDMLAB.

### 3.2 Physician Calling the Laboratory Assistant

This workflow is followed either when the physician unsuccessfully tried (several times) to access the laboratory results on WinDMLAB or when the physician simply prefers to use the phone. The reader notes that this process, shown in Figure 3.4, is already more
Figure 3.3: Physician requests the laboratory analysis results

complex than the normal case described in Section 3.1. Based on the availability of the stakeholders, the physician must decide on who to contact. This specific case might lead to a second specific case [Rup07, p. 35–36]. Additionally, the stakeholders do not communicate via WinDMLAB but over the phone. Hence the communication between the peers is not logged.

If this case occurs, the physician does the following: a) She tries to call the laboratory assistant. b) If the laboratory assistant is not available (1% probability [Rup07, p. 36]) and she has already tried to reach him for more than three times, she calls the head of laboratory. c) If the head of laboratory is available, the physician is given the results. If the head of laboratory is not available, d) she has no other option but to wait for a given time and a) start the process over again. There are no security checks to assure that the physician will get the laboratory results. If both the head of laboratory and the laboratory assistant are not reachable, this workflow has no guaranteed end (i.e. in theory might loop indefinitely). This could have potentially fatal consequences [Rup07, p. 36]. d) If the laboratory assistant is not available (1% probability) and the physician has tried less than three times, she waits a certain time period and a) tries again. If the laboratory assistant is available (99% probability [Rup07, p. 36]), e) she asks for
3.3 Laboratory Assistant Calling Physician

This workflow, depicted in Figure 3.5, applies to the situation where the physician could reach the laboratory assistant by phone but the laboratory did not, at that time, have the laboratory results ready and promised to call back once the analysis is complete. The reader will notice that the first part of the workflow uses the same pattern as the former process, discussed in Section 3.2. But with the roles inverted: It is the laboratory assistant who tries to contact the physician once the analysis is complete. In this case, the following happens:  

- **a)** The laboratory assistant tries to call the physician. If she is not available,  
- **d)** she waits for a given time period and tries again to reach the physician. If she tried to reach her for more than three times and still cannot reach her,  
- **b)** she will try to reach the head of laboratory and,  
- **c)** if she can be reached, communicates the results. If the head of laboratory is not available, the laboratory assistant has to try again at a later point in time.  
- **e)** If the physician is available, she verbally communicates the results to the physician. Again, the same problems occur as in special case one in Section 3.1: There is no guarantee that the stakeholders will ever be reached, nor are control mechanisms established to log what has been transmitted, when and by whom.
3.4. Laboratory Assistant has Critical Laboratory Results

Besides the three workflows already discussed, a situation can occur in which the laboratory results reveal an imminent danger to the patient and require the physician to act as soon as possible. In this case, the following workflow applies (shown in Figure 3.6): Like in the regular case (see Section 3.1), a) the laboratory assistant enters the laboratory results in WinDMLAB. b) If at least one part of the analysis result is critical, she tries to call the physician. f) If the physician is not available, she waits a given time and tries again later. c) If she has tried three times and she still cannot reach the physician, she contacts the head of laboratory and if she is reachable, d) communicates the results. It is then up to them to make sure the analysis results are transmitted to the right physician. If the head of laboratory cannot be reached and because the situation is critical, e) the laboratory assistant has no other choice but to personally bring the laboratory results to the medical service to which the patient has been assigned. g) If the physician can be reached, she communicates the results and h) the physician can treat the patient.

These four use cases have shown the processes between different stakeholders at the cantonal laboratory and the cantonal hospital of Fribourg. It was explained that a potential
worst case scenario might cost human lives [Rup07, p. 36]. To improve the current system, [Rup07, p. 36] suggested and developed a new solution that aims to reduce human errors and reduce the situation in which different stakeholders are not available.
Part II

Background
The software agent paradigm is a suitable approach to provide workflow automation and support. Entities in a business process are represented as autonomous agents that collaborate with their peer in their environment to achieve their given goal [Woo02, p. 245–247]. This Chapter gives an introduction to intelligent agents. Section 4.1 comments on the definition of an agent and the environment an agent can exist in. Section 4.2 presents different agent architectures followed by a discussion on agent communication in Section 4.3. Section 4.4 elaborates on ontologies, which play a central role in agent communication. Section 4.5 closes this Chapter with a brief introduction of three concrete multi-agent systems.
4.1 What is an Agent? What is an Intelligent Agent?

There is not a single definition of the term *agent*. It is generally agreed that an agent is a special software component that is autonomous, behaves like a human agent at the service of its clients and follows its own agenda [BCG07, p. 3]. This thesis is on multi-agent systems. The definition of the term “agent” is taken from Professor Wooldridge, an expert in the field of multi-agent systems. He defines the term “agent” as follows:

“An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives.” [Woo02, p. 15].

4.1.1 Properties of an Agent

An agent should have the following attributes [WJ95, p. 4–5]:

- **autonomy**: An autonomous agent can act without intervention from humans or other agents.
- **social ability**: An agent with social abilities will communicate with other agents or humans. Communication is via some agent-communication language.
- **reactivity**: An agent perceives its environment (digital and/or physical) and reacts to changes in the environment.
- **pro-activeness**: The agent may have goal-directed behaviors that are not simply caused by changes in the environment.

4.1.2 Properties of an Agent Environment

![Diagram](image)

Figure 4.1: An agent senses information from its environment and acts upon it. Its action can affect the environment [Woo02, p. 16]

An agent exists in an environment. As depicted in Figure 4.1, through its sensors it perceives information and data (referred to as “percept” [RN03, p. 32]) of its environment and generates actions through actuators (also referred to as effectors) that can affect the environment [Woo02, p. 16], [BWH07, p. 3]. The environment an agent exists in “is the aggregate of surrounding things, conditions, or influences with which the agent is
interacting.” [RH05, p. 19]. In the context of an agent software, those inputs include keystrokes, file contents, network packets to which the agent reacts by displaying, writing data to a file or transmitting data over the network [RN03, p. 32]. An agent can possess a number of actions (referred to as effectoric capability), with some actions being restricted to a limited set of situations. The same action executed by an agent twice, in apparently identical circumstances, might yield different effects. This is due to the fact that an agent is unlikely to have full control over its environment but will have at best partial control, meaning it can influence it. An agent must also be prepared to accept failures [Woo02, p. 16].

[RN03, p. 41–42] have identified the following dimensions to classify the properties of an agent environment:

**Fully observable vs. partially observable** If the agent knows the complete state of its environment through its sensors that are relevant to the choice of action at each point in time, the environment is fully observable. If the agent can only perceive parts of the environment, it is partially observable.

**Deterministic vs. strategic vs. stochastic** If the next state of the environment completely depends on its current state and the action executed by an agent, the environment is deterministic. If the environment is deterministic except for the actions of other agents, the environment is strategic. If none of the two mentioned states apply, it is stochastic.

**Episodic vs. sequential** The agent’s experience in an episodic environment is divided into atomic episodes. An agent performs in an action based on what it perceives and thus does not depend on the actions taken in previous episodes. In a sequential environment, however, an action taken in an episode could have effects in future episodes.

**Static vs. dynamic** A static environment does not change while the agent is deciding which action it should execute next; a dynamic environment, however, does change while the agent deliberates.

**Discrete vs. continuous** If only a limited set of distinct, clearly defined precepts and actions of the agent exist, the environment is called discrete. Otherwise, it is referred to as continuous.

**Single-agent vs. multi-agent** An environment is called multi-agent if two or more entities exist that aim to maximize their performance measure and depend on each others behavior. An environment with only one agent is a single-agent environment. In a competitive multi-agent environment, agents try to maximize their performance measures at the expenses of other agents (as an example [RN03, p. 32] state chess). In a cooperative multi-agent environment, agents are not competing at the expense of other agents.

### 4.2 Architectures

The central task of an agent is to decide which action it shall choose to best satisfy its design objectives. Several agent architectures have been proposed, ranging from purely reactive architectures to those that reason about their actions. In between lie hybrid
4.2. Architectures

Architectures. Agent architectures can be classified in four groups: (i) logic based, (ii) reactive, (iii) Beliefs, Desires and Intentions (BDI) and (iv) hybrid [BCG07, p. 4],[Woo02].

4.2.1 Logic Based Agents Architectures

Logic based architectures views decision making as deduction. An agent’s decision making strategy is encoded in logical theory [Woo02, p. 54]. An action is chosen using reasoning mechanisms. The logical based architecture finds its root in knowledge-based system techniques [BCG07, p. 4]. The logic based approach has the advantage that it has clean semantics. The disadvantages are, however, that, due to the computational complexity of theorem proofing, a logic based agent might not be able to operate effectively in an environment with time constraints. Additionally, it can be challenging to translate the real world into accurate, adequate symbolic description [Woo02, p. 54] [BCG07, p. 4].

4.2.2 Reactive Agent Architectures

Reactive agent architectures do not have any central symbolic model and use no symbolic reasoning. An agent’s decision making is accomplished through a set of task-accomplishing behaviors. Each behavior can be thought of as an action which continually perceives input and maps it to an action to perform. The reactive paradigm was strongly influenced by Rodney Brooks, the author of the subsumption architecture. He developed the subsumption architecture to concretize his ideas. The subsumption architecture can fire several behaviors simultaneously. To choose which behaviors will be fired, the subsumption architecture arranges behaviors in a hierarchy where the lower levels have higher priority than the higher levels [Woo02, p. 90–91]. The strength of the subsumption architecture is the computational simplicity, since the subsumption architecture does not plan; it perceives and acts, thus it performs better in a dynamic environment than a logic based agent. The disadvantage of the subsumption architecture is that the perceived information alone might be insufficient. Additionally, it is almost impossible to design agents that can learn from past experiences because they lack of an internal state. Furthermore, since the intelligence is to emerge from the interaction between different behavior, designing agents to perform specific tasks is difficult [BCG07, p. 5].

4.2.3 Hybrid Agent Architectures

Hybrid agents architectures include both reactive and proactive behaviors. The different behavior types are classified in various subsystems that are arranged in hierarchy of interacting layers. Two possibilities exist for the information and control flow between layers: horizontal layering and vertical layering. In a horizontal layered architecture, the software layers are directly connected to the sensory input and action output. Each layer acts like an agent and makes suggestions on how to act. In the vertical layered architecture, the sensory input flows from layer to layer and only one layer at a time is active [BCG07, p. 5–6].
4.2.4 BDI Architectures

BDI architectures have their origins in philosophy, in the theory of human practical reasoning. Intentions play a central role in the theory of human practical reasoning. The BDI paradigm treats computer programs as if they had a “mental state”. In the context of BDI architectures, Beliefs, Desires and Intentions can be defined as follows [BWH07, p. 15–16]:

**Beliefs:** Beliefs make up the agent’s knowledge about its environment. They can be out of date or inaccurate.

**Desires:** Desires are *state of affairs* the agent might want to achieve. Desires do not mean that the agent acts up on it, only that they have the potential to influence its actions. Desires can be mutually incompatible with one another.

**Intentions:** Intentions, in the context of BDI, are *states of mind*, in particular aimed towards some future state of affairs. Intentions can be either goals delegated to an agent or an option selected from a set of available options. This option then becomes an intention.

Practical reasoning is directed towards actions. Considering conflicting beliefs and desires, the agent chooses for or against competing options [BWH07, p. 17–18]. [BWH07, p. 18] explains that “Human practical reasoning seems to consist of two distinct activities: *deliberation* (fixing upon states of affairs that we want to achieve, i.e. our intentions); and *means-ends reasoning* (deciding how to act so as to bring about our intentions).”.

The process, during which an agent chooses his intentions, is known as *deliberation*. Intentions have the properties that the agent will attempt to achieve them, they persist until they are achieved or they have become obsolete. A chosen intention will constrain the space of possible intentions the agent has to consider and they are strongly related to believes about the future (the agent believes that under normal circumstances, achieving the intention is possible) [BWH07, p. 18–19].

The process to decide how the chosen intentions will be achieved, given the available means (actions that can be performed in the agent environment), is known as *means-ends reasoning*. In the artificial intelligence domain, means-ends reasoning is called *planning* [BWH07, p. 19]. A planner system takes the following inputs: (i) the agent’s intentions/goals, (ii) the agent’s beliefs about its environment (iii) and the actions available to the agent, to generate a plan to accomplish the selected goals/intentions [BWH07, p. 19].

4.3 Communication and Coordination

In a multi-agent environment, in addition to communicating with users and system resources, agents need to communicate with other agents in their environment to cooperate, collaborate and negotiate. Agents use an Agent Communication Language (ACL) to communicate with their peers. An ACL relies on speech act theory that provides separation between communicative acts and content language. Several ACL have been developed. Currently the most used is FIPA ACL [BCG07, p. 6]. One of the main features of FIPA ACL is the support of several content languages and managing conversation through predefined interaction protocols [BCG07, p. 6].
In an agent community, agents will coordinate their activities to assure that they are coherent. Because, according to \[BCG07\, \text{p. 6}\], “(1) agents’ goals may cause conflicts among agents’ actions, (2) agents’ goals may be interdependent, (3) agents may have different capabilities and different knowledge, and (4) agents’ goals may be more rapidly achieved if different agents work on each of them”.

The approaches to handle coordination among agents include \[BCG07\, \text{p. 7–8}]:

**Organizational structuring:** Agents are organized in a master/slave hierarchy. The master agent has to assure global coherence: it can gather information from other agents, make plans, coordinate and assign tasks in the group.

**Contracting:** In this approach, an agent can take the role of a contractor or of a manager. If an agent cannot solve a task with the local resources/expertise at its disposal, it takes the role of a manager and decompose the problem into subproblems and tries to find agents with the necessary resources/expertise that are willing to solve the subproblems.

**Multi-agent planning:** To avoid conflicting actions or interactions, using this approach agents build a multi-agent plan to achieve their goals. During execution, additional planning and replanning can occur. Multi-agent planning can be either centralized or distributed.

**Negotiation:** Negotiation is a communication process for agents to find a common accepted agreement on something. Negotiation can be either *competitive* or *cooperative*. Cooperative negotiation is used when agents share a common goal or have to achieve a common task. Hence the multi-agent system was created to pursue that goal. Competitive negotiation, on the other hand, is when agents are not in the first place cooperative but have their own, conflicting goals and are “not a priori cooperative, share information or willing to back down for the greater good” \[BCG07\, \text{p. 8}\].

### 4.3.1 FIPA Message Specifications

The IEEE standard organization FIPA promotes the interoperability of agent-based technologies \[9\]. Part of the FIPA standard specifies the structure of the asynchronous messages agents exchange to communicate. The list of parameters that can be set in an ACL message can be found in \[6\]. The parameters that are used depend on the context in which the message is sent. The only required parameter, in a FIPA-compliant message, is *performative* \[6\]. A performative is the action the receiver is expected to execute. Performative is also referred to as *communicative act* \[BCG07\, \text{p. 65}\]. In this thesis, both are used interchangeably. Commonly, an ACL message also contains the parameters *sender*, *receiver* and *content* \[6\].

### 4.3.2 FIPA-Protocols

The FIPA standard specifies a set of protocols to structure the conversations between agents. Protocols specify the roles of the participating agents and the messages exchanged \[7\]. The protocol on which every conversation is based in this thesis is the FIPA-Request-Interaction-Protocol$^1$.

---

1 FIPA-Request-Interaction-Protocol and FIPA-Request-Protocol are used interchangeably in this thesis.
FIPA-Request-Interaction-Protocol

The FIPA-Request-Interaction-Protocol defines the interaction between two agents. The message exchange is shown in Figure 4.2. The agent that starts the conversation is referred to as the *initiator*. The receiver as the *participant*. The initiator wants the participant to perform a certain action. It sends a message containing a Request performative and the action the agent is to perform. The participant may accept or refuse to perform the action. If the participant refuses, it replies with a *refuse*, otherwise, it replies with an accept. The accept can be omitted if the participant performs the action in a short time period [8]. Once the participant has completed the action, it replies with either [8]:

- **failure** The agent was not able to successfully complete the action.
- **inform-done** The action was completed successfully.
- **Inform-result** The action was completed successfully and the result from the action is returned to the initiator.
4.4 Ontologies in a Multi-Agent Environment

![Ontology-Based Communication Model](image)

In addition to being able to exchange messages, agents need to understand the messages they are exchanging. Thus they need to share “a vocabulary of predicates, functions, and constants” [RN03, p. 261] with an ontology (depicted in Figure 4.3). The term ontology has its root in Greek philosophy but is now extensively used in the domain of computer science [GPFLC04, p. 3]. In the context of this thesis, the latter is of interest. Several definitions of the term ontology can be found; one of the most cited is suggested by Gruber:

“An ontology is an explicit specification of a conceptualization” [Gru93, p. 1].

A number of ontology languages have been developed. The FIPA standard for agent-based technology adopted the frame-based knowledge model of Open Knowledge Base Connectivity (OKBC) which is, in the FIPA standard, referred to as FIPA-Meta-Ontology. The FIPA standard states that agents can use a knowledge representation language of their choosing. However to be FIPA compliant, they need to be able to provide a mapping to the FIPA-Meta-Ontology for knowledge communication [5].

A brief overview of the OKBC knowledge model is given below (for more information on OKBC, the reader is invited to consult the OKBC specifications in [15]):

**Frame:** A frame is a primitive object that represents an entity in the domain of discourse [CFF+98]. A class frame represents a class, an individual frame an individual object. A class (aka concept) is a set of entities. Entities are said to be instances of that class. An entity can be an instance of several classes. Classes that contain themselves classes, are known as meta-classes. Entities that are not classes are referred to as individuals. Classes can be organized in a hierarchy of subclasses.

**Slot:** Slots (aka attributes) define the characteristics of a frame. A slot has a set slot-values (entities) associated with it.

**Facet:** A facet is ternary relation between a slot, its value and the slot’s frame (individual or class). A slot frame can for example define the cardinality, type of a slot and default value [GPFLC04, p. 51].

The FIPA standard suggests two ways to manage ontology: (i) agents use explicit, declarative represented ontologies that are stored somewhere or, (ii) ontologies are implicitly represented in the agents’ code. It is also permitted for agents to use both approaches. For the former approach, the ontology service in an agent community is managed by a dedicated Ontology Agent (OA). The FIPA standard defines a set of functionalities that an OA can implement, such as publishing ontologies for other agents to discover, translation between different ontologies or ontologies maintenance (i.e. upload, download or
modify ontologies). An OA does not have to offer all functionalities. However, it must be able to notify clients if a requested functionality is not implemented. How ontologies are stored is not specified by the FIPA standard [5].

4.5 Applications

This Section briefly describes three implementations of multi-agent systems.

4.5.1 Multi-Agent System for Electronic Health Record Management

[TP07] propose a multi-agent system called "agent-based electronic health record system" (ABEHRS) to provide access to electronic health record information that are distributed among several medical institutions. They argue that electronic health records of patients are likely to be inconsistent as a result of being maintained in several location. With the growing need for healthcare providers to share patients' health records and a centralized approach to managing electronic health records, the authors argue that "the creation of a digital health information network is essential to our modern health care system" [TP07, p. 306]. Their framework is based on multi-agent swarm system. The individual agent by itself has only limited abilities. However, as a swarm they can perform complex tasks. The swarm approach was taken from societies in biology (i.e. bees, ants) [TP07, p. 308]. The ABEHRS framework permits distributed electronic health records of patients to self organize [TP07, p. 306].

4.5.2 Multi-Agent System for Process Automation

[JNF+00] explain that business activities need to be able to execute quicker at lower cost and with increased quality in an environment that includes several organizations and legacy information systems. The authors propose an agent-based approach. Each activity in a business process is represented by an agent. Agents can negotiate with their peers to "come to mutually acceptable agreements that coordinate their interdependent sub-activities" [JNF+00, p. 2]. This permits, the authors argue, a more flexible, adaptive and agile process management, compared to traditional workflow management systems [JNF+00, p. 2].

4.5.3 Multi-Agent System for Sensor Web Management

Remote sensing missions for Earth Science conducted by NASA uses a network of dynamic, seamlessly interconnected sensors. Critical events, such as hurricane detections, have to be handled quickly [SHC+07, p. 1]. NASA is developing a new sensor web based on a multi-agent architecture for communication and negotiation. The future sensor web for Earth Science will consists of interlinked platforms equipped with onboard information processing systems that can coordinate their activities with other platforms [SHC+07, p. 1]. NASA is developing the multi-agent framework "Multi-agent Architecture for Coordinated, Responsive Observations" (MACRO) to support the future sensor web. Agents
can describe their sensors in the Sensor Markup Language and perform collective observations [SHC+07, p. 2].
The Java Agent Development Framework (JADE) is an open source, fully FIPA compliant middleware, for multi-agent systems written in Java. Development began in late 1998 by the Telecom Italia. JADE was open sourced and distributed by Telecom Italia under the Library Gnu Public License (LGPL) in 2000. Other organizations, such as Motorola and France Telecom R&D, have become members of the board [BCG07, p. 29–30].

JADE is not domain or application specific. It offers the basic middleware functionalities for distributed applications, based on the multi-agents paradigm. A JADE environment has the interesting properties that [BCG07, p. 30–31]:

- Agents are autonomous and proactive: they have their own thread of execution and do not pass their object reference to other agents. They control their life cycle and autonomously decide when to perform their next action.

- Agents can refuse to execute an action and are loosely coupled: JADE agents use asynchronous, message-based communication. A sender addresses the receiver by using its unique name, not its object reference. Messages might even be sent without specifying a known sender, by sending it to a group of agents or to a proxy agent that will handle delivery to the appropriate agents. Additionally, a receiver can autonomously decide which messages it wants to process in the order of its choosing and which messages it wants to discard.

- The system is peer-to-peer: Agents have unique identifiers (the Agent Identifier (AID)) to directly address each other, they can join and leave the host platform at their choosing. Agents can look up other agents in the yellow pages (provided by the Agent Management System (AMS) and Directory Facilitator (DF) agents).
5.1 The JADE Architecture

A JADE agent platform consists of one or more container distributed over a network. JADE containers are Java processes that provide the JADE run-time and the services required for hosting and executing JADE agents. The main container is of special importance. It represents the bootstrap point of an agent platform and launches the other containers. Containers are identified by their logical name (by default the main container is called “Main Container”, the others “Container-1” etc.) [BCG07, p. 32]. The main container has the following additional functionalities [BCG07, p. 32–33]:

- Managing the Container Table (CT): The CT holds the object reference and transport address of each container that is part of its agent platform.
- Managing the Global Agent Descriptor Table (GADT): The GADT registers each agent in the platform with their location and status.
- Hosting the DF and AMS agents that provide the functionality of agent management, white pages and yellow pages services.

Each container has a Local Agent Descriptor Table (LADT) and (except for the main container) a local GADT cache to not have the GADT as a bottleneck. When a container needs to deliver a message, it looks up the agent’s address in its LADT. Only if the lookup is unsuccessful, it will attempt to find it in the GADT and store the result in its GADT cache. To prevent the main container from being the single point of failure, JADE offers replication mechanisms that enable the agent platform to function when the main container is unavailable [BCG07, p. 33].

5.1.1 Important Platform Management Services

According to the FIPA standard, the three most important services in a FIPA agent platform are the Message Transport Service (MTS), AMS and DF [BCG07, p. 39]:

Message Transport Service: The MTS manages the transportation of ACL messages between agents within an agent platform and agents in other agent platforms. The MTS implements all Message Transport Protocol (MTP) specifications defined by FIPA. By default, a HTTP-based MTP is activated. It is identified by an Uniform Resource Locator (URL) and accessible over a HTTP socket. Internally, JADE uses the more performant Internal Message Transport Protocol (IMTP) [BCG07, p. 39]. IMTP is a proprietary protocol used to transport messages between agents as well as internal commands for the management of the agent platform and status monitoring of the containers. By default IMTP is based on Java Remote Method Invocation (RMI). In a "Java Platform, Micro Edition environment", where RMI is not available, it uses a proprietary TCP socket protocol [BCG07, p. 41].

Agent Management System: The AMS agent manages the operations of an agent platform including registration and deletion of agents. Every agent in the agent platform must register with the AMS to obtain an AID. The AMS keeps a directory with all agents including their state. When an agent is deregistered from the AMS, it terminates. The agent’s AID is freed and can be reused by other agents that might request it. An AMS agent can request and force an agent to execute management functions (e.g. terminate execution). A platform has only one AMS agent [BCG07, p. 15–16].
5.1. The JADE Architecture

Directory Facilitator: A DF provides the yellow pages service for an agent platform. Agents that wish to publish their service, can request a registration of their agent description. A DF must maintain an accurate and complete list of its registered agents and answer queries with the most accurate information on a non-discriminatory basis. An agent platform can contain one or more DF agents that create a federation [BCG07, p. 15].

5.1.2 Administration and Debugging Tools

![Image of RMA GUI](Figure 5.1: The GUI of the RMA)

The JADE framework includes a set of agents to ease development and administration of a JADE agent platform:

Remote Monitoring Agent: The RMA is an administration tool that offers a GUI to access and visualize one or more agent platforms. An example is given in Figure 5.1. The left panel shows a agent tree. It contains three types of nodes: agent platforms, containers and agents. Each node’s life cycle in the tree can be manipulated (e.g. agents can be killed, frozen, migrated, saved) and new nodes added. A RMA subscribes with the AMS and received all platform-level events [BCG07, p. 43].

Dummy Agent: The dummy agent is a useful tool mainly used during development. Its only function is to send and receive custom made ACL messages to test “real” agents’ reactions. The dummy agent is used via a GUI [BCG07, p. 44].

Sniffer Agent: The sniffer agent sniffs and displays the communication flow and content in real time between a set of selected agents. It does this by subscribing itself to the AMS and is then notified of all events and messages related to the selected agents. Agents can be added or removed in real time to the group of sniffed agents. The messages sniffed can be viewed and saved [BCG07, p. 44–46].

Introspector Agent: The introspector agent can monitor and control in real time the life cycle of an agent (e.g. which behaviors are executing, sleeping, reacting to messages from other agents) [BCG07, p. 46].
5.2 Agent Communication Implementation

Providing an infrastructure for agents to exchange messages is a fundamental role of JADE. This Section discusses the implementation of the agent communication in JADE. Section 5.2.1 briefly describes how messages are sent and received from a programmer’s point of view. Section 5.2.2 delves into the implementational details of the objects involved and their roles, when a message is sent from an agent to another. Section 5.2.3 explains behaviors that implement agent actions of an agent in a JADE environment.

5.2.1 Sending and Receiving Messages

1. ACLMessage msg = new ACLMessage (ACLMessage.REQUEST) ;
2. msg.setLanguage("A-Language");
3. msg.setOntology("An-Ontology");
4. msg.setContent("Hello world!");
5. msg.setReceiver(new AID("Agent Doe", AID.ISLOCALNAME));
6. this.myAgent.send(msg);

Code extract 5.1: Composing and sending a message

1. ACLMessage msg = this.myAgent.receive (MessageTemplate.MatchPerformative (ACLMessage.REQUEST));
2. if (msg != null) {...}

Code extract 5.2: Receiving a message

Agents messages are asynchronous. Each agent has a mailbox to store incoming messages. By calling the agent’s receive method, the message next in the queue is fetched. receive can be provided with a message template to return only a message that matches a pattern defined by the template, for example a specific type of agent action and a given FIPA-protocol.

A simple example of how a message is sent from a behavior is shown in Code extract 5.1. When an ACLMessage is created, the performative is passed as constructor argument (line 1). In lines 2–5 the content language, ontology and message content are set. The values in this example are purely fictional. The message is sent by calling the agent’s send method in line 6.

Receiving a message is shown in Code extract 5.2. In this example, receive will only return a message if there is a message in the agent’s mailbox that matches the template specified as an argument (a message with a Request performative). In line 2, the behavior tests whether or not there is a message, otherwise, the behavior does not execute further. Both examples assume that they are executed from within a behavior.

5.2.2 Implementational Details of the JADE Messaging Service

JADE is developed with flexibility and extensibility in mind: its kernel uses a modular approach in which each service is a separate module. The architectural design is based on the concepts of aspect oriented programming that advocates a "clean separation of
concerns" [BCG07, p. 131]. The approach chosen for the implementation is the use of composition filter: each object is equipped with two filter chains: the filters of the incoming chain are executed before the method of the object is to be invoked and the filters of the outgoing chain are called before the object invokes the method of another object [BCG07, p. 131–132].

The core of JADE consists of a filter architecture that can be distributed over several containers. Each container resides on a node. A container hosts a set of agents and a node a set of services. Services may be composed of several components. To give a concrete example of such a service, the rest of this Section describes how a message is transpored in JADE. A high-level description of this process is shown in Figure 5.2. The red boxes labeled with Step 1 to 4 correspond to the UML Sequence Diagrams of Figure B.1, Figure B.2, Figure B.3 and Figure B.4 in Appendix B. When an agent request a service, such as sending a message, the request is passed to its container which forwards it to the service that is responsible to perform the operation. The service may implement the functionality directly or commit a vertical command. A vertical command is implemented in the class `jade.core.GenericCommand`. It embeds the functionality and accepts a set of parameters. The vertical command will access a component of the service to perform the task. This is referred to as the outgoing sink. Before the outgoing sink is reached, it passes through the outgoing filter chain and each filter may perform a service specific task with the command or simply pass the command to the next filter in the chain. Every service may offer a filter that will access its service. A service may overreach several platforms. The node to node interaction is referred to as horizontal command. A horizontal command is also implemented with the `jade.core.GenericCommand` class. The component within a service that accepts the horizontal from another node, is referred to as slice. A slice may either perform the command directly or execute another vertical command. Before the target service component is reached, the incoming filter chain is passed. The target service is refer to as incoming sink. Every service may provide...
5.2.3 Behaviors

An agent object does not contain any task related instructions. It contains a set of behaviors that implement its actions. Behaviors can be seen as threats. The agent contains a scheduler that uses a "round-robin non-preemptive scheduling policy" [BCTR00, p. 25] for the behaviors in the agent’s ready queue [BCTR00, p. 24–25]. In this thesis, the distinction between two types of behaviors is made:

- Behaviors that handle internal tasks of an agent and which then may, as result, also require communication with the outside world. In this thesis, they are referred to as internal behaviors.
- Behaviors that are directly involved in a request/response communication pattern between agents in the environment. In this thesis, they are referred to as communicational behaviors.

The former remains in the "internals" of an agent and thus does not expose an interface which, to maintain interoperability with other agents, is subject to standardization. The latter on the other hand, should adhere to common standards. Standardized communication interfaces are necessary to have loosely coupled agents that can interoperate with other agents that implement the same standard. This also permits to reuse agents and behaviors in a new environment without having to rewrite major parts of the code.

The JADE framework provides a set of predefined behavior classes for each actor taking part in the different FIPA protocol based conversations. The skeleton of the main communication flow during an agent conversation based on a FIPA-protocol, is managed by the Template Methods of these abstract classes. For steps that cannot be implemented in advance, the Template Method (the Template Method design pattern is discussed [GHJV95, p. 325]) calls hook operations which the subclasses can implement. Thus to implement behaviors that are in compliance with the FIPA standard, the behavior classes for the FIPA-protocol that best meet the communication, can be subclassed and the hook operations that are of interest are overwritten.

Naming Convention for FIPA-Request-Initiator-Protocol based Behaviors

Due to the nature of FIPA-Request-Interaction-Protocol based conversations (see Section 4.3.2), each conversation is centered around the agent action that the initiator requests the participant to perform. This permits to make the following conventions for behaviors developed in this thesis:

- For each initiator behavior, there is exactly one corresponding responder behavior.
- For each agent action used, there is exactly one corresponding "initiator, responder" behavior pair.

These simple conventions increase maintainability and extensibility of agents and their behaviors: It prevents from having to test an incoming agent action for its class in a behaviors code. This ultimately simplifies adding new actions and behaviors: To add the support for a new agent action, the initiator and responder each have to add a new behavior, other behaviors are not affected. It also simplifies debugging since the source
of an agent action related error can be narrowed down to two behaviors. Additionally, the naming convention, in this thesis, for behaviors that implement agent FIPA-Request-Interaction-Protocol is as follows: In each conversation, the initiator’s behavior’s name consists of the agent action and "Initiator" as suffix, and for the participant the behavior’s name consist of agent action and "Responder" suffix. Furthermore, for every conversation, the initiator creates a message containing a Request performative with the agent action and passes the message as a constructor argument to the Initiator behavior. As an example, based on a conversation discussed in Section 8.4.2, the behaviors for the "Attack" conversation are AttackInitiator and AttackResponder.

5.3 Use of Ontologies in JADE

JADE relies heavily on ontologies for agent communication. It offers a framework for developers to model domain specific ontologies. JADE does not use an Ontology Agent (OA) (see Section 4.4 for ontology management in multi-agent systems). Ontologies are written in Java and thus are directly represented as Java objects in the agents’ code [BM, p. 8]. The JADE upper ontology is defined in the content reference model. It contains:

- **Concept**, i.e. classes that are set of entities,
- **AgentAction**, a subset of Concept that is a set of all actions agents can perform and
- **Predicate** that state something about the status of the world and are either true or false [BCG07, p. 80].

An ontology is an instance of a class that extends the Java class jade.content.onto.Ontology. The concepts, agent actions and predicates of the domain specific ontology, are added as instances of PredicateSchema, AgentAgentSchema or ConceptSchema, respectively. For each concept, predicate and agent action of the ontology, the proper Java classes have to be developed. They must implement the interfaces Concept, AgentAction or Predicate [BCG07, p. 81–82]. In JADE, each ontology normally extends the basic ontology (not to confuse with extending Java classes) that is an instance of the jade.content.onto.BasicOntology. The basic ontology is represented as a Singleton (the Singleton design pattern is discussed in [GHJV95, p. 127]). It contains

- primitive types,
- aggregate types and
- domain independent predicates and concepts [BCG07, p. 82].

As an example: to declare that the JADE ontology \texttt{0}_{1} extends ontology \texttt{0}_{2}. Ontology \texttt{0}_{2} is passed as an argument to the super constructor of the ontology \texttt{0}_{1}. Ontology \texttt{0}_{1} includes all concepts, agent actions and predicates from \texttt{0}_{2} [BCG07, p. 82].

An example with a simple ontology, without inheritance, is shown in Figure 5.3. The the ontology contains three classes in addition to the default structure specified by JADE; a concept \texttt{Computer}, an agent action \texttt{Sell} and predicate \texttt{Owes}. The ontology itself is an instance of the class \texttt{ExampleOntology}. The source code for the \texttt{ExampleOntology} is shown in Code extract 5.3.

```java
public class ExampleOntology extends jade.content.onto.Ontology {
    //NAME
    public static final String ONTOLOGY_NAME = "Example";
    // The singleton instance of this ontology
```
5.3. Use of Ontologies in JADE

Figure 5.3: A simple JADE ontology with three additional classes

```java
private static ReflectiveIntrospector introspect = new ReflectiveIntrospector();
private static Ontology theInstance = new ExampleOntology();
public static Ontology getInstance() {
    return theInstance;
}
// VOCABULARY
public static final String OWNS_ITEM="item";
public static final String OWNS="Owns";
public static final String SELL_ITEM="item";
public static final String SELL="Sell";
public static final String COMPUTER="Computer";

private ExampleOntology()
    super(ONTOLOGY_NAME, BasicOntology.getInstance());
    try {
        // adding Concept(s)
        ConceptSchema computerSchema = new ConceptSchema(COMPUTER);
        add(computerSchema, ontology.jessmgmt.Computer.class);
        // adding AgentAction(s)
        AgentActionSchema sellSchema = new AgentActionSchema(SELL);
        add(sellSchema, ontology.jessmgmt.Sell.class);
        // adding Predicate(s)
        PredicateSchema ownsSchema = new PredicateSchema(OWNS);
        add(ownsSchema, ontology.jessmgmt.Owns.class);
        sellSchema.add(SELL_ITEM, computerSchema, ObjectSchema.MANDATORY);
        ownsSchema.add(OWNS_ITEM, computerSchema, ObjectSchema.MANDATORY);
    } catch (java.lang.Exception e) { e.printStackTrace(); }
}
```

Code extract 5.3: ExampleOntology class source code

The ontology classes are simple Java classes. Code extract 5.4 shows the Java implementation of the class Sell.

```java
public class Sell implements AgentAction {
    private Computer item;
    public void setItem(Computer value) {
        this.item=value;
    }
    public Computer getItem() {
        return this.item;
    }
}
```

Code extract 5.4: Sell class source code
5.3. Use of Ontologies in JADE

5.3.1 Protégé

JADE ontologies can be modeled with the open source ontology editor framework Protégé [25]. A screenshot of Protégé is shown in Figure 5.4. The Ontology Bean Generator Protégé plugin maps the ontology modeled with Protégé to Java classes that can be used by JADE [29].

![Screenshot of Protégé ontology editor](image-url)
Rule-based systems model intelligent behavior based on task-specific problem-solving knowledge of human experts [HR85, p. 921],[GFN89, p. 120],[Hil03, p. 18]. Rule-based systems are widely used in many application areas. Examples of rule-based systems frameworks are Jess [12], Prolog [14], CLIPS [23] and JRules [16]. [Hil03, p. 19] explains that many server applications integrate rule-based systems or have Application Programming Interface (API)s to do so. Thus, rule-based systems have become ubiquitous, even though they might not be mentioned explicitly. In the 1970s and 1980s, rule-based systems were used in the domain of artificial intelligence research [Hil03, p. 18] in the study of cognitive models, problem solving and learning systems [GFN89, p. 120]. Rule-based systems were also applied to problems in engineering, computer configuration tasks and oil exploration [GFN89, p. 120]. The success of rule-based systems to match or exceed the performance of their human counterparts in specific, limited situations led to the belief that one day, sophisticated rule-based systems would be able to reproduce general human intelligence. Over time, it became evident that the complexity of common sense, which underpins the general human reasoning, was vastly underestimated [Hil03, p. 18]. According to [HR85, p. 921], rule-based systems share the following properties:

1. “They incorporate practical human knowledge in conditional if-then rules,”
6.1. Rules

Unlike in procedural or object-oriented paradigm where the developer writes a sequence of instructions that the computer has to execute in the given order, rules are written in a declarative way: the programmer specifies what but not how it is done [Hil03, p. 15]. In rule-based systems, knowledge is represented as heuristics (i.e. “rules of thumb”) that specify which actions have to be performed in a given situation [22]. Thus, the operating concept radically differs from the von Neumann architectures. Rule-based programs must therefore, be executed in a run time system that can identify the heuristic rules applicable to the problems at hand and apply those rules to solve or reduce problems [HR85, p. 923]. Using this approach has several advantages: Rules are intended to be single chunks of know-how which “reflect the learned, appropriate, effective distinctions that people use to make sophisticated high-level decisions.” [HR85, p. 925]. This makes rules easier to understand compared to procedural programs [Hil03, p. 16]. Furthermore, due to its representation as a single “chunk” of information, the knowledge base is inherently modular and thus, simplifies adding, removing and updating of rules without affecting the overall performance of a rule-based system [vMSB84, p. 305]. Additionally, because the control-flow is chosen by the run time system, the program can be more flexible when the input is fragmented or with limited constraints [Hil03, p. 16].

Rules are comparable to if-then statements. As an example: a rule-based system on a Star Fleet ship might have the following rule:

```plaintext
IF
    Klingon ship is approaching
THEN
    Raise shields
END
```

The antecedent (if part) of a rule is often referred to as the left-hand side (abbreviated LHS), predicate or premise and consequent (then part) as right hand side (abbreviated RHS), action or conclusions. The set of information the rule can work with is called the domain [Hil03, p. 17]. Rule-based systems use the rules to derive conclusions from the premises. Two common methods of reasoning, when using inference rules, are forward and backward chaining [Hil03, p. 116],[BS84, p. 4–5].

**forward-chaining:** Forward-chaining is also called data-directed inference. The engine executes the RHSs of rules based on the data it knows.

**backward-chaining:** Backward chaining follows a goal-driven strategy. As with forward-chaining, rules are if-then statements but the behavior is different. The approach
6.2. The Architecture of Rule-Based Systems

The Architecture of Rule-Based Systems can be seen as “depth-first” [BS84, p. 49]. The system starts from a rule it wants to fire but with an *if* clause that only partly matches [Hil03, p. 116] and “works "backward" through inference rules, i.e., from right to left, to find the data that establish that goal” [BS84, p. 5].

6.2 The Architecture of Rule-Based Systems

The first rule-based systems were tightly integrated with the specific application they were developed for. Thus, new rule-based systems had to be written from scratch. The EMYCIN, a derivate from MYCIN (it stands for Empty MYCIN) was developed as a generic framework for rule-based systems, without the domain knowledge of the MYCIN. It was the first of its kind. This approach of developing a framework without a domain-specific knowledge is followed by most modern rule-based systems [Hil03, p. 19]. According to [Hil03, p. 19 – 20], the architecture of a rule-based system has the following components:

- A inference engine which consists of (i) a pattern matcher and (ii) an agenda
- an execution engine,
- a rule base,
- and a working memory.

[HR85, p. 924] labels the part that stores long term facts and rules as the *knowledge base*. The architecture of a rule-based system is depicted in Figure 6.1. The components are briefly described in the following sections.

6.2.1 Inference Engine

The inference engine is the central part of a rule-based system. It controls the process of applying rules to the facts in the working memory and solving the problems. [Hil03, p.
6.2. The Architecture of Rule-Based Systems

2014 explains that a rule-based system usually involves an iterative cycle that runs through the following steps:

1. The pattern matcher compares all rules to the working memory and decides which rules should be activated during the current cycle, resulting in an unordered list of active rules that, together with the rules activated in the previous iteration, make up the conflict set.

2. The next step is called conflict resolution. The conflict set is ordered to form the agenda: the set of rules of which RHSs will be fired (i.e. executed). The rule-based system does this based on its conflict resolution strategy. Developers have little control over this process.

3. In the last step, the first rule of the agenda is fired. Partial results are stored again in the working memory. Many rule-based systems use sophisticated methods to minimize redundant work. Also results from the pattern matcher and the agendas conflict resolution are preserved across iterations.

To better illustrate functionality of the parts of the rule-based system, the example with the incoming Klingon ship is continued. The rule-based system used for this example is Jess.

6.2.2 Working Memory

The working memory, also called the fact base, is the database of assertions. The working memory can contain both premises and conclusions derived from the rules. The information in the working memory is typically indexed for fast search operations [Hil03, p. 21–22].

In the example introduced above, a new fact representing an incoming Klingon ship, is asserted. The fact is of the class klingonShip (referred to as template in Jess). The template definition and fact assertion is shown in Code extract 6.1.

```lisp
1 (deftemplate klingonShip (slot type) (slot range))
3 (assert (klingonShip (type "hostile") (range 1000)))
```

Code extract 6.1: Defining a template and asserting a fact

The working memory contains now one fact:

f-0 (MAIN::klingonShip (type "hostile") (range 1000))

6.2.3 Rule Base

The rule base contains the rules available to the rule-based system. They can be stored in plain ASCII but are often compiled by the rule compiler to be used more efficiently by the rule inference engine [Hil03, p. 21]. The rule-based system used for this thesis, Jess, uses the Rete algorithm. The Rete algorithm preprocesses the rules and creates an interconnected network that consists of the tests made in the LHS of the rules. If several rules have the same test, the test is only represented once and shared among the rules that contain it, to increase efficiency of the pattern matching. When a fact is added to or removed from the working memory, the fact traverses the nodes in the network from
the top until it reaches a matching rule at the bottom (if there is any) [Hil03, p. 136–140]. This is far more efficient than checking the LHS of each rule against the working memory [Hil03, p. 135].

![Rete network diagram]

Figure 6.2: The Rete network generated for the rules detect-klingon and raise-shields

The rule-based system of the Star Fleet computer has two rules that are shown in Code extract 6.2

```
(defrule detect-klingon
  (klingonShip (type "hostile"))
  =>
  (printout t "Klingon Alert Captain!" crlf))

(defrule raise-shields
  (klingonShip (type "hostile") {range < 2000})
  =>
  (printout t "Raise shields!" crlf))
```

Code extract 6.2: Writing rules

The simplified representation of the Rete network, created by the rule compiler for the rules of Code extract 6.2, is shown in Figure 6.2.

### 6.2.4 Pattern Matcher

The pattern matcher decides which rules apply to the assertions in the working memory. This step is often the most expensive, in the sense of computational power, executed by the inference engine. If the working memory contains a large number of facts and each rule has several premises, the pattern matcher has to search through millions of combinations to determine which facts match the rules. The Rete algorithm reduces the execution time of this step [Hil03, p. 136–137].

Both rules apply to the newly assert fact. The (rule,facts) pairs are passed to the agenda.
6.2.5 Agenda

The set of activated rules that could fire, selected by the inference engine, is passed to the agenda, which uses a conflict resolution strategy to determine, depending on different criteria (e.g. age of the premises, complexity of a rule, priority), which rule should fire first.

The agenda decides to first fire `raise-shields` and then `detect-klingon`:

```plaintext
==> Activation: MAIN::raise-shields : f-0
==> Activation: MAIN::detect-klingon : f-0
```

6.2.6 Execution Engine

The execution engine fires the RHS of the activated rules. Modern rule-based systems can not only modify, add and remove facts in the working memory but also execute other instructions. In Jess, a full programming environment is available, including all Java classes installed.

Both rules are fired in the order chosen by the agenda. They print the following output to the console:

```
Raise shields!
Klingon Alert Captain!
```

6.3 Applications

This Section briefly presents three applications of rule-based systems.

6.3.1 Rule-Based Expert System in the Medical Domain

The "Protocol Assistant" is a rule-based medical expert system used in the domain parotid tumors. It was developed to assist users either as a wizard that guides a user through a decision making process or to provide assistance to find relevant information on parotid tumors. It is accessed via a web interface. The Protocol Assistant uses different categories of clinical evidence that are weighted according to their "goodness" [Bri98, p.1], i.e. the level of quality. By clicking on a decision node, the user is provided with all the relevant information objects. The authors of the Protocol Assistant point out that presenting all the relevant information object instead of only most relevant objects, is a key advantage of their application. The rule-based system used for the Protocol Assistant is Jess. It was chosen for its good integration with Java [Bri98, p. 7].

6.3.2 Workflow Automation

Business processes are likely to be adapted over time to meet the demands in a dynamic business environment. [KRSR98] propose a rule-based workflow coordination approach to provide the flexibility needed to adapt to change. The system is implemented using an
active object-oriented database and Event-Condition-Action (ECA) rules. Implementing coordination policies in ECA rules permits to codify knowledge independently of concrete workflows and eases modification and reusability [KRSR98].

6.3.3 Rule-Based System for Monitoring

The Spaceport Processing Systems Branch is responsible for the Space Shuttle prelaunch checkout at the NASA’s Kennedy Space Center. The Launch Processing System (LPS) supports the different actors involved in this process. It monitors the space shuttle as well as ground equipment, control instruments, etc. In total, around 50’000 parameters are monitored and controled, including temperatures, voltages, pressures, etc. After the Columbia accident in 2003, it was decided to improve LPS by increasing the insight and the situation awareness as well as the ability to better monitor contractors. To address this problem, NASA developed the "NASA Engineering Shuttle Telemetry Agent" (NESTA) as a complement to the existing LPS. NESTA uses the rule-based system Jess. The data sources of LPS are distributed over a local network. An agent can be attached to up to four data sources. An engineer specifies with Jess rules which measurements she is interested in and what action is executed if a given rule is fired (e.g. send an e-mail or page the engineer) [SDL+05, p. 1–6].
Jess [12] is a rule-based system shell entirely written in Java. It is developed at the Sandia National Laboratories in Livermore, California. Jess was inspired by the open-source expert system shell C Language Integrated Production System (CLIPS). Jess itself is not open-source, though the source code can be acquired by obtaining a license. Academic institutions can apply for a license for free and the binaries are available for download at no cost for 30 days usage [12]. Jess programs are written in the Jess rule language (a List Processing (LISP) like syntax) or in XML and are interpreted by Jess rule-based system [11]. This Chapter gives an introduction to Jess. Further documentation is available at [12] and [Hil03].

7.1 Running Jess

Code extract 7.1: Using the Jess Java API
There are several ways to run Jess:

**Standalone:** Jess can be executed as a standalone program. The Jess instructions are written in a ASCII file that is read by the interpreter.

**Interactive command-line interface:** Jess offers a command-line interface to dynamically write the Jess program.

**Java API:** Jess is entirely written in Java. It can be embedded in a Java program and accessed through its Java API. An example of how the Java API of Jess is used is shown in Code extract 7.1. The object that encapsulates the Jess rule-based system is instantiated in line 1. The Jess parser object is instantiated in line 2 with the file that contains the Jess instructions and the Jess rule-based system object as argument. To run Jess, the instance method run of the Rete instance is called.

### 7.2 The Jess Rule Language

The fundamental unit in the Jess rule language are **tokens**. Tokens can be either **symbols**, **numbers**, **strings** or **comments**. Internally, all Jess values are represented by instances of the `jess.Value` class [Hil03, p. 43].

**Symbols** The symbol concept can be compared to *identifiers* in Java. A symbol is a sequence that can contain letters, numbers, and the following punctuation marks: $, *, =, +, /, <, >, _, ? and #. The first character of a symbol may not be a number or punctuation mark or $, $ and =. These characters may however appear at other positions in the sequence. Jess symbols are case sensitive. Some symbols have special meanings: The symbol nil is equivalent to the keyword null in Java, TRUE and FALSE are boolean values and when printed (e.g. when printing a message on the console with `printout t crlf`) will cause carriage return [Hil03, p. 42].

**Numbers** The Jess parser uses `java.lang.Int.parseInt` and `java.lang.Double.parseDouble` to parse integer and floating points from values. The types are inferred when the token is parsed [Hil03, p. 43].

**Strings** Strings in Jess are denoted using `. Like in Java, embedded quote symbols are escaped with backslashes. Other escape sequences known in Java are not recognized, e.g. \n will not cause a carriage return [Hil03, p. 44].

**Comments** Jess supports two types of comments: LISP-style comments with semicolon `;` or C style comment, appearing between the delimiters `/*` and `*/` [Hil03, p. 44].

**Lists** are the basic unit of structures in Jess. They are both a central way of structuring code and data. A list consists of an enclosing set of parentheses with zero or more tokens or other lists, e.g.:

```
1 (+ 1 2)
(+ 1 (* 2 4))
3 (printout t "hello, world!" crlf)
```

The first element of a list is called the head. In many cases, the head of the list is special. For a function call, the head is the name of the function [Hil03, p. 45–46]. In the example above, + and printout are the function names, followed by their arguments. The function list will add 1 and 2, the second first multiply 2 by 4 and add the value to 1 and the third example print `hello, world!` on the console. Jess has different ways of interpreting
lists, depending of the context in which they appear. This will be elaborated as this Section continues.

Programming variables in Jess start with a \( \text{'?'} \). Unlike variables in Java, Jess variables are *untyped* and can thus have any value of any type assigned. A value is assigned to a variable with the `bind` function [Hil03, p. 47–48]. The following examples assignees "Foo Bar" to the variable \( \text{?x} \):

```
(bind ?x "Foo Bar")
```

Every variable created at the top-level of a Jess program are cleared when the `reset` function is called. `reset` re-initiates the working memory. To create variables that are not cleared with `reset`, Jess offers the possibility of using global variables. Global variables are created with the `defglobal` function and must begin and end with a \( \text{'*'} \) [Hil03, p. 48–49]. The following example shows how the string "bar" is assigned to the global variable \( \text{?*foo*} \). With the `bind` function, new values can be assigned to global variables.

```
(defglobal ?*foo* = "bar")
```

In Jess, control flow is handled by functions as well. Jess includes control flow functions `if`, `while`, `for`, `try` and `foreach` [Hil03, p. 50–53]. The listing below shows an `if` and `while` example:

```
(bind ?i 10)
(if (> ?i 5) then
  (println t "Go" crlf)
  else (println t "stop" crlf))
(while (> ?i 1)
  (−− ?i))
```

In this simple example, the integer 10 is assigned to the variable \( \text{?i} \). Because \( \text{?i} \) is larger than 5, "Go" is displayed on the console. Following that, in the `while` loop \( \text{?i} \) is reduced to 1. Other control flow are not discussed in this thesis. The reader is invited to consult the Jess manual at [12].

### 7.3 Working with Java Classes and Objects

As already mentioned above, a very powerful feature of Jess is the integration with Java. This Section gives as a basic introduction on how to use the Java API of Jess.

#### 7.3.1 Java Objects

Java objects are instantiated with the `new` function. The `new` function creates an object and returns the pointer of that object that can be assigned to a variable with the `bind` function. To avoid typing the full qualified name of a class, the class can be imported with the `import` function [Hil03, p. 63]. Java methods are called with the `call` function. For instance methods, `call` can be omitted [Hil03, p. 64–66]. In the following example, the `Vector` class is imported, an object of the `Vector` class instantiated and a string value "test" is added to that object by calling its `put` method:

```
(import java.util.Vector)
(bind ?v (new Vector))
(?v v put "test")
; or with the call function
(call ?v put "test")
```
7.4 Working Memory

Jess will automatically convert the type of the argument passed to a Java method to the so called target type [Hil03, p. 63].

7.3.2 Java Class Members

As mentioned in the previous Section, Java class methods must be called with the call function. Class fields can be read and modified with get-member respectively set-member [Hil03, p. 66–70] or, in a Java like syntax, with Member.Field. In the following example, Math.random is called and Math.PI printed:

```java
(call Math random)
(printout t (Math.PI) crlf)
```

7.4 Working Memory

The working memory, or fact base, contains a set of information objects called facts. Facts are a core element of Jess since rules only react to facts being added, deleted or changed. Rules are discussed in Section 7.5. Jess knows three types of facts: pure facts, unordered facts and shadow facts. The structure of facts are defined by templates (comparable to Java classes that define the structure of objects). Templates can be created with the deftemplate or defclass construct or in some cases, automatically. This Section discusses the three fact types Jess uses, with a special focus on shadow facts and a set of commands to manipulate the working memory.

7.4.1 Unordered Facts

Templates for unordered facts are defined with the deftemplate construct followed by the name of the fact, and zero or more slots. The name of deftemplate is the head of the facts. Additionally, a deftemplate construct can be described with a description between two " [Hil03, p. 82]. In this example, a template is created for bank accounts and one bank account is loaded into the working memory using assert.

```
(deftemplate account
(slot number)
(slot balance)
(slot currency))

(assert (account (number 211) (balance 500) (currency CHF)))
```

The order in which the slots are listed is irrelevant and not every slot has to be used [Hil03, p. 83].

7.4.2 Shadow Facts

Shadow facts are unordered fact that serve as a bridges between Java objects of the application that communicates with Jess and the facts in the working memory of the Jess instance that is used by that application. Its slots corresponds to the properties of the Java Beans class the shadow fact represents. Shadow facts are however not the objects themselves but “shadows” of the objects in the application. Shadow fact templates can be created automatically by telling Jess that a given template is derived from a given
Java class.
The following example explains how shadow facts are used. It consists of a Java Bean Notebook that has two fields which are accessed through getters and setters.

```java
public class Notebook implements Serializable {
    private String hostname;
    private String type;

    public void setHostname(String hostname) {
        this.hostname = hostname;
    }

    public String getHostname() {
        return this.hostname;
    }

    public void setType(String type) {
        this.type = type;
    }

    public String getType() {
        return this.type;
    }
}
```

Jess offers two possibilities to connect a Java class to a Jess template:

```jess
(deftemplate Notebook (declare (from-class Notebook)))
:or
(defclass Notebook Notebook)
```

The slots do not have to be specified, Jess uses the java.beans.Introspector API to obtain the properties. To add a shadow fact from within the Java application that uses Jess, the add method of the Rete object is called. This example assumes that the Rete object is jess:

```java
Notebook nb = new Notebook();
nb.setHostname("Galadriel");
nb.setType("Thinkpad");
jess.add(nb);
```

To add a shadow fact from within Jess, a Java object first has to be instantiated and added to the working memory:

```jess
(bind ?n (new Notebook))
(add ?n)
:or
(definstance Notebook (new Notebook)
```

To synchronize a Java object with its shadow fact, the update method is used. update takes a reference of a shadow fact as argument.

### 7.4.3 Ordered Facts

Ordered facts are flat lists. They do not have a structure of named fields defined by a template. Thus the order of the values matters, e.g.:

```jess
(location "N 51:25:07.4" "W 124:01:50.7")
```

### 7.5 Rules

Jess supports both forward-chaining and backward-chaining rules (briefly described in Section 6.1). Every time changes to the working memory occur (a fact is added, modified or removed), activation records for the rules that match the affected facts are added to or removed from the agenda (see Section 6.2.5). Thus the pattern matching of the LHS of a rule to a fact is performed at this stage, not when the rule is fired. The RHS of the activated rules in the agenda are fired one at the time when the run command is issued. [Hill03, p. 96–98].
Rules are created with the `defrule`, followed by a symbol that is the name of the rule and an optional documentation string. The `=>` symbol separates the LHS from the RHS. It can be read as `then`. The LHS consists of zero or more conditions and the RHS of zero or more actions.

### 7.5.1 Slot Constraints

In most cases, a set of constraints describe which conditions facts must satisfy to activate the given rule. Jess offers (i) **Literal constraints** (ii) **Variable constraints** (iii) **Connective constraints** (iv) **Predicate constraints** (v) **Return value constraints**. Each constraint type is briefly introduced in the following Sections.

#### Literal constraints

Literal constrains specify which exact slots values a fact must have to satisfy this constraints [Hil03, p. 101], e.g.:

```
(person (name "jon doe") (age 40))
```

This pattern matches facts of the type `person` with the name “jon doe” and age 40.

#### Variable constraints

Variable constraints bind matched values to variables. These variables are accessible in the scope of the rule [Hil03, p. 102]. The same variable can appear in several patterns but must contain the same value, e.g.:

```
(person (name ?n))
```

#### Connective constraints

Literal values and variables can be connected with the connective constraints. In a single expression, several connective constraints can be combined. the connective constraints are `&` **and**, `|` **or** and `not` [Hil03, p. 110–111]. e.g.:

```
(console (output "hello" | "world"))
```

This test is true for `console` facts that contain an output slot with either the value "hello" or "world".

#### Predicate constraints

Predicate constraints permit to use **predicate functions** (any boolean function that returns either `TRUE` or `FALSE`). A predicate function has to be preceded by `:`. To apply the predicate function to the value of a slot, the value has to be bound to a variable with the `&` connective [Hil03, p. 105–106], e.g.:

```
(product (price ?p: (> ?p 100)))
```

This matches any product that has a price higher than 100.
Return value constraints

Return value constraints are used to compare the return value of a function to the value of a slot [Hil03, p. 106–107], e.g.:

1. \((\text{product} \ (\text{price} \ ?p = (\times \ 2 \ 50)))\)

This matches \text{product} facts that have \text{price} slot with the value 100.

7.5.2 Pattern bindings

Pattern bindings is used to save a pointer to a fact that has been matched in the LHS of a rule to a variable. This permits to manipulate the fact with functions retract, modify or duplicate in the RHS of the rule [Hil03, p. 107], e.g.:

1. \((\text{defrule bind-me} \ ?a \ <-(\text{fact}) \ \Rightarrow \ (\text{retract} \ ?a))\)

In this example, the rule matches every fact of the type \text{fact}, saves a pointer to the current fact in variable ?a and removes it from the working memory.

7.5.3 Conditional Elements

While slot constrains are applied to the values of individual facts, conditional elements permit to group patterns for matching entire facts and relationships between facts. One conditional element, test, does not imply facts at all. Each conditional element is briefly explained in this Section.

"and" Conditional Element

The and conditional element can enclose a number of patterns that all have to be matched for the and conditional element to be valid [Hil03, p. 107], e.g.:

2. \((\text{defrule and-rule} \ (\text{and} \ (\text{system-up}) \ (\text{network-working})) \ =>)\)

This rule fires only if a fact of the type \text{system-up} and a fact of the type \text{network-working} are in the working memory.

"or" Conditional Element

The or conditional element encloses a list of patterns, of which at least one has to be matched for it to be true [Hil03, p. 108], e.g.:

2. \((\text{defrule or-rule} \ (\text{or} \ (\text{flight-available}) \ (\text{car-available})) \ =>)\)

This rule fires if at least a \text{flight-available} fact or a \text{car-available} is in the working memory.
"not" Conditional Element

The not conditional element applies to the nonexistence of a fact in the working memory [Hil03, p. 112]. Two examples following examples shall bring clarity:

```
(defrule on-remove
  (not (network-working))
  =>)
```

The first rule fires if a network-working fact is removed from the working memory. The second rule fires if there is a flight fact in the working memory but not a train fact.

"exists" Conditional Element

The exists conditional element is true if there exists a fact that matches its pattern in the working memory [Hil03, p. 112], e.g.:

```
(defrule check-flight
  (exists (flight (destination "Vancouver")))
  =>)
```

This rule fires if there is at least one fact in the working memory of the type flight with the string value Vancouver as the value of its slot destination.

"test" Conditional Element

A test conditional element does not evaluate a pattern that matches a fact. Instead, it matches against a boolean function.

```
(defrule test
  (test (eq (+ 1 1)) 2)
  =>)
```

This pattern is independent of any facts. It will always evaluate to true. According to [10], the test conditional element is used "when there is no preceding pattern, or with time-varying return values" [10].

"logical" Conditional Element

The logical conditional elements defines dependencies between a fact matched in the LHS of a rule with a fact asserted in the RHS of the same rule. The depending fact is also removed when the fact it logically depends is retracted from the working memory [Hil03, p. 115], e.g.:

```
(defrule logical-test
  (logical (foo))
  =>
  (assert (bar)))

(bind ?x (assert (foo)))
=> f-0 (MAIN::foo)
<Fact-0>
(run)
=> f-1 (MAIN::bar)
```
This example shows how a logical fact is removed when the fact it depends on is retracted from the working memory.
Part III

Implementation
8

Jess and JADE Integration: The Matrix Example

8.1 Introduction

The Matrix was developed to explore the possibility of using Jess in a multi-agent system to manage the decision making and working memory of an intelligent agent. The idea of the Matrix is based on the "The Matrix Trilogy" [30] movies. At one point in the second movie, the main character Neo, who is the Chosen One, has to fight a set of hostile agents Smiths that try to kill him. This prototype "replays" this combat. It prints the results
8.1. Introduction

Figure 8.1: The results of a combat between one Chosen One agent and two Smith agents of the combat to the console and to every remote Jess Administration agent, shown in Figure 8.1 (see Chapter 9 for details on remote Jess Administration). The reader is invited to test the Matrix by following the instructions in Appendix C.

The agents participating in the Matrix form a master/slave hierarchy. The master agent is called "Oracle agent". It manages and coordinates activities between all slave agents. It is fully aware of its environment and of each agent’s state. The intelligence of the Oracle agent is handled by Jess. Instead of hard-coding the business logic in Java as JADE agent behaviors, JADE is merely used for the agent communication: JADE basically accepts a message and passes the content of the message to Jess. Based on the current information obtained of its environment and what it has "learned" in the past, Jess reasons about the future actions it must take. This could be simply storing the data it obtained or ask the Oracle agent to send a message to one of its slaves. The multi-agent environment is depicted in Figure 8.2. The Jess Administration domain, used to manage the Jess instance of the master agent, is discussed in Chapter 9.

This Chapter explains how the Jess 2 JADE (J2J) tool kit (elaborated in details in Chapter 10) developed for this thesis, can be used to support process automation. The rest of this Section describes the coordination among the agents and the agent environment. Section 8.2 gives a detailed description of Jess rules developed to manage the Matrix environment. Agent communication related aspects are described in Section 8.3 followed by Section 8.4 in which agent behaviors and the conversations they support are described. Section 8.5 closes this Chapter with a discussion on the agent classes of the Matrix environment.
8.1.1 Agent Organization

The coordination among the agents is based on the Organizational structuring approach: The master agent, in this context referred to as the "Oracle agent", manages information about each agent. It is fully aware of its environment and coordinates and assigns tasks to slave agents to ensure that the system is in a coherent state. The other agents are purely reactive (agent architectures are introduced in Section 4.2). They do not keep a state of their own and do not have any sort of intelligence. Thus every action executed by a slave agent, except for looking up the "Oracle agent" with the DF and asking for its first action, is coordinated by the Oracle agent. A third type of agent that does not directly take part in the process executed by the Oracle agent and its slaves is the Jess Administration agent (discussed in Section 9.2). It is not required for the Matrix to run but offers a useful interface to access and manage the Jess rule-engine of the "Oracle agent" and to view the ontology that is used by the "Oracle agent" and its slaves to communicate.

8.1.2 Agents Environment

The agents' environment is, based on the definition of [RN03, p. 39], "artificial". Their percepts are messages from other agents and for some agents keyboard input and mouse clicks. They act on the environment by sending messages to other agents or by writing
the output to an output device such as the console. As stated above, this environment has two distinct types of agents, reactive agents and one intelligent agent. For both types, the environment from their perspective is described according to the dimensions cited in Section 4.1.2.

Master Agent’s Environment

From the perspective of the Oracle agent, the environment is:

- **Partially observable** The Oracle agent cannot obtain all information that are relevant for deciding which action it should execute next through its sensors. This is because other agents do not keep a state of their own. Thus the Oracle agent must keep an internal state to keep track of its environment.

- **Strategic** The environment is deterministic (the next state of the environment depends on the current state of the environment and the master agent’s next action). It is strategic however because other agents’ action (currently only a administrator agent for the Oracle agent) can also have an influence on the environment.

- **Sequential** The environment is sequential because the Oracle agent’s current decision depends on events and decisions made in the past, not only on the current state of the environment.

- **Dynamic** The environment can change in the course of time: New agents can be added that have to be taken into consideration, software and hardware failure may make part of the environment unavailable, etc.

- **Continuous** The environment has no defined limited amount of states, the agent can take a continuous set of actions and over a continuous time period.

- **Multi-agent** From the Oracle agent’s point of view, it is a cooperative multi-agent system. The Oracle agent does not compete with other agents to maximize its performance measurements.

Slave Agent’s Environment

From a slave agent’s point of view, the environment is:

- **Fully observable** A slave agent keeps no inner state, it can perceive all information it needs to act from its environment since it gets its instructions from a master agent.

- **Stochastic** The next state of the environment may not depend on the current state of the environment and the agent’s action.

- **Episodic** The agent perceives information from its environment and acts upon it, it does not include percepts made in the past to influence the current decision.

- **Dynamic** The agent constantly receives input from its environment on which it has to act.

- **Mainly discrete** The set of actions available to the agent are finite as well as the percepts from its environment.

- **Multi-agent** A slave agent is in a competitive multi-agent system. Slave agent’s compete for orders from the Oracle agent and to survive the combat.
8.2 The Matrix Workflow

The fight between the agents that are at enmity with each other executes the processes depicted in the UML Activity Diagrams of Figure 8.4 and Figure 8.5. The message flow between the participating agents for the same processes are depicted with the Sequence Diagrams of Figure 8.6 and Figure 8.7. When launched, every agent registers itself with the Oracle agent. The Oracle agent will decide on the agents’ strength, health and armor and assign the agent to its potential opponents (i.e. (Smith1, Chosen One) tuple). When the agent has received the confirmation that it has been registered, it will send a Command Me request to ask the Oracle agent what it is supposed to do next. If the agent is still alive (only the Oracle agent knows that), the agent will receive a Wait and wait for further instructions. If the agent has opponents, the agent will get the order to attack one of his opponents. The attacking agent then sends a message to its victim saying it has been attacked, upon which the victim will ask the Oracle agent what it should do (again with Command Me). If the agent is dead (the Oracle agent has sent Die), the agent also waits but will not be contacted in the future by the Oracle agent. If there are still opponents left that are alive, the Oracle agent will pick another agent and order it to attack its opponent. This process ends when either all Smith agents are terminated or the Chosen One agent has been beaten.

8.2.1 State of a Matrix Agent in Jess

All agents known to Jess are slave agents of the type CombatAgent. They each have the same comportment. A slave agent’s life phases are shown in the UML State Diagram of Figure 8.3. It passes through the following phases: At first the agent is registered. This is followed by a "training session" during which the agent receives its attributes healthstate, armor and power. It is then ready and waits. In this state, it can obtain orders to attack an other agent or be attacked itself. When it is attacked, its healthstate may be reduced. If the agent is still alive, it waits again, otherwise it dies and terminates.
8.2.2 Implementation of The Matrix Workflow With Jess

This Section discusses the Jess rules that implement the process. Each Jess rule managed by the Oracle agent’s Jess instance is shown in this Section. They are mapped to their corresponding activities in the Activity Diagrams in Figure 8.4 and Figure 8.5.

Initialization Workflow

Figure 8.4: Initialization Workflow

When agent first registers itself with the Oracle agent, its properties are empty. The Oracle agent "discovers" the agent’s property by training it and associates the new agent with its opponents. This workflow is shown in Figure 8.4. The Activity Diagram does not show the data used for the process. Each action is described in the remaining of this Section.

- **Train new Agent** When an agent is newly added to the working memory, its properties are not set. The Oracle agent "trains" the agent to make it ready for combat. The values are chosen randomly. A Chosen One agent is better trained than a Smith agent, otherwise it would not be able to fight a large number of Smith agents.

```
(defrule MAIN::train-agent-smith
    (?newbie <-(SmithAgent (power 0.0) (healthstate 0.0) (armor 0.0)) =>
    (modify ?newbie
        (power (+ (* (call Math random) 20.0) 5.0))
        (healthstate (+ (* (call Math random) 10.0) 5.0))
        (armor (+ (* (call Math random) 10.0) 5.0)))
    (printout t "new agent smith" crlf)
    (focus INIT-SYS))
```
8.2. The Matrix Workflow

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(defrule MAIN: :train-agent-chosenone)</td>
<td>Train new agent</td>
</tr>
<tr>
<td><img src="image" alt="Code extract 8.1: Train new agent" /></td>
<td></td>
</tr>
</tbody>
</table>

- **Assign Opponent** After an agent is trained, relations between the new agent and its opponents are set.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(defrule INIT--SYS::set-opponent)</td>
<td>Assign opponents</td>
</tr>
<tr>
<td><img src="image" alt="Code extract 8.2: Assign opponents" /></td>
<td></td>
</tr>
</tbody>
</table>

**Main workflow**

The UML Activity Diagram of Figure 8.5 depicts the workflow during which Combat agents fight each other. The data used by the activities are mostly omitted to avoid bloating the diagram.

- **Check Agent’s Health** Depending on whether or not there is a Killed fact in the working memory assigned to the agent that has sent the current CommandMe, the agent is going to wait or die.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(defrule AGENT--FATE::agent--alive :entry point if agent is alive)</td>
<td>Check agent’s health</td>
</tr>
<tr>
<td><img src="image" alt="Code extract 8.3: Check agent’s health" /></td>
<td></td>
</tr>
</tbody>
</table>

- **Remove Opponent Relations for Killed Agent** A dead agent cannot participate in the fight. Therefore, all relations to its opponents are removed from the working memory.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(defrule CLEANUP::cleanup--attackable--smith)</td>
<td>Remove opponent relations for killed agent</td>
</tr>
<tr>
<td><img src="image" alt="Code extract 8.3: Check agent’s health" /></td>
<td></td>
</tr>
</tbody>
</table>
8.2. The Matrix Workflow

---

**Figure 8.5: Main Workflow**

---

**Code extract 8.4: Remove opponent relations for killed agent**

```
(defrule CLEANUP::cleanup—attackerable—neo
  ?die <- (Die (slave ?v))
  ?attackable <- (Attackable (goodGuy ?v))
  =>
  (retract ?attackable)
  (printout t "retracting settings for neo" ?v crlf))
```

**"Assert Key" and "Set Key Secret"** If no key is yet created, a Key fact is added to the working memory and a secret is assigned to the key. The key is basically a token that an agent must possess to receive orders to attack.

```
(defrule AGENT—FATE::assert—key—fact ; only create a key if a fight is possible
  (CommandMe (slave ?s))
  =>
  (defrule AGENT::assert—key—fact
    ?agent <- (CombatAgent (agentName ?s) (healthstate ?hsc&: (> ?hsc 0)))
    (exists (Attackable (badGuy == ?s || goodGuy == ?s)))
    (not (Key)))
```
8.2. The Matrix Workflow

```lisp
(printf t " asserting key" crlf)
(assert (Key (secret "illuminati")))
(focus FIRST-KEY-ASSIGNMENT))
(defrule FIRST-KEY-ASSIGNMENT::assign-key
?cm <- (CommandMe)
?key <- (Key (secret ?sc))
=>
(printf t "assigning new secret" crlf)
(modify ?cm (secret ?sc))
(return))
```

Code extract 8.5: Add new key to the working memory

- **Wait** Every agent that is alive (i.e. its healthstate is higher than 0) is ordered to Wait.

```lisp
(defrule LIFE-STATE::send-wait-neo
(declare (auto-focus TRUE))
(Key (secret ?sc))
(CommandMe (slave ?s) {secret == ?sc})
(ChosenOne (agentName ?s) (healthstate ?hscs &: (> ?hscs 0)))
(SmithAgent (agentName ?sa) (healthstate ?hscc &: (> ?hscc 0)))
(Attackable (goodGuy ?s) (badGuy ?sa))
?w <- (Wait (slave ?s))
=>
(printf t "Sending lifestate to agent " ?s crlf)
(command-agent ?w ?s)
{return-fact-to-current-behaviour ?w)
(clear-focus-stack)
(focus DETERMINE-COMBAT)
```

Code extract 8.6: Wait

- **Die** An agent that had been killed previously by another agent is sent a Die order. If there are no more agents left that can fight each other, the workflow is terminated.

```lisp
(defrule LIFE-STATE::send-die
(declare (auto-focus TRUE))
?d <- (Die (slave ?s))
```

Wait

Every agent that is alive (i.e. its healthstate is higher than 0) is ordered to Wait.

Die

An agent that had been killed previously by another agent is sent a Die order. If there are no more agents left that can fight each other, the workflow is terminated.
8.2. The Matrix Workflow

8.2. The Matrix Workflow

(not (Attackable {goodGuy == ?s || badGuy == ?s}))
(exists (Attackable)): there are still opponents left.
(exists (SmithAgent (healthstate ?hscc&: (> ?hscc 0))))
(exists (ChosenOne (healthstate ?hscc&: (> ?hscc 0))))

(command-agent ?d ?s)
(printout t "dying information sent to " ?s crlf)
(clear-focus-stack)
(focus DETERMINE-COMBAT)

(defrule LIFE-STATE::send-die-end-of-combat
(declare (auto-focus TRUE))
?d <- (Die (slave ?s))
?cm <- (CommandMe (slave ?s))
(or
(not (ChosenOne (healthstate ?hscc&: (> ?hscc 0))))
(not (SmithAgent (healthstate ?hscc&: (> ?hscc 0))))
)

(command-agent ?d ?s)
(retract ?d)
(retract ?cm)
(printout t "dying information sent to " ?s " it's finally over now" crlf)
(halt))

Code extract 8.7: Die

- **Assert Attack and Select new Attacker** If the agent that has sent CommandMe is dead, a totally new opponents pair has to be found (rule attack-after-die). Otherwise, the agent that has sent the CommandMe is considered to be the attacker and a new opponent is assigned to it.

(defrule DETERMINE-COMBAT::attack-for-neo
?cm <- (CommandMe (slave ?s) (secret ?sc))
?key <- (Key (secret ?sc))
?neo <- (ChosenOne (agentName ?sn) (power ?p))
?smith <- (SmithAgent (agentName ?s))
?at <- (Attackable (goodGuy ?s) (badGuy ?sn))
(not (Killed (victim ?s) (killer ?sn)))

(printout t "New fight between " ?s " and " ?sn " the secret is" ?sc crlf)
(assert (Attack (victim ?sn) (attacker ?s) (strength ?p) (secret ?sc)))
(retract ?cm)
(return))

(defrule DETERMINE-COMBAT::attack-for-smith
?cm <- (CommandMe (slave ?s) (secret ?sc))
?key <- (Key (secret ?sc))
?neo <- (ChosenOne (agentName ?sn) (power ?p) (healthstate ?hscc&: (> ?hscc 0))))
?smith <- (SmithAgent (agentName ?s))
?at <- (Attackable (goodGuy ?s) (badGuy ?sn))
(not (Killed (victim ?s) (killer ?sn)))

(printout t "New fight between " ?s " and " ?sn " the secret is" ?sc crlf)
(assert (Attack (victim ?sn) (attacker ?s) (strength ?p) (secret ?sc)))
(retract ?cm)
(return))

(defrule DETERMINE-COMBAT::attack-after-die ;only smiths can die without the process stopping
?d <- (Die (slave ?slave))
?cm <- (CommandMe (slave ?slave))
?key <- (Key (secret ?sc))
?neo <- (ChosenOne (agentName ?s) (power ?p) (healthstate ?hscc&: (> ?hscc 0))))
?smith <- (SmithAgent (agentName ?sn) (healthstate ?hscc&: (> ?hscc 0)))
?at <- (Attackable (goodGuy ?s) (badGuy ?sn))

(printout t ?slave " died New fight between " ?s " and " ?sn " the secret is" ?sc crlf )
(assert (Attack (victim ?sn) (attacker ?s) (strength ?p) (secret ?sc)))
(retract ?cm)
8.3. Agents Communication

The communication between agents in the Matrix is fully compliant with the FIPA standard. The content is based on clear defined semantics. Agents’ behaviors used in this section include:

- **Attack!** The newly asserted Attack fact is sent to the agent that is set as the attacker.

  ```prolog
  (defrule COMBAT-INIT::enter-combat
    (declare (auto-focus TRUE))
  3 ?a <- (Attack (attacker ?at) (secret ?s))
  4 =>
  5 (command-agent ?a ?at))
  ```

  Code extract 8.9: Attack!

- **Reduce Opponents Life** The victim’s health state is reduced by the amount the attack exceeds its armor level.

  ```prolog
  (defrule COMBAT-INIT::execute-attack
    (declare (salience 100))
  4 ?ca <- (CombatAgent (agentName ?v) (healthstate ?hs) (armor ?ar))
  5 =>
  6 (println t "Reducing life of agent " ?v " to " (+ ?hs ?ar) ?st crlf)
  7 (modify ?ca (healthstate (+ ?hs ?ar) ?st))
  8 (focus COMBAT))
  ```

  Code extract 8.10: Reduce opponents life

- **Opponent Killed** If the opponent is killed, a Killed fact that contains a reference to the victim, is set. When the victim sends a CommandMe fact, it will be notified that it has died.

  ```prolog
  (defrule COMBAT::agent-killed
    (CombatAgent (healthstate <= 0) (agentName ?v))
  3 =>
  4 (println t "Agent " ?v " has died, he will be notified soon" crlf)
  5 (assert (Killed (victim ?v) (killer ?at))))
  ```

  Code extract 8.11: Opponent killed

- **Opponent Still Alive** If the attack was not strong enough to kill the opponent, the opponent remains in the system and will be the attacker next time it sends a CommandMe with the valid key.

  ```prolog
  (defrule COMBAT::agent-still-alive
    (CombatAgent (healthstate hsc & (> ?hsc 0)) (agentName ?name))
  3 =>
  4 (println t "Agent " ?name " is still alive" crlf)
  5 (retract ?a))
  ```

  Code extract 8.12: Opponent still alive

8.3 Agents Communication

The communication between agents in the Matrix is fully compliant with the FIPA standard. The content is based on clear defined semantics. Agents’ behaviors used in this...
thesis extend the FIPA-protocol classes provided by the JADE framework. Each behavior pair (for the initiator and the participant) handles exactly one agent action type. Hence a sequence of tests inside a behavior to determine which operations have to be executed, can be omitted. Instead, each behavior on the participant’s side uses a message template that is tailored to a given action with a given FIPA-protocol and message performative. The different aspects of communications between agent are discussed in this Section.

8.3.1 Agent Interaction

The agent interaction in the context of the Matrix main workflow can be divided into two phases, as already discussed in Section 8.2. Both steps are discusses below with focus on the messages the agents send. The reader shall note two distinctions from UML standard sequence diagrams:

- They do not represent objects sending messages to each other but agents communicating. This is more coarse grained and is caused by several objects that have to communicate.
- Additionally, the signals contain performatives (Request) and in parenthesis the action the receiving agent is to execute, not, as it is normally shown in Sequence Diagrams, method calls and parameters.

Agent Initialization

![Figure 8.6: Slave agent registration process](image)

Figure 8.6 shows the set of messages sent when an agent registers itself. With message 1, the Oracle agent registers itself with the DF. It can now be looked up by other agents in the environment. Message 3 is the query the slave agent sends to the DF to
8.3. Agents Communication

Code extract 8.13: Content of a FindMe message

```java
({
  (action
    (agent-identifier
      :name OracleAgent@wlan-per21-244-231.unifr.ch:1099/JADE
      :addresses
        (sequence http://wlan-per21-244-231.unifr.ch:7778/acc))
    (FindMe
      :newAgent
        (ChosenOne
          :agentName Neo@wlan-per21-244-231.unifr.ch:1099/JADE
          :power 0.0
          :healthstate 0.0
          :armor 0.0
          :master OracleAgent@wlan-per21-244-231.unifr.ch:1099/JADE)))
}
```

look up the Oracle agent. The fourth message is the reply that contains the requested data. If the Oracle agent is not yet online, the result in the reply would be empty. The slave agent would try the lookup again at a later point in time. Now that the slave agent has the Oracle agent’s Uniform Resource Identifier (URI), it sends a FindMe message to tell the Oracle agent to register it and to decide on its properties, namely the power, healthstate and armor. An example of a FindMe message content is shown in Code extract 8.13. The Oracle agent replies to this message by returning the slave agent’s attributes it "found" (actually generated randomly). The reader may note that the agent action FindMe is not used directly in the content of the message but inserted in an action construct. The SL content language requires that agent actions are inserted into an instance of a jade.content.onto.basic.Action class that associates the agent action with the AID (field agent-identifier) of the agent that is to perform the action [BCG07, p. 87]. The creation of messages is discussed in Section 10.2. The ontology used in this domain is presented in Section 8.3.2.

Main Process

A simple scenario of the main workflow with two agents is shown in Figure 8.7. First, the Smith1 agent sends a CommandMe message to the Oracle agent (first message). The Oracle agent confirms that it has received the request (message 2) and sends a Wait (message 3). Smith 1 confirms that it has processed the message (message 4). In this case, the Oracle agent does not send any further instructions to Smith1. This is because no opponent has yet been registered. The registration process of Neo is done with messages 5 and 6, as discussed in the Section above. Agent Neo sends a CommandMe action in message 7, which is confirmed by the Oracle agent. The situation now is the following: the Oracle agent is aware that there are two opponents, both with health levels above 0 and there is no current combat ongoing. It still sends a Wait message to Neo, which is directly followed by an Attack message (message 9 and 10). The communication is asynchronous and in this situation it is not important for the Oracle agent to wait for the confirmation that agent Neo has processed the Wait action. The slave agent Neo confirms both former messages with messages 11 and 12. Neo reacts to the Attack message by informing Smith1 that it has been attacked with a UveBeenAttacked message. This message contains the token (not shown in Figure 8.7) needed to be recognized by the Oracle agent as the agent replying to the attack (message 13 and confirmation message 14). This prompts Smith1 to send again a CommandMe request in message 15, this time with the valid token. This message
is shown in Code extract 8.14. The body of this message contains an action construct, as explained above, with the AID of the agent that is intended to perform the action. The CommandMe action has three attributes: the secret, i.e. the token that permits an agent to get additional orders besides Wait, the slave that is sending this message and the master that handles this action. The Oracle agent confirms that he is processing the message with message 16 and sends a message requesting the slave to execute a Die action. Hence the attack from the Neo agent was more than the current healthstate and armor of the Smith1 agent combined could withstand. This caused the Oracle agent to declare Smith1 dead next time it contacts the Oracle agent. There are no more opponents left, the system is now idle.

Every interaction is based on the FIPA-Request-Protocol, described in Section 4.3.2. One might note that every interaction contains two communicative acts: a Request send by the initiator and a Inform send as a reply by the participant. The original FIPA-Request-Protocol foresees that an Agree is sent by the participant after it has received the Request and before it has sent the message containing an Inform performative. This step is however optional and can be omitted if the action is performed very quickly [8]. This is the case with all actions performed in the given scenario.
8.3. Agents Communication

Code extract 8.14: Content of a CommandMe message

8.3.2 The Matrix Ontology

The Matrix ontology is used by agents that take part in the workflow that represents a combat between a set of agents of the type SmithAgent and one agent of the type ChosenOne. The ontology is shown in Figure 8.8. This ontology is used by JADE and Jess and also contains concepts and properties that are only used by Jess. Section 10.6.1 describes the algorithm that transforms the JADE ontology into Jess templates. The elements defined by JADE will not be elaborated on. They are the following: Concept, Predicate, AID and AgentAction. The reader is invited to read more on the JADE content reference model in [BCG07, p. 80].

This ontology was modeled with Protégé. The concepts and predicates are the following:

- **MatrixAgent**: MatrixAgent contains every agent involved in the Matrix process. It has one attribute, agentName that contains the name of the agent. The value should be the AID of the agent.

- **CombatAgent**: CombatAgent is the set of agents directly involved in the combat. It defines the following attributes: armor which defines the amount of the attack that is absorbed without harming the attacked agent, healthstate, which states the...

Figure 8.8: Matrix Ontology
current health of an agent and power, which states with which strength an agent can attack another agent.

- **ChosenOne** The ChosenOne set should contain no more than one instance. This instance is the opponents of all agents that are instances of the set SmithAgent.

- **SmithAgent** The SmithAgent set contains instances that are opponents of the agent instance of the ChosenOne set.

- **OracleAgent** The OracleAgent set has with the current implementation one instance, namely the master agent.

- **FindMe** A FindMe instance is the action sent by each new agent to request registration. Its attributes are newAgent, it contains the values of the agent sending the request and master, the string value of the AID of the master.

- **MatrixAction** MatrixAction is the set of all actions set during the main workflow. It defines the attribute secret, which contains the token an agent must possess to be recognized by the Oracle agent as the agent authorized to get further instructions besides Wait.

- **MatrixCombatAction** A MatrixCombatAction is an action that is involved in an attack act. It specifies the attributes victim that holds the string value of the AID of the agent that has been attacked and attacker, the string value of the AID of the attacking agent.

- **UveBeenAttacked** A UveBeenAttacked instance is sent to the attacked agent by the attacker to inform the victim of its fate.

- **Attack** An Attack instance is used by the Oracle agent to signal one of its slave agents that it must attack the opponent specified in the victim attribute. It defines the attribute strength, which contains the strength of the attack.

- **NonCombatAction** Instances of the set NonCombatAction are used for actions that are not related to an attack act. It defines the attribute slave. The value of this attribute is the string value of the AID of the agent that receives a given instance.

- **LifeStateAction** Instances of this set are agent action that tell an agent whether it is alive or dead.

- **Wait** A Wait instance is issued to an agent that has requested to be commanded and is still alive. An agent may receive several Wait instances before the main workflow is terminated.

- **Die** A Die instance is issued once to an agent which healthstate is 0 or less.

- **CommandMe** CommandMe is the set of requests sent by agents who are alive and wish to know what the Oracle agent has in mind for them.

- **Attackable** An Attackable instance relates two opponents. It specifies the following attributes: badGuy that holds the string value of the AID of a SmithAgent and goodGuy, the string value of the AID of a ChosenOne.

- **Killed** A Killed instance relates the agent that has been killed and the agent that killed it. It defines two attributes: victim that holds the string value of the AID on an agent who has died and killer, the string value of the AID of the agent that killed the victim.
8.3. Agents Communication

```java
public class MatrixCommunication extends CommunicationConfigFactory {
    private static MatrixCommunication Singleton;
    public static MatrixCommunication getInstance() {
        if (Singleton == null)
            Singleton = new MatrixCommunication();
        return Singleton;
    }
    private MatrixCommunication() {
        this.setCodec(new SLCodec());
        this.setContentLanguage(FIPANames.ContentLanguage.FIPA_SL0);
        this.setOntology(The_MatrixOntology.getInstance());
        this.setOntologyName(The_MatrixOntology.ONTOLOGY_NAME);
    }
}
```

Code extract 8.15: Extending the CommunicationConfigurationFactory to set the ontology, content language and codec

- **Attacked** An Attacked instance relate two agents involved in one attack act. For each new act, a new Attacked instance is created. It defines two attributes: victim, the string value of the AID of the agent that was attacked and attacker, the string value of the attacker’s AID.

- **Key** A Key instance holds the token, stored as a string value in the attribute secret used by a slave agent to be recognized as the agent with current "attack" access to the Oracle agent.

### 8.3.3 Using the Communication Configuration Factory

This Section explains how the Communication Configuration Factory (described in Section 10.2), is used for agents in the Matrix environment.

**Extending the CommunicationConfigurationFactory class**

The Communication Configuration Factory provides methods to create different types of objects that are relevant to agent communication. A concrete factory has to specify which ontology, content language and codec is used. This is achieved by extending the CommunicationConfigurationFactory with a concrete factory that calls the protected setters as shown in Code extract 8.15. The setters for the communication relevant parameters are called in the private constructor of the concrete factory class. Due to the nature of this concrete factory, only one instance of this class is needed, as long as it is globally accessible. The Singleton design pattern is thus the natural choice.

**Creating a message**

An agent that wishes to send a message using the Communication Configuration Factory calls the Factory Method that creates a message for the performative and FIPA-Protocol the agent requires. The example, shown in Code extract 8.16, is for the Request performative and the FIPA-Request-Protocol. First, the agent action is created (lines 1–4). At line 5, the AID of the receiver is created. The Communication Configuration Factory is
8.4 Agent Behaviors

This Section discusses internal and communicational behaviors developed for the Matrix system. The conventions to name and structure communicational behaviors is discussed in Section 4.3.2.
8.4. Agent Behaviors

Figure 8.9: Behaviors to find the master agent and conduct a set of setup tasks

8.4.1 Internal Behaviors

Setup Behavior

Every agent has a behavior of the type FindOracle that conducts a set of setup tasks. The class diagram is shown in Figure 8.9. The FindOracleTemplate abstract class implements the Template Method onTick that, once it has found the master agent (i.e. the Oracle agent), calls the primitive operation loadBehaviorsAndRegistrationMessage. For each agent type that needs to communicate with the master agent, a subclass has to be written that implements the loadBehaviorsAndRegistrationMessage method. These are agents taking part in the Matrix combat and Jess Administration agents. For both types, the concrete implementations will send registration requests; for slave agents taking part in the Matrix workflow it is the FindMe requests (see Code extract 8.14). Jess Administration agents register themselves with the Oracle agent by sending a RegisterMe request (to be able to communicate with the Jess instance hosted by the Oracle agent) and a RegisterOntologyListener to get updates of the ontology used by Jess. The Remote Jess Administration is discussed in Chapter 9.

Get Orders

The GetOrders behavior is an internal behavior that is used by a slave agent to initiate a CommandMe conversation. This behavior is activated on two occasions:

- The Oracle agent confirms that the slave agent which has just sent a FindMe request, is registered (FindMeInitiator behavior).
- The slave agent has received a UveBeenAttacked agent action from its enemy.

This behavior is instantiated once per agent and then "recycled".

8.4.2 Communicational Behaviors

This Section first describes the conversations that take place between agents in the Matrix environment, followed by an explanation of how a new communicational behavior is implemented.
Agent Conversations for the Matrix

- "Attack" Conversation An attack conversation is carried out by the Oracle agent and a slave agent. The Oracle sends a Request performative that contains an Attack agent action to a slave agent. The slave agent reacts to the agent action by initializing a "You have been attacked" conversation with the slave agent the Oracle agent decided it has to attack.

- "You Have Been Attacked" Conversation This conversation is between the attacking slave agent and its victim. The attacker sends a Request performative containing a UveBeenAttacked agent action that contains the token (called secret) the slave agent needs to be recognized by the Oracle agent as the agent responding to the Attack command. The participant’s UveBeenAttackedResponder re-activates a OneShotBehavior that initialized the "Command Me" conversation with the Oracle agent.

- "Die" Conversation A "Die" conversation takes place between the Oracle agent and a slave agent that has recently requested to be commanded. The slave agent was killed in a previous fight but does not yet know this. The Oracle agent sends a Request performative containing a Die agent action. The slave agent responds by writing to the console that is has died.

- "Wait" Conversation A "Wait" conversation is between the Oracle agent and a slave agent that requested to be commanded and is still alive. The slave agent receives the message but does not do anything.

- "Find Me" Conversation A "Find Me" conversation takes place between a slave agent and the Oracle agent. A slave agent registers itself with the Oracle agent and requests that the Oracle agent discovers its properties (health, armor, power). It does so by sending an Inform communicative act that contains a FindMe which in turn contains incomplete information on the sender. The Oracle agent activates upon reception of FindMe message its FindMeResponder behavior. This behavior adds the agent to the working memory of Jess, runs Jess to reason on its newly added fact and returns update to the slave agent. The slave agents merely prints the information to the console.

- "Command Me" Conversation After a slave agent of the type ChosenOne or SmithAgent is registered with the Oracle agent, it can ask the Oracle agent to provide it with new orders. It starts a "Command Me" conversation by sending CommandMe agent action. The Oracle agent replies to this request by returning a simple Done.

Adding a New Communicational Behavior

This Section discusses how new FIPA-Request-Protocol behaviors are added by extending the classes RequestInitatorTemplate and RequestResponderTemplate to implement a new conversation. The conversation in this example is "Find Me". Unless stated otherwise, the algorithm of these methods is of no importance for this Section.

To create a new Initiator behavior, the RequestInitatorTemplate is extended and the hook method that accepts the expected performative and predicate overwritten, as shown in
8.4. Agent Behaviors

Code extract 8.17: Creating an Initiator behavior by extending `RequestInitiatorTemplate`

```java
public class FindMeInitiator extends RequestInitiatorTemplate {
    private static final long serialVersionUID = 4858773633255342724L;

    public FindMeInitiator(Agent a, ACLMessage msg) {
        super(a, msg);
    }

    protected void executeInformResult(Concept agentAction, Object resultValue, ACLMessage msg) {
        //do something
    }
}
```

Code extract 8.18: Creating a Responder behavior by extending `RequestResponderTemplate`

```java
public class FindMeResponder extends RequestResponderTemplate {
    public FindMeResponder(Agent a, MessageTemplate mt) {
        super(a, mt);
    }

    private static final long serialVersionUID = 5898353514161562132L;

    protected Predicate executeAction(Concept agentAction) throws NotUnderstoodException, RefuseException {
        FindMe fm = (FindMe) agentAction;
        //do something
        Result r = new Result();
        r.setValue("SOMERESULT");
        this.setMessagePerformative(ACLMessage.INFORM);
        return r;
    }
}
```

Code extract 8.17. If the initiator does not intend to handle the Inform returned by the participant, no hook method has to be overwitten.

A Responder behavior is created by extending `RequestResponderTemplate`. An example based on the `FindMeResponder` class is given in Code extract 8.18. To execute successfully, this method has to set at some point the message performative by calling `setMessagePerformative` (line 14) and returning a predicate of the type `Result` or `Done`. It is to be noted that the agent action does not have to be set for the predicate, this is handled by the Template Method. Thus for a `Done`, the predicate can simply be returned by calling `return new Done()`. For the `Result`, the result created by the behavior has to be added to it by calling its method `setValue` (line 13).

Adding a new Jessable Communicational Behavior

This example shows how the `CommandMeResponder` class (shown in the Class Diagram of Figure 8.10), which is a concrete Jessable behavior, is realized. Jessable Behaviors are discussed in Section 10.7.1. It implements the participant’s behavior in a "Command Me" conversation based on the FIPA-Request-Protocol. The implementation is shown in Code extract 8.19. In addition to overwriting `executeAction`, a concrete Jessable behavior has to implement `setCommunicationConfigFactory` to specify which concrete communication factory is to be used and thus which ontology. `executeAction` does the following: it adds
8.5. Software Agents

This Section discusses the software agents for the Matrix environment. For reasons of compatibility with MediMAS [18], every Matrix agent has an agent state instance field, it is however not used since the Matrix slave agents are stateless. The class hierarchy for agents in the Matrix is shown in Figure 8.11. All classes inherit from the TheMatrixAgent abstract class that extends the JADE Agent class.

8.5.1 TheMatrixAgent

The TheMatrixAgent abstract class defines a set of primitive methods that the setup method calls. It specifies the following instance fields which can be accessed through setters and getters:

- **self** This instance field holds a MatrixAgent instance of itself. The agent sends this object during the registration process to register itself with the Oracle agent.
- **oa** This instance field holds the AID of the Oracle agent.
- **state** This instance field is of the type AgentState. It is currently not used.

The following Template Method is implemented:

- **setup** This Template Method is shown in Code extract 8.20. At lines 2–6 the agent’s content manager is setup. This is described in details in Section 10.2.
public class CommandMeResponder extends JessableRequestResponderTemplate {

public CommandMeResponder(Agent a, MessageTemplate mt) {
    super(a, mt);
}

protected Predicate executeAction (Concept agentAction) throws NotUnderstoodException, RefuseException {
    CommandMe cm = (CommandMe) agentAction;
    this.jessCNC.getJessEngineManager().addFactToJess(cm);
    this.jessCNC.getJessEngineManager().setModuleFocus("AGENT-FATE");
    this.jessCNC.getJessEngineManager().runJess();
    this.setMessagePerformative(ACLMessage.INFORM);
    return new Done();
}

public CommunicationConfigFactory setCommunicationConfigFactory() {
    return MatrixCommunication.getInstance();
}

Code extract 8.19: Implementation of the CommandMe responder class

Figure 8.11: Class hierarchy for agents in the Matrix environment

it calls the primitive operation prepareAgent. Subclasses can add additional setup instructions in this method. At line 9, it loads the behaviors the agent wants to load at initialization time. At lines 9–11 a WaitResponder behavior is loaded.

Subclasses have to implement the following primitive operations:

- **prepareAgent** This method permits subclasses to add additional agent setup instructions.
- **loadInitialBehaviours** Subclasses can add additional behaviors that have to be loaded during setup.
- **setAgentOntologyObjectShell** This method creates and configures the agent’s instance from the ontology which it will use to register itself with the Oracle agent.
8.5. Software Agents

8.5.2 Slave Agents

Slave agents in the Matrix environment are either of the type SmithAgent or ChosenOne. Both classes extend TheMatrixAgent to implement the methods setAgentOntologyObjectShell to set their agent type of the ontology and loadInitialBehaviours to load a FindOracleMatrix behavior which will handle the registration process with the Oracle agent; prepareAgent is left empty. Except for the agent type set with setAgentOntologyObjectShell, the implementation is otherwise the same. The implementation for the class SmithAgent is shown in Code extract 8.21.

public void setAgentOntologyObjectShell() {
    ontology.matrix.SmithAgent sa = new ontology.matrix.SmithAgent();
    sa.setAgentName(this.getAID().getName());
    sa.setArmor(0);
    sa.setHealthstate(0);
    sa.setPower(0);
    this.setSelf(sa);
}

protected void loadInitialBehaviours() {
    this.addBehaviour(new FindOracleMatrix(this,10000));
}

Code extract 8.21: Setup of a slave agent

8.5.3 Master Agent

The OracleAgent class extends TheMatrixAgent class and implements the JessableAgent interface. The methods of interest are shown in Code extract 8.22. loadInitialBehaviours loads the behaviors for the Matrix domain and for the Jess Administration domain. Each behavior is provided with a message template to restrict its activation to a certain conversation type. The prepareAgent method instantiates JessCNC and additionally configures its ontology manager to use messages that use the Jade Administration ontology, which it will apply to communicate with Jess Administration agents (lines 17–20). The Oracle agent then registers itself with the DF to enable its slaves to look it up (lines 21–28).

protected void loadInitialBehaviours() {
    MessageTemplateFactory mtf = FipaRequestFactory.getInstance();
    mtf.createProtocolInitialisationWithAction(this, FindMe.class));
}

Code extract 8.22: Setup of an Oracle agent
8.5. Software Agents

```java
this.addBehaviour(new CommandMeResponder(this,
    mtf.createProtocolInitiationWithAction(this, CommandMe.class)));
this.addBehaviour(new RegisterMeResponder(this,
    mtf.createProtocolInitiationWithAction(this, RegisterMe.class)));
this.addBehaviour(new ExecuteJessCommandResponder(this,
    mtf.createProtocolInitiationWithAction(this, ExecuteJessCommand.class)));
this.addBehaviour(new RegisterOntologyTreeListenerResponder(this,
    mtf.createProtocolInitiationWithAction(this, RegisterOntologyTreeListener.class)));
}

protected void prepareAgent() {
    this.jessCNC = new JessCNC(this);
    CommunicationConfigFactory cf = JessCommunication.getInstance();
    AgentConfigObject aco = cf.createAgentConfigObject();
    this.getContentView().registerOntology(aco.getOntology());
    DFAgentDescription dfd = new DFAgentDescription();
    dfd.setName(getAID());
    ServiceDescription sd = new ServiceDescription();
    sd.setActive(true);
    sd.setAgentName("Oracle-Agent");
    sd.setLocal(true);
    dfd.addServices(sd);
    try {
        DFService.register(this, dfd);
    } catch (FIPAException fe) {
        fe.printStackTrace();
    }
}

public JessCNC getJessCNC() {
    return this.jessCNC;
}
```

Code extract 8.22: Setup the Oracle agent

### 8.5.4 Jess Administration Agent

To administer the Jess instance of the Oracle agent remotely, a concrete Jess Administration agent was created, called `JessAdministrationAgent`. The GUI of the Jess Administration agent is shown in Figure 8.1. The `JessAdministrationAgent` extends the abstract class `AbstractJessAdminAgent` and implements `registerWithMasterAgentAndLoadBehaviors`. This method adds a `FindOracleJessAdmin` behavior to its agent. The `FindOracleJessAdmin` behavior registers the agent with the Oracle agent and loads the required behaviors. Remote Jess Administration is discussed in details in Chapter 9.
9

Remote Jess Administration

9.1 Introduction

A Jess Administration agent permits to work with Jess similar to having direct console access to a Jess instance. It it is an autonomous part of the J2J tool kit. This Chapter explains the implementation of the remote Jess Administration. Section 9.2 explains how the Jess Administration is used. The next Section, Section 9.3, describes the ontology that the master agent and Jess Administration agents use to communicate. In Section 9.4, the possible conversations between a Jess Administration agent and a master agent as well as the necessary extensions written to enable these conversations, are discussed. The abstract Jess Administration agent class is elaborated on in Section 9.5.

9.2 Using the Jess Administration Agent

As indicated in Figure 8.2, a master agent can be accessed by special agents to remotely administer its Jess rule-engine instance. These agents are called Jess Administration agents. The GUI of Jess Administration agent is shown in Figure 9.1. Each part is marked with a red frame. The current implementation covers basic features: sending Jess commands, receiving Jess output and viewing the ontology. To give an example of how the remote Jess Administration agent can be used, the rule MAIN::train-agent-chosenone is replaced during run time to add only low attribute values to a Chosen One agent, thus
reducing the likelihood that it can win. The new rule prints "NEW RULE TO SET NEO's ATTRIBUTES", after is was transmitted by having clicked on the eval button. The following output shows that the agent Neo is weakened and loses the fight.

JESS OUTPUT NEW RULE TO SET NEO's ATTRIBUTES
JESS OUTPUT New opponents: badguy is ...
  ...
JESS OUTPUT Reducing life of agent Neo@Kaleena.local...
JESS OUTPUT Agent Neo@Kaleena.local:1099/JADE has died, ...
Agent Smith14 is still alive and waiting ...
JESS OUTPUT retracting settings for neoNeo@Kaleena.loc...  
JESS OUTPUT dying information sent to Neo@Kaleena.local: ..

9.3 Agents Communication

For the Jess Administration agent to be able to communicate with the master, a new ontology was developed and the corresponding Communication Configuration Factory implemented (see Section 10.2 for details on the Communication Configuration Factory). The ontology is elaborated on in this Section.

9.3.1 Jess Administration Ontology

The Jess Administration ontology, shown in Figure 9.2, defines the semantics of the content communicated between Jess Administration agents and the master agent that
hosts Jess. Unlike the Matrix ontology, it is not used by Jess itself and therefore not mapped to Jess templates. The elements defined by JADE will not be elaborated on. They are the following: Concept, Predicate, AID and AgentAction. The reader is invited to read more on the JADE content reference model in [BCG07, p. 80].

The Jess Administration ontology consists of the following sets:

- **RegisterOntologyListener** A RegisterOntologyListener is sent by an agent that wishes to be notified of changes to the ontology. It specifies the attribute agentName which contains the AID of the subscriber.

- **RegisterMe** A RegisterMe instance is sent by an agent that wants to be able to access Jess remotely and obtain output generated by Jess. It defines the attribute agentName which contains the AID of the subscriber.

- **UnRegisterMe** An UnRegisterMe instance is sent by an agent that does not wish to be provided with Jess related messages anymore. The attribute agentName contains the AID of the agent that is unregistering.

- **UpdateBuffer** An UpdateBuffer instance is sent to a Jess Administration agent. It contains new output generated by Jess. The output is stored in the attribute content.

- **ExecuteJessCommand** An ExecuteJessCommand instance is sent by a Jess Administration agent to the master agent hosting Jess. It requests Jess to execute an arbitrary Jess command. The command is stored in the attribute content.

- **OntologyTreeChanged** An OntologyTreeChanged instance is sent whenever the ontology tree is changed to every agent registered as an ontology listener. It defines two attributes; agentName contains the AID of the recipient and the attribute content the structure of the tree.

### 9.4 Agent Behaviors and related Classes

This Section first states each conversation that the communicational behaviors implement. It then explains how these conversations are implemented by extending classes of the J2J tool kit.
9.4.1 Communicational Behaviors

A Jess Administration agent and a master agent that hosts Jess use the following conversations:

Agent Conversations for Jess Administration

This Section discusses the conversations between administration agent (or agents) and the master server that hosts Jess.

- "Register Me" Conversation  A "Register Me" conversation occurs when a Jess Administration agent requests to be registered with the master agent to receive the output generated by Jess. The Jess Administration agent sends an Inform performative containing a RegisterMe agent action. The master agent activates its RegisterMeResponder behavior that adds the agent to the list of agents which will be notified whenever Jess generates output.

- "Update Buffer" Conversation  When a Jess Administration agent is registered with the master agent, the master agent notifies each registered agent whenever Jess generates output by sending a Request performative containing an UpdateBuffer agent action which has the latest console output as content. The participant responds to this message by calling its UpdateBufferResponder behavior which extracts the string value and updates the object that displays the value.

- "Execute Jess Command" Conversation  A Jess Administration agent that wants to execute a Jess command on the master agent starts the "Execute Jess Command" conversation by sending a Request performative that contains an ExecuteJessCommand agent action with the Jess command as string value. The master activates its ExecuteJessCommandResponder behavior which passes the command to Jess. If the command is successful, the agent returns an Inform performative with a Done predicate, otherwise it returns a Failure performative with the error message provided by Jess as the content of a Result predicate. The results are displayed by the initiator.

- "Register Ontology Tree Listener" Conversation  A Jess Administration agent that wishes to be notified when the ontology used by Jess changes, starts this conversation by sending a Request performative containing a RegisterOntologyTreeListener. The master agent adds the agent to the list of agents that want to be notified of updates to the ontology tree.

- "Ontology Tree Changed" Conversation  Every time the ontology tree used by Jess is updated, the master agent starts an "Ontology Tree Changed" conversation with each agent registered as an Ontology Tree Listener by sending a Request performative with an OntologyTreeChanged agent action that contains the ontology tree in a fully parenthesized syntax.

9.4.2 Implementation of the Agent Conversations

The conversations for the Jess Administration focus: (i) on relaying commands to Jess and the output generated by Jess to the Jess console and (ii) on providing the structure of the ontology used by the Jess engine. For both participating agents, the implementation
protected Predicate executeAction (Concept agentAction){
    RegisterMe rm = (RegisterMe) agentAction;
    JessableAgent ja = (JessableAgent) this.myAgent;
    ja.getJessCNC().getAgentOutputWriter().addStrategy(rm.getAgentName().getName(), new JadeMessageOutputStrategy(this.myAgent, rm.getAgentName()));
    this.setMessagePerformative(ACLMessage.INFORM);
    return new Done();
}

Code extract 9.1: Adding a JadeMessageOutputStrategy

required on the Jess Administration agent’s side and on the master agent’s side are explained.

Jess Console Input and Output

This Section first explain the registration process for a Jess Administration agent to receive Jess console output and how the output generated by Jess and the command sent to the Jess engine, are transmitted.

Registration for Console Output and Receiving the Output  Before an agent can receive Jess output remotely, it must register itself with the AgentOutputWriter object. The agent accomplishes this by initiating a "Register Me" conversation (for Jess Administration, not Matrix slaves), as discussed in Section 9.4.1). The master agent’s RegisterMeResponder behavior adds a new JadeMessageOutputStrategy instance to the AgentOutputWriter object. It sets the agent’s name as the key of the Strategy (the Strategy design pattern is discussed in [GHJV95, p. 315]). Strategies are stored in a HashMap object. The JadeMessageOutputStrategy class extends the JessOutputStrategy class. The AgentOutputWriter class and the JessOutputStrategy class are discussed in Section 10.5.3. A JadeMessageOutputStrategy object sends the string it receives from its Context (i.e. an AgentOutputWriter instance) to its agent. The executeAction method of the RegisterMeResponder behavior is shown in Code extract 9.1. For each agent that wishes to receive Jess output, a new Strategy is instantiated.

When the writeOutput method of a JadeMessageOutputStrategy is called, it initiates an "Update Buffer" conversation with its designated agent and transmits the output. The receiver’s UpdateBufferResponder accepts the message and passes it to the Jess Administration agent’s GUI object to update its text field. This is shown in Code extract 9.2. The Sequence Diagram of Figure 9.3 depicts the agent communication for the conversations "Register Me" and "Update Buffer" between the master agent "OracleAgent" and one administration agent "Admin". The participants in the Sequence Diagram are agents, not objects, and their interaction is based on ACLMessages not, methods calls.

protected Predicate executeAction (Concept agentAction){
    UpdateBuffer ub = (UpdateBuffer) agentAction;
    ((JessAdminAgent) this.myAgent).getJessAdminGui().updateOutput(ub.getContent());
    this.setMessagePerformative(ACLMessage.INFORM);
    return new Done();
}

Code extract 9.2: Updating a Jess Administration agent’s Jess output console
9.4. Agent Behaviors and related Classes

OracleAgent : OracleAgent
Admin : JessAdminAgent

Figure 9.3: "Register Me" and "Update Buffer" based on the FIPA-Request-Protocol

```
protected void executeInformResult (Concept agentAction, Object resultValue, ACLMessage msg)
{
    JessAdminAgent jaa = (JessAdminAgent) this.myAgent;
    jaa.getJessAdminGui().updateOutput(resultValue.toString());
}
```

Code extract 9.4: Implementation of the ExecuteJessCommandInitiator behavior

When the `eval` button is clicked (see Figure 9.1), the content of the text field is read and sent to the master agent by initializing a "Execute Jess Command" conversation. The `ExecuteJessCommandResponder` behavior adds the command to Jess by calling the Jess Engine Manager’s `executeJessCommand` method. If the command is valid, the behavior replies by sending an Inform `Done`. Otherwise it throws a `JessException`. The exception is

```
protected Predicate executeAction (Concept agentAction)
{
    ExecuteJessCommand ejc = (ExecuteJessCommand) agentAction;
    JessableAgent ja = (JessableAgent) this.myAgent;
    Predicate p = null;
    try {
        ja.getJessCNC().getJessEngineManager().executeJessCommand(ejc.getContent());
        this.setMessagePerformative(ACLMessage.INFORM);
        p = new Done();
    } catch (JessException e) {
        Result result = new Result();
        result.setValue(e.getMessage());
        p = result;
        this.setMessagePerformative(ACLMessage.FAILURE);
    }
    return p;
}
```

Code extract 9.3: Implementation of the ExecuteJessCommandResponder behavior
handled and the error message returned as a Failure. The information is passed to the GUI of the Jess Administration agent and displayed. The `executeAction` method of the `ExecuteJessCommandResponder` behavior is shown in Code extract 9.3, the handler method of the `ExecuteJessCommandInitiator` behavior in Code extract 9.4.

### Ontology Display

This Section first explains the classes that have to be added to the master agent for a Jess Administration agent to receive information on the ontology followed by a discussion on how the ontology information is handled by the Jess Administration agent.

#### Adding Observer and Visitor

To add support on the master’s side for transmitting the ontology remotely, two classes had to be added: An `Observer` of the Ontology Tree Manager and a `Visitor` that traverses the ontology on behalf of the Observer (the Observer design pattern is described in [GHJV95, p. 293], the Visitor design pattern in [GHJV95, p. 331]). The ontology management is part of the J2J tool kit. It is discussed in Section 10.6. The new Observer is the `RemoteTreeListenerManager` class, its new Visitor is the `LispingTreeVisitor` class:

- **RemoteTreeListenerManager** The `RemoteTreeListenerManager` class extends the abstract class `TreeUser`, as shown in Figure 10.10. It forwards the string representation of the ontology generated by its Visitor, `LispingTreeVisitor`, to a set of agents that have applied to receive updates of the ontology. Hence the `RemoteTreeListenerManager` object of the master agent can be seen as the `Subject` and the registered agents as its `Observers`. The Subject initiates a "Ontology Tree Changed" conversation (Section 9.4.1) to push the entire representation of the ontology to the Observers. This is the opposite to the approach chosen by the Ontology Tree Manager where the Observers pull the information they require. The Remote Tree Listener Manager updates its Observers on two occasions: when the Ontology Tree Manager notifies the `RemoteTreeListenerManager` object that the ontology has changed and every time a new agent registers itself. The Subject will only update Observers that have a time stamp older than the last update. A new agent has its time stamp set to 0, thus it will get the update.

  For an agent to receive information on the ontology composite, it has to register itself by initializing a "Register Me" conversation, as discussed in Section 9.4.1.

- **LispingTreeVisitor** A `LispingTreeVisitor` implements the `OntologyVisitor` interface, as shown in the UML Class Diagram of Figure 10.14. It creates string representation of the ontology using a fully parenthesized syntax (similar to the programming language LISP, therefore the name LISPing). Code extract 9.5 shows the result for the Matrix ontology, depicted in Figure 8.8. This syntax was chosen due to the fact that it is easy to create and parse and has, with two parenthesis per node, little overhead. The current implementation ignores attributes.

```
1 (ROOT Concept ( AgentAction ( MatrixAction ( MatrixCombatAction ( UveBeenAttacked ) ( Attack ))) ( NonCombatAction ( LifeStateAction ( Wait ) ( Die ) ( CommandMe ))) ( FindMe ))) ( MatrixAgent ( CombatAgent ( SmithAgent ) ( ChosenOne ))) ( OracleAgent ))) ( Predicate ( Attacked ) ( NoGo ) ( Attackable ) ( Killed ) ( Key )))
```

Code extract 9.5: Parenthesized representation of the ontology
9.4. Agent Behaviors and related Classes

Parsing the Ontology Representation The ontology string representation sent to a Jess Administration agent is parsed and transformed into a JTree. The transformation is handled by the Jess Administration agent’s OntologyTreeChangedResponder behavior. The transformation is implemented in two steps:

1. The character stream is parsed and divided into substrings. Each node and parenthesis is a string object. These objects are stored in a Vector.
2. The Vector generated in the previous step is read to generate a JTree that contains each class in the ontology.

Code extract 9.6: A simple tree that is parsed according to Figures 9.4(a) to 9.4(h)

1. (Root(A(B)))(C)

Figure 9.4: Steps performed to create a JTree representation of the ontology used by Jess

The algorithm performing the second step is discussed in detail in Appendix D, Section D.1. The algorithm first creates a "((String, DefaultMutableTreeNode)" pair for every String object created in the first step and adds these pairs in a Vector in the same order as the Strings in the Vector created in the first step. The second element of each pair is set to null. The DefaultMutableTreeNode class is used to create nodes of a JTree. For each pair in the new Vector, the algorithm executes the following steps: If the first element of the current pair is a left parenthesis, it is ignored. If it is a right parenthesis, it means that the end of a list has been reached and the element on top of the stack is the first element in a list. The element on top of the stack is popped and, if the stack is not empty, it is assigned as a child of the DefaultMutableTreeNode instance of the element that is now on the top of the stack (if the DefaultMutableTreeNode of the popped element or of the element that is now on the top of the stack is null, a new DefaultMutableTreeNode has to be created for each reference that is null). If the stack is empty, the root of the ontology has been reached. If the first element of the current pair is not a parenthesis, it is a node in the ontology tree representation and pushed onto the stack. To illustrate this algorithm, Figures 9.4(a) to 9.4(h) depict each step performed by the algorithm of the second step for the tree of Code extract 9.6.
9.5 Jess Administration Agents

The AbstractJessAdminAgent abstract class extends the GuiAgent class of the JADE framework. It aids the implementation of a GUI for JADE agents. The class hierarchy is depicted in Figure 9.5. A screenshot of a Jess Administration agent’s GUI is shown in Figure 9.1. The AbstractJessAdminAgent configures its content manager and instantiates the GUI object. Specifying AID of the master agent is delegated to its subclasses. The agent GUI was developed with the GUI designer plugin Jigloo [1] and will not be discussed further. The AbstractJessAdminAgent class is discussed in the next Section.

AbstractJessAdminAgent

This abstract class has the following methods:

- **setMaster** This method is called by a subclass to set the AID of the master agent.
- **getMaster** This method is called by adherent classes that require to know the AID of the master agent the Jess Administration agent needs to contact.
- **setup** This method configures the agent and then calls its instance method registerWithMasterAgentAndLoadBehaviors to set the master agent’s name and load the behaviors that are required to communicate with the master agent.
- **getJessAdminGui** This method returns a pointer to the GUI object of the Jess Administration agent.
- **registerWithMasterAgentAndLoadBehaviors** This abstract method is to be implemented by its subclasses to set the master agent and load the behaviors necessary to communicate with the master agent.
- **onGuiEvent** is specified by its super class. It is left empty.
- **loadDefaultBehaviors** This method loads the default behaviors needed for the remote Jess Administration agent to communicate with the master agent. It has to be called by a subclass once the master agent is known.
• **gUI** This instance field holds a reference to the GUI object of the agent.
• **master** This instance field holds the AID of the master agent. A subclass sets the value of this field by calling `setMaster`.
"This is your last chance. After this, there is no turning back. You take the blue pill - the story ends, you wake up in your bed and believe whatever you want to believe. You take the red pill - you stay in Wonderland and I show you how deep the rabbit-hole goes."

Morpheus

10

Jess and JADE Integration: The
Jess 2 JADE Tool Kit
10.1 Introduction

Chapter 8 explained by means of "the Matrix" example how the J2J tool kit, developed for this thesis, is used. This Chapter discusses in detail how the toolkit is implemented. In this Chapter, an agent that uses Jess to reason is referred to as a master agent. Section 10.2 elaborates on the generic agent communication issues. Section 10.3 describes generic behavior classes. In Section 10.4, the interface to access an agent's Jess instance, is discussed. Section 10.5 introduces the class `JessCNC` that is the point of contact for accessing the J2J tool kit. Section 10.6 delves into the details of the ontology management. Next, the framework to add JADE-integrated Jess user functions is explained in Section 10.7. This Chapter closes with a brief discussion in Section 10.8 of a master agent for the Matrix environment that is implemented without Jess.

10.2 Agent Communication Configuration

When an agent sends a message to another agent in JADE, it has to create an instance of the class `ACLMessage`. After that, it sets the receivers, the content language, the ontology and the content. This sequence of steps can be automated. Two assumptions are made:

- The steps involved in creating a message for a given communicative act and a FIPA-protocol will follow the same pattern.
- The content in a given domain using a given ontology will use the same codec and content language.

Two types of objects are interested in the parameters that define the syntax and semantics of the information transferred in a multi-agent environment:

- `ACLMessage` objects
- `AgentConfigObject` instances
• and the agents’ content managers that transform the content of a message from their string or bytes representation to Java POJOs and back from their Java object representation to a string or bytes.

Because both the ACLMessage and the content managers depend on the same information, it would make sense to store this information in one single class and have the configuration objects for the content managers and ACLMessages created by the same class. The Factory methods (the Factory method design pattern is described in [GHJV95, p. 107]) to create configuration objects and the different ACLMessages for each communicative act and FIPA-protocol can thus be reused for every ontology. Based on these considerations, an Abstract Factory (this design pattern is discussed in [GHJV95, p. 87]) was developed. For each ontology, the Abstract Factory is extended. A child class only needs to specify the ontology, the content language and the codec. The class diagram of the Abstract Factory is shown in Figure 10.1. The ACLMessage class (its interface is omitted in the diagram) is part of the JADE framework and will not be elaborated on in this Section. The rest of this Section discusses the Abstract Factory. It illustrates on the basis of the child class for the Matrix ontology how the Abstract Factory is extended and used to create messages and to configure the content manager of an agent.

Classes

• CommunicationConfigurationFactory The CommunicationConfigurationFactory is an abstract class that contains methods to create AgentConfigObject instances and ACLMessages. It defines the following private instance fields:
  
  – codec specifies the codec for the content language.
  – contentLanguage states the content language used for messages.
  – ontology holds a pointer the ontology object.
  – ontologyName contains the string value of the ontology name.

The fields are accessed through private setters and getters. The subclasses of the CommunicationConfigurationFactory class have to call these methods to set the proper values. As mentioned above, for each communicative act and FIPA-protocol for which messages are to be created, a Factory method has to be implemented. The Matrix application only relies on the FIPA-Request-Protocol and only the initial request message has to be build from scratch (the replies are generated automatically by calling the createReply method of a received ACLMessage instance). The method to create the request message for a FIPA-Request-Protocol performative is shown in Code extract 10.1. createFIPARequestMessage accepts as argument a pointer to the current Agent object, the AID of the receiver and the AgentAction the receiving agent is to perform (line 1). At line 2, an ACLMessage is instantiated and its constructor provided with the argument ACLMessage.REQUEST to specify that this message is used for a FIPA Request performative. Lines 3–5 configure the content language, the ontology and the receiver of the ACLMessage. In line 6–8 a Action is instantiated and provided with the receiver that is expected to perform the action and the AgentAction the receiver is to perform. Line 9 calls the agent’s content manager that transforms the Java objects into the specified content representation. The ready message is returned to the caller at line 10.

Agent configuration objects are created by calling the createAgentConfigObject in-
public ACLMessage createFIPARequestMessage(Agent myAgent, AID receiver, AgentAction agentAction) throws CodecException, OntologyException {
    ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
    msg.setLanguage(this.getContentLanguage());
    msg.setOntology(this.getOntologyName());
    msg.addReceiver(receiver);
    Action a = new Action();
    a.setActor(receiver);
    a.setAction(agentAction);
    myAgent.getContentManager().fillContent(msg, a);
    return msg;
}

Code extract 10.1: createFIPARequestMessage instance method

public AgentConfigObject createAgentConfigObject() {
    AgentConfigObject aco = new AgentConfigObject();
    aco.setCodec(this.getCodec());
    aco.setContentLanguage(this.getContentLanguage());
    aco.setOntology(this.getOntology());
    return aco;
}

Code extract 10.2: createAgentConfigObject instance method of the abstract class CommunicationConfigurationFactory

instance method, shown in Code extract 10.2. An AgentConfigObject object is created in line 1. Lines 3–5 set the content codec, the language and a pointer to the ontology that the content manager of the agent is to use. The ready object is returns in line 6.

- AgentConfigObject The AgentConfigObject is a simple JavaBean that encapsulates information an agent’s content manager needs to be able to use an ontology. It defines the following instance fields:
  - codec Contains the codec the agent’s content manager is to use.
  - contentLanguage Contains the content language the agent’s content manager is to use
  - ontology contains the pointer to the ontology object the agent’s content manager is to use.

10.3 Agent Behaviors

This Section discusses the behaviors class templates. Class templates were implemented as part of the generic toolkit. No generic internal behaviors are provided with this toolkit. Behaviors that are tightly linked with the J2J tool kit, referred to as Jessable behaviors, are elaborated on in Section 10.7.1 due to their tight integration with Jess.

10.3.1 Communicational Behaviors

Abstract FIPA-Request-Protocol Behaviors

The JADE framework provides two implementations for the FIPA-Request-Protocol:
• The AchieveREInitiator and AchieveREResponder
• The SimpleAchieveREInitiator and SimpleAchieveREResponder

The former is a more heavy weight implementation. Both take as constructor argument an ACLMessage that initiates the protocol, thus the initiator themselves contain the steps to send the initial message. The SimpleAchieveREInitiator and the SimpleAchieveREResponder are the lightweight implementation of the AchieveREInitiator and AchieveREResponder. The main difference is that heavyweight implementations can delegate states of the protocol to other, application specific behaviors [BCTR00, p. 8]. For the Matrix, the lightweight behaviors were sufficient. The SimpleAchieveREResponder and SimpleAchieveREResponder abstract class are both extended with the RequestInitiatorTemplate abstract class respectively the RequestResponderTemplate. These classes implement hook operations that can be overwritten by their subclasses. The RequestInitiatorTemplate and RequestResponderTemplate are described in the rest of this Section.

• RequestInitiatorTemplate The RequestInitiatorTemplate, shown in Code extract 10.3, is used by the initiator of a FIPA-Request-Protocol based conversation. As stated in the preceding section, the message to initiate the protocol is not created by the behavior but passed as a constructor argument to the behavior. This class contains two template methods, handleInform and handleFailure. handleInform is called when the participant successfully performed the action and returns either a Done to indicate that the communicative act was successful or Result to return results created by the performative. A subclass has to, depending on what kind of response is expected, either overwrite executeInformDone (called in line 24) or executeInformResult (called in line 28). executeInformDone expects as arguments an agent action and a message. This permits the hook operation to access the entire message, if required. executeInformResult expects, in addition to the agent action and the ACLMessage, the result returned by the participant.

The SimpleAchieveREResponder calls handleFailure when the participant could not successfully execute the performative requested by the initiator and returns a failure. If the failed performative was to return a Done predicate, the executeFailureDone operation in line 48 is called. Otherwise, if a result predicate was excepted, executeFailureResult is executed (line 52).

```java
public abstract class RequestInitiatorTemplate extends SimpleAchieveREInitiator {

    public RequestInitiatorTemplate(Agent a, ACLMessage msg) {
        super(a, msg);
        this.setProtocol(msg);
    }

    public RequestInitiatorTemplate(Agent a, ACLMessage msg, DataStore store) {
        super(a, msg, store);
        this.setProtocol(msg);
    }

    private void setProtocol(ACLMessage msg) {
        msg.setProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST);
    }

    protected void handleInform(ACLMessage msg) {
        try {
            ContentElement ce = this.myAgent.getContentManager().extractContent(msg);
            if (ce instanceof Done) {
                Done done = (Done) ce;
                Action action = (Action) done.getAction();
                this.executeInformDone(action.getAction(), msg);
            }
        } catch ( [...] }
```
10.3. Agent Behaviors

Code extract 10.3: RequestInitiatorTemplate

• RequestResponderTemplate The RequestResponderTemplate behavior, shown in Code extract 10.4, is used by the participant in a FIPA-Request-Protocol. It contains the Template Method `prepareResponse` that generalizes the algorithm for accepting a message containing a Request performative. The algorithm extracts the action and calls a hook operation that a subclass has to overwrite in order to execute the performative requested by the initiator. This class requires a more detailed description than its peer. At the beginning and at the end of the Template Method, the hook operations `doAtBeginning` (line 15) and `doAtEnd` (line 35) are called. These methods can be overwritten by their subclasses to handle tasks that should always be executed before the agent action is handled and when the behavior has finished executing the agent action. Because a JADE agents use cooperative multi-tasking by default and not preemptive multi-tasking to schedule behaviors [BCTR00, p. 25], it is guaranteed that a running behavior will not be interrupted to run another behavior of the same agent without its consent, thus no additional multi-threading handling is necessary. In lines 16–19, the message reply is automatically created and the agent action extracted from the incoming message. At line 20, the handling of the agent action is delegated to the subclass by calling the hook operation `executeAction` which accepts as argument the incoming agent action. To successfully execute, this method has to return a predicate of the type `Done` or `Result` and, at some point, set the message performative by calling `setMessagePerformative`. The
instance `messagePerformative` (line 3) is set to $-1$. This value will cause JADE to throw an error. By calling `setMessagePerformative` the subclass sets the `messagePerformative` field to a valid value. The message performative is set at line 26 with the `messagePerformative` value.

```java
public class RequestResponderTemplate extends SimpleAchieveREResponder {

    private int messagePerformative = -1;

    public RequestResponderTemplate(Agent a, MessageTemplate mt) {
        super(a, mt);
    }

    protected void doAtBeginning() {}

    protected void doAtEnd() {}

    protected ACLMessage prepareResponse(ACLMessage request) throws NotUnderstoodException, RefuseException {
        this.doAtBeginning();
        ACLMessage reply = request.createReply();
        try {
            Action a = (Action) this.myAgent.getContentManager().extractContent(request);
            Concept agentAction = a.getAction();
            Predicate p = this.executeAction(agentAction);
            if (p instanceof Done) {
                ((Done) p).setAction(a);
            } else if (p instanceof Result) {
                ((Result) p).setAction(a);
            }
            reply.setPerformative(this.messagePerformative);
            this.myAgent.getContentManager().fillContent(reply, p);
            catch (UngroundedException e) {
                e.printStackTrace();
            }
            catch (CodecException e) {
                e.printStackTrace();
            }
            catch (OntologyException e) {
                e.printStackTrace();
            }
            finally {
                this.doAtEnd();
            }
            return reply;
        }

        protected Predicate executeAction(Concept agentAction) throws NotUnderstoodException, RefuseException{
            return null;
        }

        protected void setMessagePerformative(int messagePerformative) {
            this.messagePerformative = messagePerformative;
        }
    }
}
```

Code extract 10.4: RequestResponderTemplate

### 10.3.2 Message Templates

The Section above on agent communication stated that the participating agent accepts the incoming message and activates the proper `Response` behavior. To restrict a behavior that reacts to messages from being activated for every incoming message, the agent’s `receive` method called within a behavior has to be provided with the message template. `receive` will return the first message that matches the pattern defined by the template. A `SimpleAchieveREResponder` accepts a template as a constructor argument. An Abstract
10.3. Agent Behaviors

Factory was developed to automate the creation of message templates. The Class Diagram is shown in Figure 10.2. Classes and interfaces of the JADE framework (MessageTemplate, Literal, MatchActionExpression, AndExpression) have their interface omitted.

### Abstract Factory

**MessageTemplateFactory** The Abstract Factory MessageTemplateFactory currently defines one method signature to create message templates. `createProtocolInitialisationWithAction` expects a Java class object of an agent action as argument and returns a message template for the FIPA-protocol type and step in the conversation. For each FIPA-protocol used, a concrete factory should be developed.

**FipaRequestFactory** The concrete factory for the FIPA-Request-Protocol is the FipaRequestFactory class. In addition to the `createProtocolInitialisationWithAction` it offers a class method `getInstance` to access its Singleton object. The implementation of the `createProtocolInitialisationWithAction` for the FIPA-Request-Protocol is shown in Code extract 10.5. This method generates a template that matches a message with the following properties: it is a Request performative of a FIPA-Request-Protocol conversation and contains an agent action that is the instance of a class this method received as an argument. An object diagram of the object structure created by this method is shown in Figure 10.3. For
the sake of simplicity, the diagram is not fully UML conform. The object fields contain the name of class fields instead their values and the agent action class is referenced to as "AnAgentAction.class" to indicate that this field will hold a reference to an agent action class object.

Figure 10.3: Object composition for the message template created by the createProtocolInitialisationWithAction method

**Template Extension**

- **MatchActionExpression** To match the agent action of an incoming message, the MatchActionExpression was developed. It implements the MatchExpression interface and can thus be used in any template.

### 10.4 Software Agents

This Section introduces the interface to access the Jess instance of an agent.

#### 10.4.1 JessableAgent

JessableAgent is an interface for agents to implement that use Jess to reason. It specifies one instance method:

- **getJessCNC** This method is to return the JessCNC instance of the agent.

The approach of keeping the Jess relevant objects in the JessCNC object and letting an agent implement the JessableAgent interface was chosen in favor of developing an abstract Jess agent. With this approach, the Jess implementation can be added to existing agents without having a larger impact on their class structure.
This Section describes the class that manages the Jade and Jess integration. A high-level overview of the system is depicted in Figure 10.4. It shows how the system is assembled of several subsystems and reveals the interaction points between sub systems. The picture shows that the integration is managed by the Jess Command and Control (JessCnC) object. Inside this object, two objects stand out:

- The Jess engine that manages the working memory and rules to reason on facts that are in the working memory. Only Jess Engine Manager has a pointer to that object, all other objects have to go through Jess Engine Manager to access it.

- The ontology that structures the working memory and defines the semantics of the ACLMessage contents. The ontology is never accessed directly by objects that require its service. It is managed by the Ontology Tree Manager which lets other managers that fulfill a specific service, access it through Visitor objects.

The J2J tool kit can be used to control the entire business logic of an agent. In this case behaviors can be seen as channels used by Jess to communicate with other agents. It can however also be used to replace a certain subset of the decision making and thus also have part of the agent’s intelligence managed by standard behaviors.
### 10.5.1 Jess Integration

The core of the integration approach chosen for this thesis forms the class `JessCNC`. It contains a set of manager classes. Each manager assumes certain functionalities. `JessCNC` offers an interface to access to these managers and handles their integration. It additionally contains the instance method `configureMasterForJessRemoteAdministration` that will configure a master agent to communicate with remote Jess Administration agents.

The Class Diagram of `JessCNC` and the classes it uses is shown in the class diagram of Figure 10.5. The interfaces of the adjacent classes are omitted.

The initialization of a `JessCNC` object is shown in Code extract 10.6. It is divided in two steps:

1. It first loads the ontology and instantiates objects that will use the ontology (line 3 and lines 28–33). These objects are Observers that need the ontology to perform their tasks. Each Observer receives a Visitor object that will traverse the ontology for them and return information the Observers require. The Ontology Tree Manager, the object that manages the ontology, is the Subject that will notify its Observers when the ontology is changed or is newly created. The Observers must therefore register themselves with the Ontology Tree Manager. This is described in detail in Section 10.6.1.

2. Once the ontology is loaded into memory and the ontology Observers (also referred to as listeners) are ready, the second step is executed to prepare and launch the Jess rule-engine. It starts by instantiating a `AgentOutputWriter` object (line 8) (more in Section 10.5.3) and then instantiates the `JessEngineManager` (line 10), the object that controls access to the Jess rule-engine (Section 10.5.2).

This order of initialization steps is important: Jess will cause errors if it contains rules that use unknown templates. Because all templates in this implementation are generated from the ontology created for JADE, the ontology has to be loaded first. The algorithm

---

1 CNC stands for Command aNd Control and is a reference to the TV Series Babylon 5 [24]
verifies whether it is the first time `startJess` is executed (line 13). This check is necessary for the JADE to Jess Ontology Manager, the manager that maps the JADE ontology to Jess templates. The manager will restart Jess if the ontology class structure is altered or none is available. The `updateTree` method of the Ontology Tree Manager rebuilds the ontology tree and causes all Observers to update their representation of the ontology and in some cases perform additional tasks for their clients. This would cause the JADE to Jess Ontology Manager to restart Jess with new templates, thus causing an undesired infinite loop. Therefore, `JessCnC` and the JADE to Jess Ontology Manager need to treat the first time `startJess` is called as a special case. This is further discussed in Section 10.6.3.

Next, managers that require a valid reference to a Jess Engine Manager are started by calling the method `JessDependentManagers` in line 17. In this implementation this is merely the Jess Function Manager (line 36). Next, new Jess functions are added to Jess (line 18) and Jess is started (line 19). Finally, the `AgentOutputWriter` object is added to the Jess engine (line 20).

```java
public JessCNC(Agent masterAgent) {
    this.masterAgent = masterAgent;
    this.loadOntologyTreeManagers();
    this.startJess();
}

protected void startJess() {
    try {
        this.out = new AgentOutputWriter();
        FileReader JessDefaultRules = null;
        URL url = getClass().getResource("/expertSystem/rulefiles/ipd.clp");
        JessDefaultRules = new FileReader(url.getPath());
        this.jem = new JessEngineManager(this, JessDefaultRules);
        if (this.firstTime) {
            this.getOntologyTreeManager().updateTree();
            this.firstTime = false;
        }
        this.JessDependentManagers();
        this.addInitialFunctionsToJess();
        this.jem.initializeJess();
        this.jem.setAgentOutputWriter(this.out);
    }
    catch (FileNotFoundException e1) {
        e1.printStackTrace();
    }
    catch (JessException e) {
        e.printStackTrace();
    }
}

protected void loadOntologyTreeManagers() {
    this.treebeard = new OntologyTreeManager();
    this.classReg = new OntologyClassManager(new ClassCollectorVisitor(), this.
            getOntologyTreeManager());
    this.j2om = new Jade2JessOntologyManager(new Jade2JessVisitor(), this.
            getOntologyTreeManager(), this, this.firstTime);
    this.remoteTL = new RemoteTreeListenerManager(new LispTreeVisitor(), this.
            getOntologyTreeManager(), this.masterAgent);
}

protected void JessDependentManagers() {
    this.jfr = new JessFunctionsManager(this);
}
```

Code extract 10.6: Initializing `JessCNC`

### 10.5.2 Jess Engine Manager

The `JessEngineManager` class (see UML Class Diagram of Figure 10.6) encapsulates a `jess.Rete` object. The `jess.Rete` class implements the Jess rule-engine. At initialization
to the Jess rule-engine. The rule file is only parsed when `initializeJess` is called. This class contains the following instance fields:

- **jess** This instance field contains a reference to the Jess `jess.Rete` class.
- **initializationModule** This field contains the name of the Jess module that is to have the focus when the rule engine starts to run. It may be empty. It is set by calling `setInitializationModule` and accessed with the instance method `getInitializationModule`.
- **config** This instance field contains a `java.io.FileReader` object with the location of the default rule file. The rules are parsed when `initializeJess` is called for the reasons stated in Section 10.5.1. `initializeJess` parses the rule file into the Jess rule-engine and calls its `reset` method to initialize the working memory.

To add a new `AgentOutputWriter`, the `setAgentOutputWriter` is called with an `AgentOutputWriter` instance as argument. This object replaces the default routers set in Jess, thus all output generated by Jess is passed to the `AgentOutputWriter` object (this is discussed in the next Section). The other methods forward requests to the `Rete` object, in some cases offering a simplified interface.

### 10.5.3 Jess Output Routing

The Jess API offers methods to manipulate the I/O routers [Hil03, p. 319]. The Jess function `printout` takes as first argument the output router, e.g. `(printout t "message")` to which the string message is passed. By default this is `t`. New routers can be added with the instance method `addOutputRouter` and passing the new router name and the object itself to which the output is sent to when the router is called. For this this implementation, only the output routers were changed. An output router is `java.io.Writer` object. It was decided to replace the default routers with an own router for the master agent. The reason for this decision is simplicity: when writing Jess rules, the developer should not
Jess and JADE Integration Object

Figure 10.7: AgentOutputWriter and Strategies

worry about whether or not a rule should route the output to the master agent’s router or to the Jess default router. The new router class offers a central point to which all Jess output is sent. It then decides to which destinations it will forward the output. Jess remains unaware of how the output is further processed. The Java I/O library is based on the Decorator design pattern [Eck06, p. 915] (more on the Decorator design pattern in [GHJV95, p. 175]). Thus rather than creating one object to handle the entire I/O algorithm, functionality is shared among several objects, which can be be connected. The new Jess router for this thesis is the class AgentOutputWriter. It is wrapped inside a java.io.BufferedWriter to increase I/O operations which is decorated by a java.io.PrintWriter to format the output [Eck06, p. 930]. This configuration is done by the Jess Engine manager when its instance method setAgentOutputWriter is called. The class diagram of the AgentOutputWriter and its adherent Strategies is shown in Figure 10.7. This diagram assumes that two Strategies have already been added, a Strategy to print the output to stdout and a Strategy that sends a message to a Jess Administration agent (discussed in Section 9.4.2) that wishes to be notified when Jess generates output.

When the write instance method of an AgentOutputWriter object is called, it accepts a character array, transforms it into a string and then iterates over a set of Strategies which handle the output. This process is shown in the Sequence Diagram of Figure 10.8. The output handling of the Strategies is omitted. The motivation for choosing strategies instead of developing additional Decorators is the following: new Strategies can be added (and removed) to the Context without having to worry about the nesting of the Decorators; adding new Decorators would require to keep a pointer of the last Decorator which would in addition need a method to add a child. This would be contradictory to the Decorator’s philosophy of "Changing the skin of an object" [GHJV95, p. 179]. It would resemble changing its guts, which is the metaphor for what Strategy pattern does [GHJV95, p. 179]. To adhere to the philosophy of the Decorator would require to, every time a new Decorator is added during run time, to rewrap the current object structure and add it back to Jess. In the author’s opinion, this would increase the complexity compared to that of adding a new Strategy.

Jess Output Strategies

Jess Output Strategies extend the JessOutputStrategy abstract class, as depicted in Figure 10.7. The JadeMessageOutputStrategy is part of the Remote Jess Administration and...
10.6. Ontology Management

The management of the ontology forms an integral part of Jess management object. It is located in the Ontology Tree Manager. Because the master agent uses Jess (instead of agent behaviors) for the business intelligence, a mapping between the ontology representation used by JADE and the ontology representation used by Jess is needed. The approach chosen for this thesis is the following:

1. The ontology is modeled using the Protégé framework and with the Protégé Bean Generator Plugin (see Section 5.3.1 of Chapter 5), the ontology for JADE is generated.

2. The Ontology Tree Manager reads the ontology and creates a Composite object tree (based on the Composite design pattern [GHJV95, p. 165]) in which each node represents one class in the ontology. A node contains in addition to pointers to its parent and child nodes the name of the class it represents and its Class object.

3. The Ontology Tree Manager is the Subject to which a set of Observers are attached that are notified when the ontology changes. The Observers are referred to as

---

**Figure 10.8: Jess output routing with the AgentOutputWriter**

- **10.6. Ontology Management**
  
  The management of the ontology forms an integral part of Jess management object. It is located in the Ontology Tree Manager. Because the master agent uses Jess (instead of agent behaviors) for the business intelligence, a mapping between the ontology representation used by JADE and the ontology representation used by Jess is needed. The approach chosen for this thesis is the following:

  1. The ontology is modeled using the Protégé framework and with the Protégé Bean Generator Plugin (see Section 5.3.1 of Chapter 5), the ontology for JADE is generated.

  2. The Ontology Tree Manager reads the ontology and creates a Composite object tree (based on the Composite design pattern [GHJV95, p. 165]) in which each node represents one class in the ontology. A node contains in addition to pointers to its parent and child nodes the name of the class it represents and its Class object.

  3. The Ontology Tree Manager is the Subject to which a set of Observers are attached that are notified when the ontology changes. The Observers are referred to as

---
**Ontology Management**

In this thesis, managers do not contain the algorithms to traverse the ontology. Each manager has its own Visitor which traverses the tree and generates an output their manager can use.

---

### Ontology Composite

The ontology is represented as an object composition. Each node represents a class in the ontology. The Class Diagram for the ontology Composite is depicted in Figure 10.9. The distinction between leaves and composites is not made. A leaf is a composite with the child null. The classes in the Composite do not contain operations that operate on the nodes. These operations are packaged separately in Visitors. This is discussed in more detail in Section 10.6.2. The classes used in for the ontology Composite are the following:

- **Ontology Element** `OntologyElement` is a generic abstract class. It implements methods to manipulate the following fields:
  - `children` Is a `java.util.Vector` that contains the node’s children. If the node is a leaf, `children is null`.
  - `parent` Contains the parent of the node. For the root node, `parent is null`.
  - `javaClass` Contains the class object of the JADE ontology class it represents.
  - `name` Contains the string value of the JADE ontology’s Java class name.

The `OntologyElement` class defines the abstract method `accept`. This method accepts `OntologyVisitor` object that needs to access the node. Every subclass must implement this method separately owing to the fact that it has to call the Visitor method that is explicitly assigned to its class. An example is depicted in Figure 10.15.

- **Ontology Root Class** The `OntologyRootClass` creates an "artificial" root for the object hierarchy. As shown in [BCG07, p. 81], an ontology, as defined by the content reference model of JADE, consists of three predefined classes: `Predicate`, `Concept` and `AgentAction` as a subclass of `Concept`. They do not have a direct root class. A common root does however simplify traversing the tree. Instead of "polluting" the hierarchy
with classes that will never be used by representing the whole content reference model of JADE, this artificial root was created.

- **Concept Class** A **ConceptClass** object represents a class that is a subclass of the `jade.content.Concept` interface. This includes subclasses of `jade.content.AgentAction`.
- **Predicate Class** A **PredicateClass** object represents a class that subclasses `jade.content.Predicate`.

### 10.6.1 Ontology Tree Manager

The Ontology Tree Manager covers two crucial tasks: it generates the composite tree representing the ontology and offers an interface to access the root of the ontology and the root of the subtrees **Concept** and **Predicate**. The class diagram for the **OntologyTreeManager** and its surrounding managers (i.e. Observers) is shown in Figure 10.10. The **RemoteTreeListenerManager** class is part of the Jess Remote Administration and discussed in Section 9.4.2. This Section elaborates on the **Subject** of in the Observer pattern, thus the classes **TreeSubject** and **OntologyTreeManager**. Both classes are described, followed by detailed discussion on how the composite hierarchy is created.
- **TreeSubject** The `TreeSubject` class contains two private instance fields:

```java
class TreeSubject{
    private OntologyTreeManager aSubject;
    private TreeUser anObserver;

    public TreeSubject()
    {
        // Constructor
    }

    public void registerObserver(TreeUser observer)
    {
        anObserver = observer;
        // Register observer
    }

    public OntologyTreeManager getSubject()
    {
        return aSubject;
    }

    public void updateObservers()
    {
        // Update observers
    }

    public void processChanges()
    {
        // Process changes
    }
}
```

![Diagram](image.png)

**Figure 10.11:** A new Observer registering itself with the concrete Subject

- **Observers** A `java.util.Vector` that contains tuples (using the helper class `expertSystem.mgmt.utils.Tuple`) of `Long` and `OntologyTreeObserver`. The first value is the time when the Observer was last updated, the second value is a reference to that Observer. Observers are added by calling `registerOntologyTreeObserver`. The algorithm performed by `registerOntologyTreeObservers` is shown in Figure 10.11. It shows a new Observer that is being attached to the Subject. The participants in this example are an arbitrary Observer of the type `TreeUser` and a concrete Subject of the type `OntologyTreeManager`. The algorithm does the following:

1. A `TreeUser` object registers itself with the Subject by calling `registerOntologyTreeObserver` with its reference as an argument (step 1).
2. A new `Tuple` is created containing a reference of the Observer. The update time is set to 0.
3. The Subject calls its instance method `updateObservers` (step 2).
4. `updateObservers` calls `treeChanged` of each Observer that has an update time smaller than `lastUpdate` (step 3). `treeChanged` causes an Observer to "pulling" the Subject’s data (step 4). Next, the update time of the Observer is set to the current system time in milliseconds.

- **lastUpdate** A `Long` value stating the time in milliseconds when the ontology composite was last updated. This field is accessed and modified with the protected methods `getLastUpdate` and `setLastUpdate`.

- **OntologyTreeManager** The `OntologyTreeManager` is the concrete Subject that creates the ontology composite and offers an interface to its Observers to access the ontology. Its instance fields are:
10.6. Ontology Management

- **ontologyClasses** This instance field is a **HashMap** that stores each **OntologyElement** node in the ontology composite managed by the Ontology Tree Manager. The key of a value is the class name the node represents. This field is used during the ontology composite creation to quickly access nodes that have already been created.

- **factories** This instance field is a **HashMap**. The values are the Java class objects nodes in the ontology Composite: **OntologyRootClass**, **ConceptClass** and **OntologyRootClass**. The keys are the simple name of the classes. This field is used to simplify the creation of the ontology Composite. It is accessed by the private instance method **createOntologyElement**. It takes two string arguments: The class name of the concrete composite element and the class name of the class the node is to represent. It returns an instance of a concrete **OntologyElement** subclass.

- **root** This instance field is of the type **OntologyElement**. It stores a reference to the root of the ontology composite. This field is accessed with the instance method **getRoot**.

- **conceptRoot** This instance field is of the type **OntologyElement**. It stores a reference to the top-level of the **Concepts** subtree. This field is accessed with the instance method **getConceptRoot**.

- **predicateRoot** This instance field is of the type **OntologyElement**. It stores a reference to the top-level of the **Predicates** subtree. This field is accessed with the instance method **getPredicateRoot**.

In addition to the methods for the fields that were described above, the **OntologyTreeManager** implements the following methods:

- **setupTree** This method creates the top-level and the second-level of the ontology composite. Every new node is attached to either one of them, excluding the root node.

- **buildSimpleTree** This method reads the ontology classes from the file system and adds the classes to a data structure from which the ontology composite can be generated.

- **buildVisitorWorthyTree** This method generates the ontology composite that will be used by the Visitors of its Observers.

- **updateTree** This method causes the Ontology Tree Manager to generate the ontology composite and notify its Observers that the ontology has changed by executing **updateObservers**.

The composite generation is discussed in detail in the next Section.

**Creating the Jade Ontology Composite**

An ontology composite managed by the Ontology Tree Manager is created in three stages:

1. The top of the tree is created. The top consists of the artificial root of the type **OntologyRootClass** and its two children **PredicateClass** and **ConceptClass**. This structure has to be generated in a separate step from the rest of the ontology because it cannot be generated from the information available on the file system and from the ontology instance.
2. The ontology Java classes that were created by Protégé are loaded from the file system and their names and class objects added to two Vectors: one Vector containing the name of the class and one Vector the Class objects. A class has its name and Class object at the same position in both Vectors. The classes are sorted to be in the following order: The class at position $p$ in the Vector has to be at a higher level in the tree than the class at the position $p + 1$, if $p + 1$ is below $p$ in the ontology class hierarchy. This permits to create the ontology composite top-down and knowing that, if the class has a parent, it has already been processed and can be attached to it.

Basically the sorting algorithm works as follows: It uses two vectors: one vector that contains the unsorted classes read from the file system and one vector that will contain the classes in the order defined above. In the implementation of this algorithm with the method buildSimpleTree, shown in Code extract D.2, these are the Vectors domainOntologyContent for the unsorted classes and stringResult (respectively classResult for the Class objects) for the sorted classes. The algorithm loops over the vector that contains the unsorted classes and executes the following steps until all classes have been sorted and the vector containing the unsorted classes is empty:

It removes the first element of the vector containing the unsorted classes and stores it in the variable current. If the current element’s parent is not in the in the vector containing the unsorted classes, it is added to vector that contains the sorted classes and the next iteration of the loop is started. Otherwise, the current class is placed at the position of its parent in the vector containing the unsorted classes and current is set the parent (of the previously current class) for which the next iteration is started. The implementation of this algorithm is explained in detail in Appendix D, Section D.2.

![Figure 10.12: Simple class hierarchy for the sorting example](image)

<table>
<thead>
<tr>
<th>STEP</th>
<th>DOC</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$C_3$</td>
<td>$C_4$</td>
</tr>
<tr>
<td>1</td>
<td>$C_1$</td>
<td>$C_4$</td>
</tr>
<tr>
<td>2</td>
<td>$C_4$</td>
<td>$C_3$</td>
</tr>
<tr>
<td>3</td>
<td>$C_2$</td>
<td>$C_4$</td>
</tr>
<tr>
<td>4</td>
<td>$C_3$</td>
<td>$C_5$</td>
</tr>
<tr>
<td>5</td>
<td>$C_5$</td>
<td>$C_4$</td>
</tr>
<tr>
<td>6</td>
<td>$C_4$</td>
<td>$C_1$</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>$C_1$</td>
</tr>
</tbody>
</table>

Table 10.1: Steps performed by the `buildSimpleTree` of Code extract D.2 for the class hierarchy in Figure 10.12
An example of how this sort works is given for the class hierarchy shown in Figure 10.12. Table 10.1 contains each step executed by the algorithm. Step 0 shows the order of the classes after they were loaded from the file system. Each row shows the state of the Vectors after a iteration is performed. The variable current is represented by row C. The arrays are domainOntologyContent, in column DOC and the final result (Vectors classResult and stringResult), in column R. The entries that are underlined in column DOC and C are the entries that were swapped.

3. The Vector array created in second step is used to create the ontology Composite. The Composite structure of the ontology is created top down. Each child node is attached to its parents node. This is shown in Figures 10.13(a) to 10.13(e) for the example given in Figure 10.12. The example shows the construction of the tree, top down, from left to right. The reader shall note that the order of nodes at the same level is not of relevance. The implementation of this algorithm is described in details in Appendix D, Section D.3.

Figure 10.13: Steps performed to construct a composite structure of the ontology of Figure 10.12

10.6.2 Ontology Visitors

Figure 10.14: Ontology Composite Visitor Class Diagram
An Observer of the ontology composite does not access the ontology Composite itself, instead, it uses a Visitor which "knows" how the tree is traversed to collect the information the Observer needs. An example in Figure 10.15 is given with a OntologyClassManager Observer that uses its ClassCollector Visitor to traverse the ontology composite containing a parent node with two child nodes. The Visitor and the nodes collaborate to traverse the composite. The manager accesses the ontology composite when the Ontology Tree Manager calls its treeChanged method. A OntologyClassManager is only interested in Concepts, hence it calls getConceptRoot of the OntologyTreeManager. When the Visitor has finished, the Ontology Class Manager accesses the Visitor’s results by calling getResults.

The Class Diagram depicted in Figure 10.14 shows the OntologyVisitor and the concrete Visitor implementations that were developed for this thesis. The LispingTreeVisitor class is part of the Jess Remote Administration and discussed in Section 9.4.2. The remaining Visitors are described in the rest of this Section.

- **OntologyVisitor** OntologyVisitor is a generic interface. It defines an operation signature for each class in the Composite structure introduced in Section 10.6 and getResults that returns a result of the type of the parameter the subclass specifies.

- **Jade2JessVisitor** The Jade2JessVisitor maps the ontology to a Jess template hierarchy for Jess and JADE to be able to interoperate seamlessly. An example template for the FindMe agent action is shown in Code extract 10.7.

```
(deftemplate FindMe extends AgentAction (declare (from class ontology . matrix . FindMe)));
```

Code extract 10.7: Jess template for FindMe
10.6. Ontology Management

This entry only specifies the header of the template. Jess is able to automatically determine the slots of the template based on the setters and getters methods of the class specified with the from-class declaration.

- **ClassCollectorVisitor** A ClassCollectorVisitor traverses the Concept subtree and collects the name and the class object of each node.

### 10.6.3 Ontology Dependant Managers

As shown in Figure 10.4, a set of managers depend on the ontology to perform their work. These managers and their superclass and interface are discussed in this Section.

- **OntologyTreeObserver** The OntologyTreeObserver interface is the type of Observer known to a TreeSubject. It defines one abstract method for its subclasses to implement:
  - `treeChanged` This method is called by a TreeSubject object when its ontology is changed.

- **TreeUser** The TreeUser abstract generic class has two instance fields:
  - `visitor` This instance field is of the generic type `OntologyVisitor<V>`. It is set during construction time by passing a concrete `OntologyVisitor` as constructor argument.
  - `treebeard` This instance field holds a reference to the `OntologyTreeManager` object that is the Subject of the Observer.

It implements the following method that are to be used by its subclasses:

- `getOntologyTreeManager` This method returns the `OntologyTreeManager` of the Observer object.

- `getVisitor` This method returns the concrete `OntologyVisitor` of the Observer object.

- **TreeUser** In addition to setting the values of the instance fields, the constructor registers the Observer with the `OntologyTreeManager`.

- **Jade2JessOntologyManager** The Jade2JessOntologyManager assures interoperability between JADE and Jess by mapping the ontology used by JADE to Jess templates. This permits for both applications to seamlessly exchange information objects. The JADE to Jess Ontology Manager uses a Jade2JessVisitor that traverses the tree and returns a string with the template headers. Changes to the template structure of Jess may have fundamental consequences for the working memory of Jess. The Jade2JessOntologyManager therefore restarts Jess when the ontology it changed. However the first time this manager runs, this case has to be handled specially. As shown in Code extract 10.6 in Section 10.5.1, `startJess` is first called when JessCNC is instantiated. At this stage, the ontology composite has not yet been created by the Ontology Tree Manager. When the Ontology Tree Manager is called, it will cause all its Observers to update their representation of the ontology, causing the Jade2JessVisitor to restart JessCNC and so forth. JessCNC and Jade2JessVisitor thus need a boolean value to coordinate the initialization. The `treeChanged` method of the Jade2JessVisitor class is shown in Code extract 10.8.
public void treeChanged() {
    if (!this.firstTime)
        this.jesscnc.stopJess();
    this.getOntologyTreeManager().getRoot().accept(this.getVisitor());
    this.loadJadeOntologyIntoJess();
    if (!this.firstTime)
        this.jesscnc.startJess();
}

Code extract 10.8: treeChanged instance method of the Jade2JessVisitor class

- OntologyClassManager The OntologyClassManager provides the Jess Function Manager with access to class objects of the classes in the ontology. Further, it maps agent actions to behaviors, based on the rules discussed in Section 10.3.1. This class contains the following instance fields:
  
  - ontologyConceptClassManager This field is a HashMap that contains the class objects of the Concept classes in the ontology. This hash is created by the manager’s Visitor. Single elements of this hash are accessed by passing the key of an element as argument to the instance method getOntologyConcept. A list of all keys of this hash is fetched by calling the instance method getOntologyConceptNames.
  
  - abmap The abmap field is a HashMap that contains tuples of (agent action, behavior). A key of such a tuple is the class name of the agent action. A (agent action, behavior) tuple is fetched through the instance method getActionInitiatorBehaviourTuple. Currently the values of this hash are hard coded in the constructor, to serve as a proof of concept.

10.7 Jess Functions for JADE and Function Management

private void addInitialFunctionsToJess() {
    this.jfm.registerJadeJessFunction(new CommandAgent(this.masterAgent, this.jfm, this.classReg));
    this.jfm.registerJadeJessFunction(new ReturnFactToCurrentBehaviour(this.masterAgent, this.jfm, this.classReg));
}

Code extract 10.9: Registering JADE aware Jess functions

A set of Jess functions to access JADE seamlessly from within Jess were developed for this thesis. They are managed by an instance of the JessFunctionManager class. Jess functions are registered with the Jess Function Manager when the JessCNC object of the master agent is instantiated, as shown in Code extract 10.9. These functions are coupled with Jessable behaviors, discussed in Section 10.7.1. The Jess Function Manager maintains a pointer to every Jess function. Additionally, it registers each Jess function with the Jess Engine Manager that registers the functions with Jess. The Class Diagram for the Jess functions and the Jess Function Manager is shown in Figure 10.16. This Section
discusses the Jess functions developed for this thesis and the Jess Function Manager. The integration with Jessable behaviors is discussed in Section 10.7.1.

**JessFunctionsManager** This class provides the means of communication between Jess functions that communicate with JADE and Jessable behaviors. Functions and behaviors that require to interact register with the **JessFunctionsManager**. The **JessFunctionsManager** class defines the following instance fields:

- **jessableBehaviours** This field is a HashMap that contains the Jessable behaviors that are known to the **JessFunctionsManager**. The key of a Jessable behavior is an integer generated with the behavior’s `hashCode` instance method which provides a unique value for an object [Fla05, p. 133]. This value is used by a behavior to identify itself with the **JessFunctionsManager**. A behavior registers itself by calling `addJessableBehaviour`. It is removed with `removeJessableBehaviour` and the hash of the behavior as argument. The current behavior is accessed by calling `getCurrentBehaviour`. If no behavior is registered a `NoCurrentBehaviourRegistered` exception is raised.

- **currentBehaviour** This integer holds the identification number of the Jessable behavior that is currently running. If no behavior is running, the value is −1. A behavior sets itself as current behavior by calling `setCurrentBehaviour` and removes itself as current behavior by calling `removeAsCurrentBehaviour`.

Figure 10.16: JADE aware Jess function and Jess Function Manager
• **jessFunctions** This field is a `Vector<JadeJessFunction>` that contains functions that communicate with JADE. They are of the type `JadeJessFunction`. A function is added by calling `registerJadeJessFunction`.

• **jesscnc** This field contains a reference to the master agent’s `JessCNC` object. Its value is set when the manager is instantiated.

A JADE aware Jess function can require to send messages to agents in the master’s environment. To create a message for the proper FIPA protocol, it needs access to the instance of a `CommunicationConfigFactory`. Owing to the fact that it cannot be known at compile time on which FIPA protocol the conversation will be based, the FIPA protocol has to be specified at run time. The approach chosen for this implementation is to let the current behavior set the concrete `CommunicationConfigFactory`. This approach has the advantage that Jess does not have to be aware of communication related issues. The disadvantage is however that only one FIPA protocol is supported per Jessable behavior. To set the `CommunicationConfigFactory` a behavior calls the following method:

• **setCommunicationConfigFactory** This method sets the `setCommunicationConfigFactory` for each Jess function object known to the Jess Function Manager.

**JadeJessFunction** `JadeJessFunction` is an abstract class from which all JADE aware functions inherit. It implements the `jess.Userfunction` interface to be recognized by Jess as a Jess user function. It contains instance fields and methods required by its subclasses to communicate with the current behavior and other agents. It defines the following instance fields:

• **myAgent** This instance field contains a reference to the master agent on which Jess runs. It can be accessed by a subclass through the protected method `getAgent`.

• **jessFunctionManager** This instance fields contains reference to a `JessFunctionManager` with which the Jess function is registered. Subclasses can access it by calling the protected method `getJessFunctionManager`.

• **ocr** This instance field holds a reference to the `JessCNC`’s `OntologyClassManager` instance. Subclasses access it by calling the protected method `getOntologyClassManager`.

• **commConfigFactory** This instance field holds a reference to the concrete `OntologyClassManager` class set by the Jessable behavior during run-time. Subclasses access this field through the protected method `getCommunicationConfigFactory`.

A goal of the integration approach chosen for this thesis is to keep Jess "unaware" of interactions with the "outside world", i.e. Java classes that depend on it. Thus, when a fact is asserted that is to be used outside Jess, it should not be asserted any differently from facts that are only used internally by Jess. However natively only shadow facts are connected to POJOs. Thus, if a fact is asserted that is meant to be used by the surrounding Java application, a shadow fact has to be created by calling `definstance` (or first instantiated as a normal Java object and then added to the working memory by calling `add`). To avoid this, the `JadeJessFunction` class contains a protected instance method that converts a Jess fact into a POJO of the same type. This is possible because Jess and JADE both use the same ontology. The convention is handled by the following method:

**jessFactToJadeOntologyInstance** `jessFactToJadeOntologyInstance` takes a `Fact` and `Context` instance as argument and returns a `Concept` of the same type as the fact with the
public Value call(ValueVector vv, Context context) throws JessException {
    if (vv.size() != 3)
        throw new JessException(this.getName(),
            "Wrong number of arguments", vv.size() - 1);
    AgentAction action = (AgentAction) this.jessFactToJadeOntologyInstance(vv.get(1).factValue(context), context);
    String reciever = vv.get(2).stringValue(context);
    Tuple<Class<AgentAction>, Class<Behaviour>> abt = this.getOntologyClassManager().
        getActionInitiatorBehaviourTuple(action.getClass().getName());
    try {
        AID rec = new AID(reciever, AID.ISGUID);
        ACLMessage msg = this.getCommunicationConfigFactory().createFIPARequestMessage(rec, action);
        Constructor<Behaviour> c = abt.getSecondElement().getDeclaredConstructor(Agent.class, ACLMessage.class);
        c.setAccessible(true);
        Behaviour b = c.newInstance(this.getAgent(), msg);
        this.getAgent().addBehaviour(b);
    } catch (InstantiationException e) {
        // Exception handling, code omitted
    }
    return Funcall.TRUE;
}

Code extract 10.10: Algorithm for the command-agent Jess user function

fact’s content. The reader shall note that this method will not work for facts that are of the type Predicate.

CommandAgent An instance of this class is used to send an agent action to an agent. It takes as argument the agent action and the String value of the AID of the receiver; e.g. (command-agent ?a ?r) where ?a is a variable containing an AgentAction and ?r a variable containing the String value of the AID of the receiver. The implementation of the command-agent function is shown in Code extract 10.10. This function takes precisely two arguments (the first value of the vector is the function itself), otherwise a JessException is thrown (lines 2–4). Next, the fact is converted to an AgentAction of the ontology used by the agents in the multi-agent system. This is achieved by calling jessFactToJadeOntologyInstance and passing the fact (first argument) and context as an argument (line 5). At line 9, the receiver is extracted and at line 10 the Ontology Class Manager is called to get the behavior that matches the given agent action. As stated in Section 10.3.1, every agent action corresponds to a conversation, thus it is not necessary to specify which behavior is used when calling command-agent. In lines 9–14, an AID object containing the address of the receiver is created. It is passed with the agent action as a constructor argument to the ACLMessage created by calling the Communication Configuration Factory set by the behavior that is currently running. Using the Java reflection API, a new behavior instance is created from the class object in the tuple that was returned from the Ontology Class Manager. The current agent and the ACLMessage is passed as constructor argument of the behavior (lines 11–12). This behavior is added to the agent’s behavior list to be executed (line 14). This function returns a Funcall.TRUE since there is no other data to return.

ReturnFactToCurrentBehaviour This class implements the (return-fact-to-current-behaviour) Jess function. With this functions, facts (currently only Concepts) can be returned to the behavior that is running by calling (return-fact-to-current-behaviour ?c), where ?c is a variable
10.7. Jess Functions for JADE and Function Management

```java
public Value call(ValueVector vv, Context context) throws JessException {
    try {
        Concept concept = this.jessFactToJadeOntologyInstance(vv.get(1).factValue(context), context);
        this.getJesseFunctionManager().getCurrentBehaviour().addOntologyConcept(concept);
    } catch (NoCurrentBehaviourRegistered e) {
        throw new JessException(this.getName(), "No Jessable Behaviour registered", e.getCause());
    }
    return Funcall.TRUE;
}
```

Code extract 10.11: Algorithm for the return-fact-to-current-behaviour Jess user function containing a `Concept` instance that is to be returned to the active Jessable behavior. While facts can of course always be retrieved by using the query interface of Jess, this function offers an easy alternative to return facts that are intermediate results without having to store these facts separately or interrupt Jess to let the current behavior run a query. The algorithm for this function is shown in Code extract 10.11. First the fact is converted to its corresponding type in the ontology in line 3 and in line 4 the concept is added to its behavior. When Jess has finished executing, the current behavior can fetch the results by calling its instance method `getConcepts`.

### 10.7.1 Jess Integrated Behaviors

A Jess integrated behavior, known a Jessable Behavior in this thesis, is a behavior that is tightly coupled with the Jess Function Manager. The behavior is not solely a service user of JessCNC and its managers. Some objects are aware of the Jessable behavior that is running. A Jessable behavior extends the `JessableBehaviour` interface. For this thesis, the abstract Jessable behavior `JessableRequestResponderTemplate` for the participant in a FIPA-Request-Protocol was developed. It inherits from the `RequestResponderTemplate` class. The Class Diagram is shown in Figure 10.17. Each participant shown on Figure 10.17...
is described, followed by an detailed description of the interaction between a Jessable behavior and JADE aware functions.

**Jessable Classes**

- **JessableBehaviour** The JessableBehaviour interface specifies the following method signatures that its subclasses have to implement:
  - `registerWithJessFunctionManager` This method is to register the behavior with the JessFunctionManager object of the master agent.
  - `unregisterWithJessFunctionManager` This method is to remove the behavior from the list of known Jessable behaviors of the JessFunctionManager object.
  - `setAsCurrent` This method is to notify the Jess Function Manager of the current behavior.
  - `removeAsCurrent` This method is to notify the Jess Function Manager.
  - `addOntologyConcept` This method is to add a Concept object to the behavior.
  - `setCommunicationConfigFactory` This method is to return the concrete CommunicationConfigFactory.

- **JessableRequestResponderTemplate** This class extends the RequestResponderTemplate (see Section 10.3.1) and adds the additional functionalities required by a Jessable behavior. The source code is shown in Code extract 10.12. JessableRequestResponderTemplate has three instance fields:
  - `jessCNC` This instance field holds a reference to the JessCNC object of the master agent. It is assigned during construction (line 14).
  - `concepts` This HashMap contains concepts added by a JADE aware Jess function. When a concept is added, the class name for the concept is accessed and set as key for that entry.
  - `toClear` This instance field is set to true when the current behavior accesses the concepts that were added (line 8–9) causing the Vector to be emptied before new Concepts are added again (line 48–50).

This class implements the hook operations `doAtBeginning`, `doAtEnd` and `onEnd` and the primitive operation `registerWithJessFunctionManager`:

- `doAtBeginning` This method is called every time the behavior runs. It calls its method `setAsCurrent` which notifies the Jess Function Manager that it is the current behavior (line 34). It then informs the Jess Function Manager of the current Communication Configuration Factory (line 19).

- `doAtEnd` This method is called when the behavior finished handling the request. It informs the Jess Function Manager that is no longer the current behavior. The Communication Configuration Factory is not removed, it will simply be updated by the next Jessable behavior.

- `onEnd` This is a hook operation of the Behavior class. It is for subclasses to add termination code for the behavior. This implementation calls the super class implementation of the same method and deregisters itself with the Jess Function Manager (lines 42–44).
- `registerWithJessFunctionManager` This method registers the behavior with the Jess Function Manager. The behavior passes its hash code and its reference as argument to Jess Function Manager’s `addJessableBehaviour` instance method (line 26). `registerWithJessFunctionManager` is called when the behavior is instantiated (line 15).

```java
public abstract class JessableRequestResponderTemplate extends RequestResponderTemplate implements JessableBehaviour {
    protected JessCNC jessCNC = null;
    private Vector<Concept> concepts = new Vector<Concept>();
    private boolean toClear = false;

    protected Vector<Concept> getConcepts () {
        this.toClear = true;
        return this.concepts;
    }

    public JessableRequestResponderTemplate(Agent a, MessageTemplate mt) {
        super(a, mt);
        this.jessCNC = ((JessableAgent)this.myAgent).getJessCNC();
        this.registerWithJessFunctionManager();
    }

    protected void doAtBeginning () {
        this.setAsCurrent();
        this.jessCNC.getJessFunctionManager().setCommunicationConfigFactory(this.
            setCommunicationConfigFactory());
    }

    protected void doAtEnd () { this.removeAsCurrent(); }

    public void registerWithJessFunctionManager () {
        this.jessCNC.getJessFunctionManager().addJessableBehaviour(this.hashCode(), this);
    }

    public void removeAsCurrent () {
        this.jessCNC.getJessFunctionManager().removeAsCurrentBehaviour();
    }

    public void setAsCurrent () {
        this.jessCNC.getJessFunctionManager().setCurrentBehaviour(this.hashCode());
    }

    public void unregisterWithJessFunctionManager () {
        this.jessCNC.getJessFunctionManager().removeJessableBehaviour(this.hashCode());
    }

    public int onEnd () {
        int tmp = super.onEnd();
        this.unregisterWithJessFunctionManager();
        return tmp;
    }

    public void addOntologyConcept(Concept concept) {
        if (this.toClear) {
            this.concepts.clear();
            this.toClear = false;
        }
        this.concepts.add(concept);
    }
}
```

Code extract 10.12: FIPA-Request abstract Responder behavior with Jess support
Interaction Between a Jessable Behavior and JADE Aware Functions

An example was created to illustrate the interaction between a Jessable behavior and JADE aware Jess functions. Due to its size, it is divided in the Sequence Diagrams of Figure 10.18, Figure 10.19 and Figure 10.19. Figure 10.18 shows the beginning and the end of the example. The example demonstrates the following: A concrete Jessable behavior extends the `JessableRequestResponderTemplate` class. It first runs Jess which at some point fires a rule that calls the `return-fact-to-current-behaviour` function and another rule that calls `command-agent`. It is assumed that the system is set up, thus all objects that require registration have already done so.
10.8 A Brief Discussion of an Oracle Agent Without Rule Engine

To verify the assumption that using a rule-engine instead of hard coding the business logic in Java, a Oracle agent without Jess was developed, however only with the basic functionalities. The Oracle agent that does not use Jess has the same ontology, protocol and
content language as the Oracle agent that uses Jess, hence the interface remains the same, the slave agents do not have to be modified. The implementation of the `handleRequest` of the `CommandMeResponder` behavior class is shown in Code extract 10.13. The reader shall note that this is based on a older version of this project but the quintessence remain the same. The implementation for this rather short example seems to be simpler to be implemented in Java than in Jess. This however quickly changes when one considers the possible effects of adding new logic to the agent. Jess handles the order in which rules execute, with Java, the programmer will have to pay close attention where the next `if` or `while` statement will be added. The author believes that Jess is a far more superior tool to manage large amount of rules without loosing complete oversight. Additionally, the Java code will have to contain logic to address the working memory management, possibility by using a database. Thus the logic will be mixed with data handling code that adds nothing to the business logic. This also applies for example if one wishes to add logging functionality. Jess has an elegant event handling interface for object that could for example log the activities of Jess. And, as shown with the Jess Administration agent (Chapter 9), Jess rules can be added and removed on th fly. A Java implementation would require to stop the agent, update the class and start the agent again.

```java
public class CommandMeResponder extends AchieveREResponder {
    protected ACLMessage handleRequest(ACLMessage request)
        throws NotUnderstoodException, RefuseException {
        ACLMessage reply = null;
        this.oa = (agents.OracleAgent) this.myAgent;
        try {
            reply = request.createReply();
            Action a = (Action) oa.getContentManager().extractContent(request);
            CommandMe commandMe = (CommandMe) a.getAction();
            MatrixCombatAction mca = null;
            if (!this.isLockedFor(commandMe.getSecret())) {
                throw new Unauthorised();
            }
            ontology.matrix.CombatAgent slave = (CombatAgent) this.oa.getAgent(commandMe.getSlave());
            double requesterHealth = slave.getHealthstate();
            String secret = null;
            if (requesterHealth > 0) {
                double health = 0;
                ontology.matrix.CombatAgent victim = this.getNextTarget((CombatAgent) this.oa.getAgent(slave.getAgentName()));
                if (victim == null)
                    throw new Unauthorised();
                Attack attack = new Attack();
                health = health - (slave.getPower() + Math.random() * 5.0);
                victim.setHealthstate((float) health);
                attack.setAttacker(slave.getAgentName());
                attack.setVictim(victim.getAgentName());
                mca = attack;
                secret = this.createSecret();
            } else if (requesterHealth <= 0) {
                Die die = new Die();
                die.setVictim(slave.getAgentName());
                die.setAttacker(request.getSender().getName());
                this.oa.unregisterAgent(slave.getAgentName());
                mca = die;
                secret = ""; // reset lock, the agent is going to die, there is no more communication between agents
            }
            this.setLockFor(secret);
            mca.setSecret(secret);
            CommunicationConfig cf = MatrixCommunication.getInstance();
            ACLMessage orderMsg = cf.createFIPARequestMessage(this.myAgent, request.getSender().getName(), mca);
            this.registerPrepareResultNotification(new CommandSlaveAgent(oa, orderMsg, this.getDataStore()));
        }
    }
```
10.8. A Brief Discussion of an Oracle Agent Without Rule Engine

```java
ContentElementList cel = new ContentElementList();
cel.add(a);
reply.setPerformative(ACLMessage.AGREE);
oa.getContentManager().fillContent(reply, cel);
catch (UngroundedException e) {
    e.printStackTrace();
} catch (CodecException e) {
    e.printStackTrace();
} catch (OntologyException e) {
    e.printStackTrace();
}
return reply;
```

Code extract 10.13: ChangeMeResponder without Jess rule-engine
11

Using the J2J Tool Kit to Add Jess to MediMAS

11.1 Introduction .............................................. 128
11.2 Creating a Jess Administration Agent ..................... 129
11.3 Modifying the MediMAS Ontology ......................... 130
   11.3.1 Create a Communication Configuration Factory for the Medi-
          MAS ontology ........................................ 131
11.4 Adding Jess Support to MediMAS ............................ 131
   11.4.1 Extending JessCNC ................................. 131
   11.4.2 Making the MediMAS Alert Manager Agent Jessable .... 131
   11.4.3 Creating a Jessable Behavior to Handle the Alert Notification 132
   11.4.4 Implementing the Alert Notification Logic with Jess Rules . 135
   11.4.5 Adding a JADE Aware Jess User Function ............ 136
11.5 Changing the Behavior at Run Time ......................... 137
   11.5.1 Sending Alerts To the Head of Laboratory .......... 137
   11.5.2 Reacting to Laboratory Results with H5N1 ........... 137
11.6 Conclusion .............................................. 139

11.1 Introduction

In the introduction of this thesis it was stated that this thesis was initiated to make Medi-
MAS more adaptive to changing requirements of the healthcare workflows it supports. It was argued that using a rule-based system to handle the business logic was a suitable solution. Processes can be described in declarative way instead of writing procedures. Furthermore, if the rule-based system engine is an interpreter, rules can be changed on the fly. In the current Part, a tool kit has been developed that integrates Jess and JADE. This tool kit, called the J2J tool kit, is now applied to MediMAS: MediMAS contains a set of alerts. Alerts are classified, among other criteria, according to their criticality and their degree of urgency. Alerts that are critical or urgent (or both) are monitored by
MediMAS and, if they are not handled by their dedicated receiver within a given time period, sent again or escalated. This is depicted in the Event-driven Process Chains of Figure 3.3, Figure 3.4, Figure 3.5 and Figure 3.6 in Chapter 3. This logic is currently written in Java in the AgentHandleAlertActionsBehaviour class. The algorithm that conducts these verifications is shown in Code extract 11.1.

```java
protected void handleAlert(Concept concept) {
    AlertAction alert = (AlertAction)concept;
    long lastalerttime = alert.getLastAlertTime().getTime();
    long currenttime = new Date().getTime();

    int intval = 1000 * HospitalConstants.NORMAL_PENDINGTIME;
    if (alert.getCritical() == true) intval = 1000 * HospitalConstants.CRITICAL_PENDINGTIME;
    else if (alert.getUrgent() == true) intval = 1000 * HospitalConstants.URGENT_PENDINGTIME;
    if ((currenttime - lastalerttime) >= intval) {
        // Update alert infos in the database
        this.updateAlert(alert);
        // Send alert to requester / (and) labdirector
        this.sendAlert(alert);
    }
}
```

Code extract 11.1: The handleAlert method of the AgentHandleAlertActionsBehaviour class

Modifying the logic of this behavior in run-time is currently not possible. It was therefore decided to use Jess to handle the decision making for the alert management. In addition, a new JADE aware Jess user function was added. This new Jess user function permits to modify the time intervals in which the behavior that processes the alert action, runs. This Chapter explains how the J2J tool kit is added to MediMAS. It is structured as follows: Section 11.2 discusses the implementation of the remote Jess Administration agent developed for MediMAS. In Section 11.3, the slight changes that were made to the MediMAS ontology, are commented. Section 11.4 explains the approach chosen to apply the J2J tool kit to MediMAS followed by an example in Section 11.5 of how the alert notification of the Alert Manager agent can be changed during run time with Jess. In Appendix C, the libraries that have to be known to the MediMAS project, are listed.

### 11.2 Creating a Jess Administration Agent

To write a concrete Jess Administration agent, the AbstractJessAdminAgent abstract class has to be extended and its abstract instance method registerWithMasterAgentAndLoadBehav-
Modifying the MediMAS Ontology

The remote Jess Administration agent class for MediMAS is JessMedicalAdmin, as depicted in Figure 11.1. The implementation of the registerWithMasterAgentAndLoadBehaviors instance method must first lookup the master agent that is to be administered remotely and then, as shown in Code extract 11.2 call the methods setMaster and loadDefaultBehaviors. In this example, the AID of the master agent that was queried, is stored in the variable rba.

```java
1 this.setMaster(rba);
2 this.loadDefaultBehaviors();
```

Code extract 11.2: The instructions to set the address of the master and load the behaviors needed to communicate with the master agent

### 11.3 Modifying the MediMAS Ontology

The MediMAS ontology had to be slightly modified for this Chapter. The affected classes of the MediMAS ontology, AlertAction and Time, are shown in Figure 11.2. Time, with the slot `time`, was added to represent a time in Jess. This is required to determine whether or not an AlertAction is due. Additionally, a String slot icd10 (more on International Classification of Diseases 10 (ICD-10) in [21]) was added to the AlertAction class. This change was not mandatory and is currently not supported by MediMAS. It is however useful to demonstrate how MediMAS could also react to certain diseases in a specific way.
11.3.1 Create a Communication Configuration Factory for the MediMAS ontology

A Communication Configuration Factory can be added for the MediMAS ontology. Since MediMAS already creates message in its own way, it is, strictly speaking, only needed for JessJadeFunctions that want to send messages. Such functions are not used for MediMAS. Thus, a Communication Configuration Factory was not developed.

11.4 Adding Jess Support to MediMAS

This Section describes in detail how the J2J tool kit was added to MediMAS. First the JessCNC class is extended to add a new JADE aware Jess user function to the Jess engine. This is explained in Section 11.4.1. In Section 11.4.2, the new Alert Manager agent with Jess support is discussed and Section 11.4.3 elaborates on the new behavior that uses Jess to decide how to treat alerts. Section 11.4.4 explains the Jess rules that handle the alert management logic. Section 11.4.5 presents the JADE aware Jess user function developed for MediMAS to set the time intervals between the alert notification.

11.4.1 Extending JessCNC

Because a new JADE aware Jess user function was required, JessCNC had to be extended and the instance method addInitialFunctionsToJess overwritten to add the newly developed JADE aware Jess user function to the Jess engine. The new class is called MediMAS-JessCNC (see Figure 11.3). Its addInitialFunctionsToJess is shown in Code extract 11.3. It first calls the method addInitialFunctionsToJess of its parent class to add the default JADE aware Jess user functions (line 2) and then adds the class ChangeAlertCheckTime. This class implements the Jess user function change-alert-check-time, described in Section 11.4.5.

```java
protected void addInitialFunctionsToJess() {
    super.addInitialFunctionsToJess();
    this.getJessFunctionManager().registerJadeJessFunction(
        new ChangeAlertCheckTime(this.masterAgent, this.getJessFunctionManager(),
        this.getOntologyClassManager()));
}
```

Code extract 11.3: Extending JessCNC to add a new JADE aware Jess user function

11.4.2 Making the MediMAS Alert Manager Agent Jessable

Instead of changing the current AlertManagerAgent class, it is extended with the JessableAlertManagerAgent class that implements the JessableAgent interface of the J2J tool kit. The class hierarchy is depicted in Figure 11.3 (the parent classes of the AlertManagerAgent class and the interface of the JessCNC class are omitted). The source code is shown in Code extract 11.4. It contains one instance field (line 2):

- jesscnc This instance field holds the JessCNC object of the master agent.

This class implements two instance methods:

- getJessCNC This method provides access to the master agent’s JessCNC object.
11.4. Adding Jess Support to MediMAS

Figure 11.3: The class hierarchy of the JessableAlertManagerAgent class

- **setup** This method configures the agent. It first calls the `setup` method of its parent class (line 8) and then instantiates a `MediMASJessCNC` object (line 9). In line 10, the `configureMasterForJessRemoteAdministration` method of the agent’s `JessCNC` object is called. This method configures the agent for the remote Jess Administration.

```java
public class JessableAlertManagerAgent extends AlertManagerAgent implements JessableAgent {
    private JessCNC jesscnc;

    public JessCNC getJessCNC() {
        return this.jesscnc;
    }

    public void setup() {
        super.setup();
        this.jesscnc = new MediMASJessCNC(this);
        this.jesscnc.configureMasterForJessRemoteAdministration();
    }
}
```

Next, the `OntologyTreeManager` class (discussed in Section 10.6) has to be edited to read the proper ontology: The static `String` field `DOMAIN_ONTOLOGY` must contain the relative path of the directory that stores the classes of the ontology. The static `Ontology` field `ONTOLOGY` must contain the reference to the current ontology object. This is shown in Code extract 11.5.

```java
private static String DOMAIN_ONTOLOGY = "hospital/ontology";
private static Ontology ONTOLOGY = HospitalOntology.getInstance();
```

The reader shall note that the Ontology Tree Manager only works for ontologies created with Protégé and if all the class files are in the same directory. Additionally, only single inheritance is supported.

11.4.3 Creating a Jessable Behavior to Handle the Alert Notification

The `AgentHandleAlertActionsBehaviour` class that currently manages the alert notification, is extended and the method that hard codes the alert notification logic (shown in Code
11.4. Adding Jess Support to MediMAS

extract 11.1), is overwritten. It was decided to make this behavior Jessable. The class hierarchy of the JessAgentHandleAlertActionsBehaviour class is shown in Figure 11.4. The parent classes of the AgentHandleAlertActionsBehaviour class are omitted.

Jess can be used from any sort of behavior, it need not be Jessable. However, a Jessable behavior can communicate with JadeJessFunctions. This feature is useful for the implementation of the alert notification handling with Jess. When a rule responds to an alert, it updates certain fields of the alerts it responded to. Instead of having to run a query on the working memory to find the facts that were changed, a rule can simply call the JADE aware Jess user function return-fact-to-current-behaviour to return the modified facts to the current behavior. In the author’s opinion, this is easier than running a query on the working memory after Jess has finished executing.

Implementing the JessAgentHandleAlertActionsBehaviour class

```java
public class JessAgentHandleAlertActionsBehaviour extends AgentHandleAlertActionsBehaviour {
    private JessCNC jesscnc = ((JessableAgent) this.myAgent).getJessCNC();
    private Vector<Concept> returnedConcepts = new Vector<Concept>();
    private WorkingMemoryMarker marker;

    public JessAgentHandleAlertActionsBehaviour(Agent agent, Vector<?> concepts) {
        super(agent, concepts);
        this.registerWithJessFunctionManager();
        this.marker = this.jesscnc.getJessEngineManager().mark();
    }

    protected void handleAlert(Concept concept) {
        this.setAsCurrent();
        AlertAction alert = (AlertAction)concept;
        Time time = new Time();
        time.setTime(new Date());
        this.jesscnc.getJessEngineManager().resetToMark(this.marker);
        this.jesscnc.getJessEngineManager().addFactToJess(alert);
        this.jesscnc.getJessEngineManager().addFactToJess(time);
        this.jesscnc.getJessEngineManager().setModuleFocus("IMPORTANT");
        this.jesscnc.getJessEngineManager().runJess();
        AlertAction aa = null;
        for (Concept c: this.returnedConcepts) {
            aa = (AlertAction) c;
            this.updateAlert(aa);
        }
    }
}
```
11.4. Adding Jess Support to MediMAS

The implementation of the `JessAgentHandleAlertActionsBehaviour` behavior is shown in Code extract 11.6. This class adds the following private instance fields in lines 3–5:

- `jesscnc` This instance field holds the reference to the `JessCNC` instance of the master agent for shorter access.
- `returnedConcepts` This instance field will contain the `Concept` instances returned by Jess rules that called `return-fact-to-current-behaviour`.
- `marker` This instance field is of the type `WorkingMemoryMarker`. It is used to store a snapshot (a given state at a given time) of the working memory of the Jess engine.

The constructor initializes the object by first calling the constructor of its parent class (line 8). The object then registers itself with the Jess Function Manager by calling its method `registerWithJessFunctionManager` (line 9). Next, it records the current state of the working memory to which the working memory will be resetted every time before Jess is used (line 10).

The main method of the `JessAgentHandleAlertActionsBehaviour` class is the following:

- `handleAlert` This method is called for each `AlertAction` object. This method first notifies the Jess Function Manager that its object is the current behavior (line 14). It then casts its argument into an `AlertAction` and creates a new `Time` object that is set to the current time (line 15–17). Next, the working memory is set to a known state by calling the method `resetToMark` of the Jess Engine Manager and providing it with the `WorkingMemoryMarker` object created when the behavior was instantiated (line 18). This is necessary for the two reasons:
  - The facts are shadow facts. Calling `reset` will reinitialize each shadow fact but will not remove them from the working memory [Hil03, 94].
– Keeping AlertAction facts in the working memory until they expire is problematic. AlertActions that have not been treated by their dedicated recipient are loaded by the MediMAS framework from the database and added again to Jess. If a given alert is already in the working memory of Jess, it will be added again to the working memory. This is because the same alert is added as a new AlertAction object.

In lines 19–20, the AlertAction and the fact that represents the current system time (an instance of the class Time) are added to the working memory. The Jess module IMPORTANT that contain rules which should be able to run first, is set in line 21. In line 22, Jess is executed. When the Jess rule-engine has completed reasoning, the facts (in this case it is only going to be at most one, the AlertAction fact the behavior was launched for) that were added by calling return-fact-to-current-behaviour are passed as argument to the functions updateAlert and sendAlert (lines 23–28). The Vector object that contained the returned facts is cleared (line 29) and the behavior deregisters itself by calling removeAsCurrent (line 30).

Java does not support multiple inheritance of classes. Thus, the methods not commented in this Section are implemented in a default way.

The JessAgentHandleAlertActionsBehaviour object has to be instantiated by the Alert Manager agent’s AgentHandlePendingResultBehaviour behavior instead of the AgentHandleAlertActionsBehaviour. This is the only class of the MediMAS framework that needed to be altered.

11.4.4 Implementing the Alert Notification Logic with Jess Rules

Default rules are stored in expertSystem/rulefiles/ipd.clp (relative path). The two default rules for MediMAS are critical-alert and urgent-alert. They replace the Java algorithm of Code extract 11.1. They are both in the default module MAIN. In addition, a module IMPORTANT was defined to contain prioritized rules. It is currently empty. Each element is described in the following Sections.

Rule for Critical Alert Actions

This rule fires if a critical alert (the slot critical is true) that has not been treated by the recipient within the last 10 minutes, is in the working memory. In the right hand side (RHS) of the rule, the number of times this alert was raised in vain is increased (the slot nbOfAlerts) and the slot lastAlert time that contains the last time this alert was raised, is set the current time. The fact is returned to the current behavior with return-fact-to-current-behavior and removed from the working memory.

```
(defrule MAIN::critical-alert
  (Time (?now))
  (?a <- (AlertAction
            (critical TRUE)
            (nlab ?nlab)
            (senderPID ?sp)
            (requesterPID ?rp)
            (lastAlertTime ?l &:(>= (?now getTime) (?l getTime)) 10000))
            (nbOfAlerts ?nboa &:(> ?nboa))
        )
  =>
      (modify ?a
        (nbOfAlerts (+ ?nboa 1))
        (lastAlertTime ?now))
      (printout t "Critical Alert NLAB " ?nlab " Sender " ?sp " Requester " ?rp crlf)
```
11.4. Adding Jess Support to MediMAS

Rule for Urgent Alert Actions

This rule fires if an AlertAction that is urgent is in the working memory and has not been treated during the last 30 minutes by its recipient. The RHS of this rule is identical to the RHS of the critical-alert rule discussed above.

```
(defrule MAIN: :urgent-alert
  (Time (time ?now))
  (?a <- (AlertAction
    (urgent TRUE)
    (nlab ?nlab)
    (senderPID ?sp)
    (requesterPID ?rp)
    (lastAlertTime ?l 
      (>= (- (?now getTime) (?l getTime)) 30000))
    (nbOfAlerts ?nboa
      (> 7 ?nboa))))
  =>
  (modify ?a
    (nbOfAlerts (+ ?nboa 1))
    (lastAlertTime ?now))
  (printout t "Urgent Alert NLAB " ?nlab " Sender " ?sp " Requester " ?rp crlf)
  (return-fact-to-current-behaviour ?a)
  (retract ?a))
```

Module for High Priority Rules

An additional module IMPORTANT was defined which will contain prioritized rules. It has the focus when Jess runs and thus its rule will fire before the rules in MAIN.

11.4.5 Adding a JADE Aware Jess User Function

change-alert-check-time is a JADE aware Jess user function added for MediMAS. With this function, the time intervals of the ticker behavior AgentHandlePendingResultBehaviour that runs the JessAgentHandleAlertActionsBehaviour behavior, can be changed during run time. This is done by calling (change-alert-check-time TIME) where TIME is the time interval in

```
millisecond. The \textit{change-alert-check-time} Jess user function is implemented with the class \texttt{ChangeAlertCheckTime} (see Figure 11.5). This class extends the abstract class \texttt{JadeJessFunction}. The \texttt{call} method of the \texttt{ChangeAlertCheckTime} class is shown in Code extract 11.7. In line 2, the function’s agent is casted into a \texttt{JessableAlertManagerAgent} object. In line 3, the \texttt{setAgentHandleAlert} method of the \texttt{JessableAlertManagerAgent} object is called with the value provided by the \texttt{ValueVector} as argument. The \texttt{setAgentHandleAlert} instance method creates a new \texttt{AgentHandlePendingResultBehaviour} behavior with the time interval specified as argument.

\begin{verbatim}
public Value call(ValueVector arg0, Context arg1) throws JessException {
    JessableAlertManagerAgent jama = (JessableAlertManagerAgent) this.getAgent();
    jama.setAgentHandleAlert(arg0.get(1).intValue(arg1));
    return Funcall.TRUE;
}
\end{verbatim}

Code extract 11.7: The \texttt{call} method of the \texttt{ChangeAlertCheckTime} class

11.5 Changing the Behavior at Run Time

The rules in Section 11.4.4 do not add additional value to MediMAS. They simply replace the hard coded Java algorithm of Code extract 11.1. Indeed, with the J2J tool kit, the alert notification logic can be modified during run time by accessing the Jess instance of the Alert Manager agent with a remote Jess Administration agent. In the introduction of this thesis, two examples were given that would benefit from system that can have its comportment changed during run time. The examples were:

1. The head of the laboratory has decided that she wants to be notified of every result that is classified as \texttt{critical} and \texttt{urgent}.
2. A new strain of H5N1 was discovered in Switzerland. The Swiss Ministry of Health requires immediate notification if this strain is found in a human.

Both requests are implemented with Jess rules in the following Subsections. These rules can be added on the fly with the Jess Administration agent.

11.5.1 Sending Alerts To the Head of Laboratory

\begin{verbatim}
(defrule MAIN::urgent-and-critical-alert
    ?a <-- (AlertAction
        (critical TRUE)
        (urgent TRUE)
        (nlab ?nlab)
        (senderPID ?sp)
        (requesterPID ?rp))
    =>
    (printout t "Urgent Alert NLAB " ?nlab " Sender " ?sp " Requester " ?rp " crif"))
\end{verbatim}

Code extract 11.8: Handle urgent and critical cases

Rule \texttt{urgent-and-critical-alert} in Code extract 11.8 fires if a \texttt{AlertAction} that is critical and urgent is in the working memory. This example rule simply prints a message to the console.

11.5.2 Reacting to Laboratory Results with H5N1
11.5. Changing the Behavior at Run Time

Rule h5n1-found in Code extract 11.9 fires for every AlertAction in the working memory with the value of the slot icd10 set to "J09", the ICD-10 code for the avian influenza virus [20]. This rule prints a warning to the console. The reader shall note that the icd10 slot is currently not used by MediMAS. It was added for this Chapter. To test the rule, the following fact of Code extract 11.10 can be asserted with the Jess Administration agent:

```
(assert (MAIN::AlertAction
  (senderPID 10)
  (urgent TRUE)
  (critical TRUE)
  (requesterPID 3)
  (nlab 'nlab−117')
  (nbOfAlerts 1)
  (lastAlertTime (new java.util.Date))
  (notificationTime (new java.util.Date))
  (icd10 "J09"))
```

Code extract 11.10: A test fact for the h5n1-found rule of Code extract 11.9

A more sophisticated version of this rule would not only print a warning to the console but, for example, call a Jess user function to send an e-mail to the Swiss Ministry of Health. The following example shall clarify this idea: It shall be assumed that MediMAS has a class MediMASMail that has an instance method sendEMailTo. This method takes an e-mail address and a Concept instance as argument. Furthermore, it is assumed that the following JADE aware Jess user function send-email, implemented with the class SendEmail, takes an e-mail address and a fact as argument to send the fact to the owner of the e-mail address (i.e. line 4 of Code extract 11.9 would be (send-email "alert@bag.admin.ch" ?a)). This Jess function would call sendEMailTo of a MediMASMail object to send an e-mail. The source code of the SendEmail class is shown in Code extract 11.11

```
public class SendMail extends JadeJessFunction {
  public String getgetName() {
    return "send-email";
  }

  public Value call(ValueVector vv, Context context) throws JessException {
    if (vv.size() != 3)
      throw new JessException(this.getName(),
        "Wrong number of arguments", vv.size() - 1);
    MediMASMail m = new MediMASMail();
    String receiver = vv.get(1).stringValue(context);
    Concept concept = this.jessFactToJadeOntologyInstance(vv.get(2).factValue(context),
      context);
    m.sendEMailTo(receiver, concept);
    return Funcall.TRUE;
  }
}
```

Code extract 11.11: The SendEmail class
11.6 Conclusion

This Chapter demonstrated that the J2J tool kit can be applied, with only minor changes, to a new environment and in a different context. This being a result of the modular design of the J2J tool kit.

It is the author’s opinion that handling the alert notification management with the Jess does indeed offer considerable advantages compared to hard coding the decision making in Java instructions:

- The alert notification module with Jess is flexible: new rules can be added during run time.
- As it was shown with the Jess user function `change-alert-check-time`, the remote Jess Administration can potentially be used to manage the entire agent, not only its Jess instance.
- The alert notification module can easily be logged through the event listener interface of Jess, without "polluting" the business logic with code needed for the logging mechanism.
- By switching on the verbose mode of Jess, the alert notification can be monitored in real time.
12

Conclusion

12.1 Achievements

In the first part of this thesis, the business processes that support the exchange of laboratory results between the Laboratoire Cantonal de Fribourg and the Hôpital Cantonal de Fribourg are reviewed and modeled with the EPC notation. The potential for improvement of these processes let the development of MediMAS and the J2J tool kit.

The main goal of this thesis is the development of the J2J tool kit. It permits the use of Jess to handle the entire intelligence of a software agent or only a subset of the decision making. Jess, it was argued, has the following properties that make it an ideal choice to handle the decision making:

1. Decision making is written in a descriptive way: the knowledge engineer describes what happens when the working memory is in a given state, not when.

2. The Jess engine is an interpreter, thus rules can be added and removed during runtime.

3. Jess is written in Java and can seamlessly interact with Java applications.

Combined with the multi-agent framework JADE that lays the foundation for a (i) scalable, (ii) distributed and (iii) failsafe environment, it offers a comprehensive framework for the development of flexible, extensible and scalable distributed systems. The J2J tool kit provides seamless integration between Jess and JADE with the goal that a knowledge engineer does not have to be familiar with JADE or Java to write rules and a Java developer does not need to master Jess to develop a multi-agent system with Jess. Additionally, the J2J tool kit can be easily integrated in an existing multi-agent system without requiring major changes to its design and implementation. The following design choices have contributed to these goals:

- **The intelligent agent as a service user:** An agent is a service user of the class that manages the JADE and Jess integration, it does not have to extend an abstract class. It merely needs to implement an interface to permit its adherent objects (e.g. behaviors) to use its instance of the class *JessCNC* which manages
the Jess and JADE integration. Providing the Jess and JADE integration with an abstract agent class would severely limit its usefulness for an existing multi-agent system because it would require to alter the existing system’s class hierarchy. This is due to the fact that Java does not provide multi-inheritance of classes, only of interfaces.

- **Interoperability:** The knowledge artifacts managed by JADE and Jess are based on the same semantics and thus can seamlessly be exchanged between both systems without loss of information or cumbersome transformation rules. This is achieved by creating a generic representation of the domain specific ontology already used by the JADE agents, which can be mapped to different formats such as templates that represent classes in Jess or the string representation of the ontology that is sent to the remote Jess Administration agents. The mapping from the generic ontology representation to a specific format is achieved with Visitors and is not part of the generic ontology representation. Hence the generic ontology representation can be easily mapped to a variety of formats by adding new Visitors.

- **JADE services for Jess:** The instance of the Jess rule-engine of a particular agent that uses the J2J tool kit, can communicate with other agents in its host agent’s environment or with the agent’s behavior, without needing to use the Java API, by calling Jess user functions that were implemented as part of the J2J tool kit. If these functions are provided with a fact as argument, the fact is automatically transformed into a Java object that can be used by JADE. This is possible since both systems use the same ontology.

- **Remote Administration:** The Jess instance of an agent can be accessed remotely through a command line interface to remotely update the knowledge base. Thus rules can be added and removed and facts added, removed and modified during run time. The remove Jess Administration interface is also a useful tool for debugging.

- **Generic tool kit:** The J2J tool kit is not bound to a specific domain nor does it dictate a certain type of agent organization. This thesis was initiated to improve MediMAS. Agents in the MediMAS environment handle the coordination primarily based on "organizational structuring". The Alert Manager agent, which is the master agent, coordinates the activities of its slaves that represent the actors involved in a healthcare process. Therefore, the Matrix example which was used to develop and test the framework, uses the same type of agent organization. Nevertheless, the J2J tool kit can easily be used for any type of agent and agent system.

- **Modular design:** The design of the J2J tool kit is based on loosely coupled classes by applying design patterns. New functionalities can be added without modifying the existing tool kit, thus further promoting the use of the J2J tool kit in a wide variety of applications.

The J2J tool kit was successfully applied in two different multi-agent environments: first the Matrix, which consists of one master agent and several reactive slave agents. The master agent coordinates the entire workflow and is fully aware of each slave agent’s state. In the second use case, the J2J tool kit was added to an already existing multi-agent framework, MediMAS. The J2J tool kit is not, as it is the case with the master agent in the Matrix environment, in charge of the entire process. Instead, the Alert Manager, which is the master agent in the MediMAS environment, uses Jess only to decide how to
react to individual alerts based on their current state. The other actions in the workflow are assigned to behaviors.

12.2 Future Work

The J2J tool kit will be deployed and further developed by the Software Engineering Research Group. In this Section, the author makes a set of recommendations for the future development of the tool kit:

**Configuration File:** Currently the J2J tool kit does not use a central configuration file. Thus it is required to import the source code of the J2J tool kit into a project that uses the tool kit to change certain values, notably the path for the ontology class files and the reference to the current ontology instance in the class `OntologyTreeManager`. The relative path to the file that contains the default rules is hard coded as well in the `JessCNC` class. The author recommends to implement a central configuration file for the J2J tool kit.

**Automatic Generation of "(agent action, behavior)" Pairs:** Part of the service provided by the `OntologyClassManager` class are "(Agent Action, Behavior)" pairs. This specific functionality of the `OntologyClassManager` class is used for the `command-agent` Jess user function. The `command-agent` function takes an `AgentAction` instance and the `AID` object of the recipient as argument. It sends the agent action to the recipient with an `ACLMessage`. This `ACLMessage` object is passed as constructor argument to the behavior that corresponds to the agent action. An agent action can be assigned unambiguously to a behavior if the conventions defined in Section 5.2.3 are followed. That is, for every agent action, there is exactly one behavior to initiate a FIPA-Request conversation and one behavior to react to a FIPA-Request conversation. Currently the generation of these pairs is hard coded in the constructor of the `OntologyClassManager` class, as a proof of concept. The list of "(agent action, behavior)" pairs can be generated automatically by looking up, for every agent action, the behavior that contains the agent action’s name and the suffix "Initiator", assuming the behaviors are named as recommended in Section 5.2.3.

**Further Development of Remote Jess Administration:** New functionalities could be added to the remote Jess Administration, such as:

- Improving the GUI of the remote Jess Administration agent. The current implementation is a prototype.
- Saving newly added rules as default rules and permanently deleting existing rules.
- Displaying the attributes of the ontology classes.
- Uploading a new rule file.
- Dumping the current state of the working memory to the file system.
- Authentication and Authorization: Currently there is no access restriction implemented. In a testing environment with dummy data, this is not an issue but it would be required, if ever real data were to be processed.
12.2. Future Work

**Improved Facts to Object Mapping:** The algorithm that maps facts passed to JADE aware Jess user functions, located in the abstract class `JadeJessFunction`, can only transform facts that are concepts in the JADE ontology into objects, not predicates. Furthermore, lists are not supported as well. This algorithm should be improved to support all types of facts.

**Support for Multiple Ontologies:** An Ontology Tree Manager is limited to one ontology. It would be useful to have an Ontology Tree Manager that could manage several ontologies. As an example, an Ontology Tree Manager could manage one enterprise ontology and one information ontology. The enterprise ontology models the context in which the multi-agent system executes, such as employees, processes and activities. The information ontology describes the information objects, their attributes and the relationships between the information objects. The terms enterprise ontology and information ontology are discussed in [ABH+98, p. 44].

**Adding Support for UMLS:** The Unified Medical Language System (UMLS) framework provides the means to develop applications "that behave as if they "understand" the meaning of the language of biomedicine and health" [26]. UMLS can be used in a wide range of applications that interact with medical data in a variety of medical domains, e.g. patient care, index, cataloging biomedical literature, health service billing [27]. UMLS could be used to classify laboratory results by their medical content. The author recommends to use a separate Ontology Agent for UMLS, due to the frameworks high resource requirements [28].

**Add Support for Backup Agents:** Agents that fulfill critical tasks and are unique in a multi-agent environment, for example the master agent in the context of a master/slave organization, represent a single point of failure for the multi-agent environment they exist in. The risks posed by this single point of failure could be mitigated by adding backup agents for the critical agents. When a critical agent fails, one of its backup agents takes over; the system continues to function. The author suggests the following approach to explore: In the master's sub environment, each agent knows all of its peers. The master agent forwards every change of its state (e.g. a fact or a rule is added or removed from the knowledge base) to its backup agents, making the backup agents identical copies of the master agent. When the master agent is inactive, its sends "still alive" messages to its backup agents, for them to know the master agent is still operational. When the backup agents do not receive a message from the master agent, either an update of the state of the master agent or a "still alive" message, they have to assume that the master agent is down. The backup agents start a negotiation process to determine which backup agent will take over as master. The new master registers itself with the Directory Facilitator Agent as the master agent. After a given time period, the slave agents notice, by the absence of acknowledges to their messages from the master agent that the master agent is not responding anymore. They lookup the master agent in the Directory Facilitator agent and resend the messages. The failure recovery will only work for hardware failures owing to the fact that a software bug should affect the backup agents as well. Thus each master must run on a separate computer. Additionally, the slave agents will have to be slightly adapted to handle the non-responsiveness of the master agent.

A simple example with two slave agents, one Directory Facilitator agent and one master
12.2. Future Work

(a) Master agent operates normally

(b) Master agent fails

(c) Backup agents negotiate and slave lookup the new master

(d) The system continues to function

Figure 12.1: Eliminating the single point of failure by providing redundancy with backup agents

agent with three backup agents assigned, is given in Figures 12.1(a) to 12.1(d). Figure 12.1(a) shows the system when it functions normally. The agents Slave 1 and Slave 2 send message \( m_1 \) respectively message \( m_2 \) to the master agent. Both are acknowledged by the master agent and also forwarded to its backup agents. In Figure 12.1(b), the master agent malfunctions and cannot acknowledge the messages \( m_3 \) and \( m_4 \). Furthermore, the backup agents receive no signal from the master agent. After a given timeout period, the master agent is presumed dead. The backup agents start a negotiation process to decide which agent is going to act as master agent. In this example, agent Backup 1 is chosen and registers itself with the Directory Facilitator agent as the new master agent. The slave agents, after a given time out period, lookup the new master agent in the Directory Facilitator agent and receive the address of the agent Backup 1. This is shown in Figure 12.1(c). Next, as depicted in Figure 12.1(d), the slave agents send the messages \( m_3 \) and \( m_4 \) again to the acting master agent. The new master agent acknowledges the messages and forwards both messages to the remaining backup agents. The system did not fail.

When a new backup agent enters the system, it looks up the master agent and registers itself with the master agent. If there is no master agent in the system, the new backup
12.2. Future Work

The forwarding of the messages from the master agent to its slaves could be implemented by using the event listener interface of Jess [Hil03, p. 418–419]. For each event of interest, a given event listener could react to the event by forwarding the event to the backup agents.

**MediMAS with Multiple Intelligent Agents:** It would be interesting to develop a different MediMAS framework with several intelligent agents. The decision making could be distributed among each agent that represents an actor in the healthcare workflow. For example, instead of having the Alert Manager agent as the only intelligent agent in the MediMAS environment, each laboratory agent could use Jess. Thus there would not be a single intelligent master agent but a set of intelligent laboratory agents that would need to negotiate to execute the workflows they are designed to support. This would eliminate the single point of failure at the cost of more negotiation and the lack of one agent with the global view.
Part IV

Appendix
Healthcare Business Processes at the HCF and LCF

These images were designed by Minh Tuan Nguyen and Patrik Fuhrer of the Software Engineering Group, University of Fribourg, Switzerland.
Physician accesses WinDMLAB to get the results

- Request result on WinDMLAB
- Wait for 1 hour
- [Physician can wait]
- [Result not available]
- [Result available]
- [Physician cannot wait any longer]
- Physician asks for result by phone
Physician asks for results by phone

Laboratory Assistant

- Result becomes available
  - [Physician cannot wait any longer]
  - Lab Assistant calls the physician back by phone to give him the results

Physician

- [Result not available]
- [Result available]

- [Physician can wait]
  - [Unreachable]
  - Wait for 45 min
  - [Unreachable 1%]
  - Physician asks for result

- [K < 3]
  - Physician phones the laboratory
  - [Reachable]
  - Physician calls the Head of Laboratory
  - [Reachable 99%]
Laboratory Assistant calls the physician back by phone to give him the results

- Lab Assistant phones the physician
- Wait for 30 min
- Lab Assistant notifies the Head of Laboratory of the situation
- [Reachable]
- Head of laboratory physically delivers results to physician
- Lab Assistant physically delivers results to physician
- Lab Assistant phones the Head of Laboratory
- [Urgent OR k>=3]
- [NOT Urgent AND k < 3]
- [Unreachable]
- Lab Assistant gives the result
- [Reachable]
- [Unreachable AND NOT Urgent]
- [Unurgent]
The following UML Sequence Diagrams correspond to the red demarcations of Figure 5.2, Page 37.
Figure B.2: Step 2
Figure B.3: Step 3
Figure B.4: Step 4

1. serveVerticalCommand(gCmd)
2. filter(gCmd)
3. postMessage(msg)
4. processIncoming(gCmd)
5. addLast(msg)
6. accept(gCmd)
7. filter(cmd)
8. filter(cmd)
9. handleSendMessage(cmd)
10. sendMessage(msg, receiverID)
11. postMessageToLocalAgent(msg, receiverID)
12. handleSendMessage(cmd)
13. doWake
14. doWake

Inactive intermediate filters are omitted

Prompts a vertical message to be sent to the notification service. This step is omitted

Setting parameters based on the values of the incoming generic command
Installing and Running the Matrix

To run the Matrix, the following libraries must be in the class path:

- **JADE**: For the multi-agent framework
  - http.jar
  - iiop.jar
  - jade.jar
  - jadeTools.jar
  - commons-codec-1.3.jar

- **Jess**: For the rule-based system
  - jess.jar

- **Jigloo**: For the GUI framework used by the remote Jess Administration agent
  - appFramework-1.0.jar
  - jnlp.jar
  - looks-2.1.4.jar

blah blah run ant and set profile
D.1 Creating the JTree of the Ontology

```java
private void createJTree() {
    Vector<Tuple<String, DefaultMutableTreeNode>> helper = new Vector<Tuple<String, DefaultMutableTreeNode>>() {
        Tuple<String, DefaultMutableTreeNode> t = null;
        for (String s : this.parseme) {
            t = new Tuple<String, DefaultMutableTreeNode>();
            t.setFirstElement(s);
            t.setSecondElement(null);
            helper.add(t);
        }
        Stack<Tuple<String, DefaultMutableTreeNode>> main = new Stack<Tuple<String, DefaultMutableTreeNode>>() {
            Tuple<String, DefaultMutableTreeNode> child = null;
            Tuple<String, DefaultMutableTreeNode> parent = null;
            Tuple<String, DefaultMutableTreeNode> root = null;
            for (Tuple<String, DefaultMutableTreeNode> tn : helper) {
                if ("(".equals(tn.getFirstElement())) {
                    continue;
                }
                if (")".equals(tn.getFirstElement())) {
                    child = main.pop();
                    if (child.getSecondElement() == null) {
                        child.setSecondElement(new DefaultMutableTreeNode(child.getFirstElement()));
                        if (main.size() > 0) {
                            parent = main.peek();
                            if (parent.getSecondElement() == null) {
                                parent.setSecondElement(new DefaultMutableTreeNode(parent.getFirstElement()));
                                parent.getSecondElement().add(child.getSecondElement());
                            } else {
                                root = child;
                                break;
                            }
                        } else {
                            continue;
                        }
                    }
                    main.push(tn);
                }
            }
            JessAdminAgent jaa = (JessAdminAgent) this.myAgent;
            jaa.getJessAdminGui().setJTreeRoot(root.getSecondElement());
        }
    };
}```

Code extract D.1: Algorithm to create a JTree that contains the ontology

The algorithm of Code extract D.1 is located in the class `OntologyTreeChangedResponder`. It uses a vector of strings to generate a JTree that represents the ontology used by Jess. In line 2, the `Vector` that will contain "`String, DefaultMutableTreeNode`" pairs is declared and line 3 declares the `Tuple` variable to store a "`String, DefaultMutableTreeNode`" pair. The `String` of such
an object holds the name of a node in the ontology tree, the `DefaultMutableTreeNode` the corresponding node in the JTree. The `for` loop in lines 4–9 creates, for each `String` object of the `Vector` containing the names of the ontology classes, a “`String, DefaultMutableTreeNode`” object and adds it to the `helper Vector`. Lines 10–13 declare and initialize the variables required to create the JTree. The JTree is created in the `for` loop in lines 14–34: If the first element of the current object contains a left parenthesis, it is dropped (line 15–16). If the first element of the current object contains a right parenthesis, it indicates that the end of a list has been reached, thus the next element in the stack `main` that contains the nodes of the ontology that have not yet been added to the JTree, is popped (lines 17–19). If the second element of the current object is `null`, a `DefaultMutableTreeNode` with the name of the ontology class it represents is instantiated (line 20). If there are still nodes left to be treated (line 21), the parent of the current object is accessed and the current object is linked to its parent (lines 22–25). If the parent node does not yet have a `DefaultMutableTreeNode` object, a `DefaultMutableTreeNode` object is instantiated and a reference to that object is saved as the second element of the current object (line 24). If there are no more objects left in `main`, the current object is the root (line 27) and the loop is exited. If the current object is not a parenthesis, it is pushed onto the `main` stack (line 32). In line 34 the GUI object of the behavior’s agent is accessed and the second element of the `root` object set as the root of the JTree root (line 35).

**D.2 Sorting the Ontology Classes**

The algorithm to sort the classes in the ontology is shown in Code extract D.2. The two dimensional `Vector` that will contain the final result is created at lines 2–5. Line 6 states the directory location of the ontology Java classes. At line 8 and 9 two `Vector<String>` objects are created. The instance methods `getConceptNames` and `getPredicateNames` of the ontology object return all class names in lower case. Because for dynamic loading of Java classes will require the correct name of the Java class, the `directoryContent Vector` is required. A class at position $p$ is in `directoryContent` is the same class as in `searchDirContent`. The classes are read from the file system. Each class that is in the directory and in the lists returned by the ontology object is added to `domainOntologyContent` (line 10–12, not shown in the source code extract). At lines 13–15 additional variables are declared that will be used by the sorting algorithm. If the ontology contains elements, the first element is taken (lines 16–17). The classes names in `domainOntologyContent` are in lower case, therefore the proper class name has to be first looked up in `searchDirContent` to get the position of the class in the `Vector` and access the original name of the class in `directoryContent`. This name is needed to dynamically load the class (lines 19–22). At lines 23-25, the super class is accessed, dynamically loaded and the name transformed to lower case for the lookup. The `if` statement at line 27 is true for a class that either has no super class in the ontology because it is at the highest level or because its parent node has already been processed and removed from `domainOntologyContent`. Thus the element in `current` is not at the top of the tree, it is added back to `domainOntologyContent` at the position of its parent and the parent is set as the current element (lines 27–30). Next, he current iteration is skipped (line 31).

If the current class does not have a super class in `domainOntologyContent`, the class’s name and `Class` object are added to `stringResult`, respectively `classResult` (lines 33–34). If there are still classes left to be sorted, first element of `domainOntologyContent` is removed and set as the value of `current`. Otherwise, `current` is set to `null` and the loop is exited (lines 35–38).
D.3 Creating the Ontology Composite

The algorithm that builds the ontology composite is shown in Code extract D.3. The instance method `buildVisitorWorthyTree` takes a two dimensional `Vector` array, created by `buildSimpleTree`, as argument. At lines 2–7 variables are declared that will be used by the algorithm. As long as there are classes that have not yet been added to the ontology composite (line 8), the following is done: The class that is at the beginning of `currentS` and `currentC` is removed from these `Vectors` and its super class loaded (9–12). If the super class is already part of the final composite tree, the (future) parent node is looked up the in the `ontologyClasses` hash (lines 13–14). Otherwise, possible interfaces the class may extend are fetched and each looked up in the `ontologyClasses` hash. A result will be found for the classes are the direct child nodes of the second-level nodes in the initial ontology composite. The for loop is aboarded as soon as a parent is found (lines
The current node is created with the `createOntologyElement` Factory method, the parent and child relationships are set for the current node and the parent of the node, and the current node is added to the `ontoloyClasses` hash since it now part of the tree and can accept child nodes (lines 23–27). The loop is exited when all classes have been added to the ontology composite.

```java
private void buildVisitorWorthyTree(Vector[] tree) {
  Vector<String> className = tree[0]; Vector<Class> classClass = tree[1];
  String currentS = null; Class currentC = null;
  String superClassS = null; Class superClassC = null;
  Class[] interfaces = null;
  OntologyElement parent = null; OntologyElement currentE = null;
  boolean found = false;
  while (className.size() > 0) {
    currentS = className.remove(0);
    currentC = classClass.remove(0);
    superClassC = currentC.getSuperclass();
    superClassS = superClassC.getName();
    if (this.ontoloyClasses.containsKey(superClassS)) {
      parent = this.ontoloyClasses.get(superClassS);
    } else {
      interfaces = currentC.getInterfaces();
      for (Class c : interfaces)
        if (this.ontoloyClasses.containsKey(c.getName())) {
          parent = this.ontoloyClasses.get(c.getName());
          break;
        }
    }
    currentE = this.createOntologyElement(parent.getClass().getSimpleName(), currentS);
    currentE.setJavaClass(currentC);
    parent.addChild(currentE);
    currentE.setParent(parent);
    this.ontoloyClasses.put(currentE.getName(), currentE);
  }
}
```

Code extract D.3: Algorithm that creates the composite representing the ontology.
Common Acronyms

ACL  Agent Communication Language
ADs  Activity Diagrams
AD   Activity Diagram
AID  AgentIdentifier
AMS  Agent Management System
API  Application Programming Interface
ARIS Architecture of Integrated Information System
BDI  Beliefs, Desires and Intentions
BPM  Business Process Management
CLIPS C Language Integrated Production System
CT   Container Table
DF   Directory Facilitator
DF   Directory Facilitator
EPC  Event-Driven Process Chain
EPC  Event-driven Process Chains
FIPA Foundation for Intelligent Physical Agents
GADT Global Agent Descriptor Table
GUI  Graphical User Interface
ICD-10 International Classification of Diseases 10
ICT  Information and Communications Technology
IMTP Internal Message Transport Protocol
IT   Information Technology
J2J  Jess 2 JADE
JADE Java Agent Development Framework
**JessCnC**  Jess Command and Control

**LADT**  Local Agent Descriptor Table

**LEAP**  Lightweight Extensible Agent Platform

**LGPL**  Library Gnu Public License

**LHS**  left-hand side

**LISP**  List Processing

**MTP**  Message Transport Protocol

**MTS**  Message Transport Service

**OA**  Ontology Agent

**OKBC**  Open Knowledge Base Connectivity

**OWL**  Web Ontology Language

**PDA**  Personal Digital Assistant

**RHS**  right hand side

**RMA**  Remote Monitoring Agent

**RMI**  Remote Method Invocation

**UMLS**  Unified Medical Language System

**UML**  Unified Modeling Language

**URI**  Uniform Resource Identifier

**URL**  Uniform Resource Locator

**eEPC**  extended Event-Driven Process Chain
License of the Documentation

Copyright (c) 2008 Joël Vogt.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

The GNU Free Documentation Licence can be read from [4].
Website of the Project

A web-page was created for this project: http://www.gmipsoft.com/rfid\(^1\). On this page you will find:

- The API of the project.
- This binaries and sources of this documentation.
- The binaries and sources of the.
- Parts of the content (see H).

Figure G.1 provides a screenshot of this website.

\(^1\)This URL is a shortcut to http://diuf.unifr.ch/softeng/student-projects/completed/guinard/index.html
Figure G.1: Screenshot of the project's official web-page
On the CD-ROM Figure H.2 of the project you will find:

- The source code, Ant files and compiled binaries of the.
- The APIs of the.
- The binaries and sources of this documentation.
- Various documents that were of great use during this bachelor thesis.

Figure H.1 provides a tree view of the CD-ROM.
The content of the CD-ROM can also be downloaded from the official website of the project (see G).
Figure H.1: Tree view of the content of the CD-ROM

Figure H.2: The CD-ROM of this project
References


[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Indianapolis IN USA, 1995.


Referenced Web Resources


