Visualizing Agent Communications Graphical Interfaces for Conversation Paradigms

Aurini, Tristan

http://hdl.handle.net/11023/2633
master thesis

University of Calgary graduate students retain copyright ownership and moral rights for their thesis. You may use this material in any way that is permitted by the Copyright Act or through licensing that has been assigned to the document. For uses that are not allowable under copyright legislation or licensing, you are required to seek permission.

Downloaded from PRISM: https://prism.ucalgary.ca
UNIVERSITY OF CALGARY

Visualizing Agent Communications

Graphical Interfaces for Conversation Paradigms

by

Tristan J. A. Aurini

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

October, 2015

© Tristan J. A. Aurini 2015
Abstract

This research puts forth a graphical user interface for interacting with and modifying conversations used by software agents in a Multi Agent System (MAS) architecture (CASA). MAS Communications is a field of Artificial Intelligence focusing on improving the way agents communicate. Thus message exchanges are made meaningful by implementing concepts surrounding context. Additionally, they should be flexible in order to avoid problems due to unpredictability. Lastly, these paradigms should be simple to ease their adoption.

Using this interface one can visualize how different conversations progress and why they do so allowing for the identification of error sources in the system. The user can also modify existing conversation types, create new ones, and instantiate implementations of them for individual agents.

The conversation types are typically distinguished by what is called the performative of the communicative act. This is a concept adopted from linguistics where the performative indicates a conversation’s intent/desires.
Acknowledgements

I would like to thank my program Supervisor Dr. Robert Kremer for all of the help and guidance he has given me. It has been invaluable.
Dedication

For my wife, Mayumi, and our daughter, Amelia.
# Table of Contents

**Abstract** .......................................................... ii
**Acknowledgements** .............................................. iii
**Dedication** .......................................................... iv
**Table of Contents** ................................................ v
**List of Tables** ..................................................... viii
**List of Figures** ..................................................... x
**List of Symbols** ................................................... xiii
**List of Definitions** ................................................ xiv

1. **Introduction** ..................................................... 2
   1.1 **Agents and Multi Agent Systems** ............................. 2
   1.2 **Design Considerations of Multi Agent Systems** .............. 4
   1.3 **Why Agent Communications?** .................................. 6
   1.4 **MAS design platforms** ....................................... 7
   1.5 **Contribution** ................................................ 10
   1.6 **Objectives** .................................................. 11
   1.7 **Summary** .................................................... 13

2. **Background** ...................................................... 14
   2.1 **The Basis of all Communications** ............................. 14
   2.2 **Applications to Agent Communications** ....................... 17
   2.3 **Conversations in CASA** .................................... 22
   2.4 **CASA Conversation Lisp Definitions** ......................... 27
      2.4.1 **Lisp Overview** ................................ .......... 27
      2.4.2 **Conversation Types** ................................... 30
      2.4.3 **Conversation Calls** ..................................... 33
      2.4.4 **Policies** ................................................ 35
      2.4.5 **Conditionals** .......................................... 36
      2.4.6 **Stray Lisp Code** ....................................... 37
      2.4.7 **Comments** .............................................. 37
   2.5 **Summary** .................................................... 38

3. **Requirements** ................................................... 40
   3.1 **Visualization Requirements** .................................. 40
   3.2 **Error Reporting Requirements** ............................... 42
   3.3 **Saving Requirements** ....................................... 42
   3.4 **Animation Requirements** .................................... 42
   3.5 **Summary** .................................................... 43

4. **Design** ........................................................... 44
   4.1 **Selecting an Appropriate Scheme** ......................... 44
   4.2 **Design Choices** ............................................ 45
   4.3 **Summary** .................................................... 49

5. **Implementation** ................................................ 50
   5.1 **Implementation Options** .................................... 50
   5.2 **Overview of UMLet** ......................................... 51
5.3 Meeting the Design Objectives ........................................ 57
  5.3.1 Element Implementations ........................................ 57
  5.3.2 User Interactions .................................................. 68
  5.3.3 Input and Output Implementation ................................ 70
5.4 Classes and their Operations ........................................ 72
5.5 Diagrams and their operations ...................................... 73
  5.5.1 The Speech Act Editor ........................................... 74
  5.5.2 The Agent Startup File/Action Level Editor .................... 77
  5.5.3 The Animation Viewer ............................................ 78
5.6 Other Utilities and Methodology .................................... 80
  5.6.1 The Layout Manager ............................................ 80
  5.6.2 Popup Menu Maker .............................................. 80
  5.6.3 Listeners ......................................................... 81
  5.6.4 Abstract Lisp File Class ........................................ 81
5.7 Summary ............................................................... 82
6 Testing ................................................................. 83
  6.1 Performing the Tests .................................................. 85
    6.1.1 Reading of Lisp Files .......................................... 85
    6.1.2 Writing of Lisp Files .......................................... 86
    6.1.3 Supporting Logic: ............................................. 86
    6.1.4 Diagram Interface: ............................................ 86
    6.1.5 User Interaction: ............................................. 87
  6.2 Bugs .................................................................. 88
  6.3 Summary ............................................................... 89
7 Analysis ................................................................. 91
  7.1 Creating A Single Use Subscription Conversation ................ 91
    7.1.1 Making the Client & Server-side Conversation Types ........ 92
    7.1.2 Making the Concrete Client & Server-side Conversations .... 103
    7.1.3 Testing the New Conversations ................................ 114
  7.2 Results ............................................................... 126
  7.3 Related Work ......................................................... 126
  7.4 Summary ............................................................... 128
8 Conclusion ........................................................... 130
  8.1 Goals and Implementations ......................................... 130
    8.1.1 Unintended Design Choices and Benefits ................. 132
    8.1.2 Drawbacks of the Implementation ............................ 134
    8.1.3 Future Work .................................................... 136
  8.2 Concluding Remarks ................................................ 138
  8.3 Summary ............................................................... 140
Bibliography ............................................................. 141
A CASA Client-Side Request Conversation ................................. 146
B Project Requirement Points ........................................... 151
B.1 Visualization Requirements Points ................................. 151
B.2 Error Reporting Requirements Points ............................. 155
B.3 Saving Requirements Points ...................................... 156
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Conversation Types Defined in CASA</td>
<td>22</td>
</tr>
<tr>
<td>2.2</td>
<td>List of the FIPA communicative acts</td>
<td>39</td>
</tr>
<tr>
<td>E.1</td>
<td>Palette File Name</td>
<td>204</td>
</tr>
<tr>
<td>E.2</td>
<td>Switch Palette File</td>
<td>204</td>
</tr>
<tr>
<td>E.3</td>
<td>Create New Palette File</td>
<td>205</td>
</tr>
<tr>
<td>E.4</td>
<td>Save Palette File</td>
<td>205</td>
</tr>
<tr>
<td>E.5</td>
<td>Import Additional Files</td>
<td>205</td>
</tr>
<tr>
<td>E.6</td>
<td>Reorganize elements in Palette</td>
<td>206</td>
</tr>
<tr>
<td>E.7</td>
<td>Hide Palette Comments</td>
<td>206</td>
</tr>
<tr>
<td>E.8</td>
<td>Reorganize elements in Palette with hidden comments</td>
<td>206</td>
</tr>
<tr>
<td>E.9</td>
<td>Drag Comment from Palette to Speech Act diagram</td>
<td>207</td>
</tr>
<tr>
<td>E.10</td>
<td>Drag Comment from Palette to non speech act diagram</td>
<td>207</td>
</tr>
<tr>
<td>E.11</td>
<td>Remove Comment from diagram</td>
<td>208</td>
</tr>
<tr>
<td>E.12</td>
<td>Modify Comment Contents and Save to Palette</td>
<td>208</td>
</tr>
<tr>
<td>E.13</td>
<td>Add New Comment to Palette</td>
<td>209</td>
</tr>
<tr>
<td>E.14</td>
<td>Delete New Comment from Palette</td>
<td>209</td>
</tr>
<tr>
<td>E.15</td>
<td>Drag Lisp Conversation Template from Palette</td>
<td>209</td>
</tr>
<tr>
<td>E.16</td>
<td>Minimize open Lisp Conversation Template</td>
<td>210</td>
</tr>
<tr>
<td>E.17</td>
<td>Maximize open Lisp Conversation Template</td>
<td>210</td>
</tr>
<tr>
<td>E.18</td>
<td>Close open Lisp Conversation Template</td>
<td>210</td>
</tr>
<tr>
<td>E.19</td>
<td>Add New Lisp Conversation Template to Palette</td>
<td>211</td>
</tr>
<tr>
<td>E.20</td>
<td>Add Global Policy to new Lisp Conversation Template</td>
<td>211</td>
</tr>
<tr>
<td>E.21</td>
<td>Add Global Policy to Lisp Conversation Template</td>
<td>212</td>
</tr>
<tr>
<td>E.22</td>
<td>Add Local Policy to new Lisp Conversation Template</td>
<td>212</td>
</tr>
<tr>
<td>E.23</td>
<td>Add Comment to new Lisp Conversation Template</td>
<td>213</td>
</tr>
<tr>
<td>E.24</td>
<td>Add Code Block to new Lisp Conversation Template</td>
<td>213</td>
</tr>
<tr>
<td>E.25</td>
<td>Set ordinality of global policy node to beginning</td>
<td>213</td>
</tr>
<tr>
<td>E.26</td>
<td>Set position of node to end</td>
<td>214</td>
</tr>
<tr>
<td>E.27</td>
<td>Add Conditional with default condition to Lisp Conversation Template</td>
<td>214</td>
</tr>
<tr>
<td>E.28</td>
<td>Add Conditional with proper condition to Lisp Conversation Template</td>
<td>215</td>
</tr>
<tr>
<td>E.29</td>
<td>Remove Conditional's 'then-case' child</td>
<td>215</td>
</tr>
<tr>
<td>E.30</td>
<td>Add Comment to Conditional</td>
<td>215</td>
</tr>
<tr>
<td>E.31</td>
<td>Add Local Policy to Conditional</td>
<td>216</td>
</tr>
<tr>
<td>E.32</td>
<td>Swap children of Conditional</td>
<td>216</td>
</tr>
<tr>
<td>E.33</td>
<td>Add Global Policy to Conditional</td>
<td>217</td>
</tr>
<tr>
<td>E.34</td>
<td>Add Conversation call to new Lisp Conversation Template</td>
<td>217</td>
</tr>
<tr>
<td>E.35</td>
<td>Bind Variable to Conversation call</td>
<td>218</td>
</tr>
<tr>
<td>E.36</td>
<td>Bind State to Conversation call</td>
<td>218</td>
</tr>
<tr>
<td>E.37</td>
<td>Add Imported Conversation Call</td>
<td>218</td>
</tr>
<tr>
<td>E.38</td>
<td>Switch Contains Visibilities</td>
<td>219</td>
</tr>
<tr>
<td>E.39</td>
<td>Switch Equivalence Visibilities</td>
<td>219</td>
</tr>
<tr>
<td>Task</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>E.40 Switch Transition Visibilities</td>
<td>219</td>
<td></td>
</tr>
<tr>
<td>E.41 Switch Comment Visibilities</td>
<td>220</td>
<td></td>
</tr>
<tr>
<td>E.42 Switch Code block Visibilities</td>
<td>220</td>
<td></td>
</tr>
<tr>
<td>E.43 Switch Policy Visibilities</td>
<td>221</td>
<td></td>
</tr>
<tr>
<td>E.44 Switch Conversation Visibilities</td>
<td>221</td>
<td></td>
</tr>
<tr>
<td>E.45 Switch Mouse Over Text Visibilities</td>
<td>221</td>
<td></td>
</tr>
<tr>
<td>E.46 Reactivate all Visibilities</td>
<td>222</td>
<td></td>
</tr>
<tr>
<td>E.47 Remove child from Lisp Conversation Template</td>
<td>222</td>
<td></td>
</tr>
<tr>
<td>E.48 Edit documentation of Lisp Conversation Template</td>
<td>222</td>
<td></td>
</tr>
<tr>
<td>E.49 Edit comments of Lisp Conversation Template’s variable</td>
<td>223</td>
<td></td>
</tr>
<tr>
<td>E.50 Open Agent initialization file</td>
<td>223</td>
<td></td>
</tr>
<tr>
<td>E.51 Create Agent initialization file</td>
<td>224</td>
<td></td>
</tr>
<tr>
<td>E.52 Drag Conversation Call from Palette with success</td>
<td>224</td>
<td></td>
</tr>
<tr>
<td>E.53 Drag Conversation Call from Palette with fail</td>
<td>225</td>
<td></td>
</tr>
<tr>
<td>E.54 Drag Conversation Call from Palette with cancel</td>
<td>225</td>
<td></td>
</tr>
<tr>
<td>E.55 Add Comment to File Node</td>
<td>225</td>
<td></td>
</tr>
<tr>
<td>E.56 Add Code Block to File Node</td>
<td>226</td>
<td></td>
</tr>
<tr>
<td>E.57 Add Global Policy to File Node with success</td>
<td>226</td>
<td></td>
</tr>
<tr>
<td>E.58 Add Global Policy to File Node with cancel</td>
<td>226</td>
<td></td>
</tr>
<tr>
<td>E.59 Migrate Conversation Call to File Node</td>
<td>227</td>
<td></td>
</tr>
<tr>
<td>E.60 Save Agent Initialization File</td>
<td>227</td>
<td></td>
</tr>
<tr>
<td>E.61 Modify agent initialization files and save</td>
<td>228</td>
<td></td>
</tr>
<tr>
<td>E.62 Play Conversation Instance Animation</td>
<td>228</td>
<td></td>
</tr>
<tr>
<td>E.63 Clear Conversation Instance Animation</td>
<td>229</td>
<td></td>
</tr>
<tr>
<td>E.64 Close Conversation Instance</td>
<td>229</td>
<td></td>
</tr>
</tbody>
</table>
# List of Figures and Illustrations

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Finite State Machine</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>BDI Rule Engine</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>Performative Subsumption Lattice</td>
<td>23</td>
</tr>
<tr>
<td>2.4</td>
<td>Request Conversation Sequence Diagram</td>
<td>25</td>
</tr>
<tr>
<td>2.5</td>
<td>Lisp Function Definition</td>
<td>28</td>
</tr>
<tr>
<td>2.6</td>
<td>Recursive Lisp Function Example</td>
<td>29</td>
</tr>
<tr>
<td>2.7</td>
<td>Recursive Function Call Example</td>
<td>29</td>
</tr>
<tr>
<td>2.8</td>
<td>General Function Calls in Lisp</td>
<td>30</td>
</tr>
<tr>
<td>2.9</td>
<td>Conversation Type in Lisp</td>
<td>31</td>
</tr>
<tr>
<td>2.10</td>
<td>Defining Variables at Run Time</td>
<td>32</td>
</tr>
<tr>
<td>2.11</td>
<td>Binding Variables to a Called Conversation’s</td>
<td>33</td>
</tr>
<tr>
<td>2.12</td>
<td>Binding States to a Called Conversation’s</td>
<td>33</td>
</tr>
<tr>
<td>2.13</td>
<td>Lisp Code for a Make Policy Call</td>
<td>35</td>
</tr>
<tr>
<td>2.14</td>
<td>Lisp Code for a Make Global Policy Call</td>
<td>35</td>
</tr>
<tr>
<td>2.15</td>
<td>An If statement in Lisp</td>
<td>36</td>
</tr>
<tr>
<td>4.1</td>
<td>Hierarchical Diagram</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>UMLet at Startup</td>
<td>52</td>
</tr>
<tr>
<td>5.2</td>
<td>Custom Element Design Interface</td>
<td>53</td>
</tr>
<tr>
<td>5.3</td>
<td>Mail Interface</td>
<td>54</td>
</tr>
<tr>
<td>5.4</td>
<td>UML Class Elements</td>
<td>55</td>
</tr>
<tr>
<td>5.5</td>
<td>UML Relation Elements</td>
<td>55</td>
</tr>
<tr>
<td>5.6</td>
<td>UML Custom Elements</td>
<td>56</td>
</tr>
<tr>
<td>5.7</td>
<td>CASAUMlet at Startup</td>
<td>56</td>
</tr>
<tr>
<td>5.8</td>
<td>New Conversation Type in UML</td>
<td>57</td>
</tr>
<tr>
<td>5.9</td>
<td>New Local Policy in UML</td>
<td>58</td>
</tr>
<tr>
<td>5.10</td>
<td>New Global Policy in UML</td>
<td>58</td>
</tr>
<tr>
<td>5.11</td>
<td>Instance Diagram Startup</td>
<td>59</td>
</tr>
<tr>
<td>5.12</td>
<td>Conversation Type and Delegate Function Calls</td>
<td>60</td>
</tr>
<tr>
<td>5.13</td>
<td>Running in Context of Agent</td>
<td>61</td>
</tr>
<tr>
<td>5.14</td>
<td>Action Level File Node</td>
<td>61</td>
</tr>
<tr>
<td>5.15</td>
<td>Containment Relations</td>
<td>62</td>
</tr>
<tr>
<td>5.16</td>
<td>Instantiation Relations</td>
<td>63</td>
</tr>
<tr>
<td>5.17</td>
<td>New Conditional in UML</td>
<td>63</td>
</tr>
<tr>
<td>5.18</td>
<td>Transition Relation</td>
<td>64</td>
</tr>
<tr>
<td>5.19</td>
<td>Equivalence Relation</td>
<td>64</td>
</tr>
<tr>
<td>5.20</td>
<td>New Comment Block in UML</td>
<td>65</td>
</tr>
<tr>
<td>5.21</td>
<td>Lisp Code Block in UML</td>
<td>65</td>
</tr>
<tr>
<td>5.22</td>
<td>Documentation Mouseover Text</td>
<td>66</td>
</tr>
<tr>
<td>5.23</td>
<td>Comment Mouseover Text</td>
<td>67</td>
</tr>
<tr>
<td>5.24</td>
<td>Speech Act Background Menu Popup</td>
<td>69</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>7.38</td>
<td>Test Request When - Send Request When</td>
<td>118</td>
</tr>
<tr>
<td>7.39</td>
<td>Test Request When - Captured Conversations</td>
<td>119</td>
</tr>
<tr>
<td>7.40</td>
<td>Test Request When - Client Sends Message</td>
<td>120</td>
</tr>
<tr>
<td>7.41</td>
<td>Test Request When - Client Receives Agree</td>
<td>121</td>
</tr>
<tr>
<td>7.42</td>
<td>Test Request When - Client Receives Inform</td>
<td>122</td>
</tr>
<tr>
<td>7.43</td>
<td>Test Request When - Client Side Mouse Over Text</td>
<td>123</td>
</tr>
<tr>
<td>7.44</td>
<td>Test Request When - Server Receives Request When</td>
<td>123</td>
</tr>
<tr>
<td>7.45</td>
<td>Test Request When - Server Sends Agree</td>
<td>124</td>
</tr>
<tr>
<td>7.46</td>
<td>Test Request When - Server Detects Event</td>
<td>125</td>
</tr>
<tr>
<td>7.47</td>
<td>MASVIS Project</td>
<td>127</td>
</tr>
<tr>
<td>8.1</td>
<td>Function Definition with Duplicate Required Variables</td>
<td>132</td>
</tr>
<tr>
<td>8.2</td>
<td>Duplicate Variables of Different Kinds</td>
<td>133</td>
</tr>
<tr>
<td>8.3</td>
<td>The contents of File1.lisp</td>
<td>135</td>
</tr>
<tr>
<td>8.4</td>
<td>The contents of File2.lisp</td>
<td>135</td>
</tr>
<tr>
<td>A.1</td>
<td>Request Conversation Lisp Representation - Parameters</td>
<td>146</td>
</tr>
<tr>
<td>A.2</td>
<td>Request Conversation Lisp Representation - Documentation and Global Policies</td>
<td>147</td>
</tr>
<tr>
<td>A.3</td>
<td>Request Conversation Lisp Representation - Delegate Calls</td>
<td>148</td>
</tr>
<tr>
<td>A.4</td>
<td>Request Conversation Lisp Representation - State Binds</td>
<td>149</td>
</tr>
<tr>
<td>A.5</td>
<td>Request Conversation Lisp Representation - Local and Non Local Variable Bindings</td>
<td>150</td>
</tr>
</tbody>
</table>
# List of Symbols, Abbreviations and Nomenclature

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>U of C</td>
<td>University of Calgary</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>MAS</td>
<td>Multi Agent System</td>
</tr>
<tr>
<td>ACL</td>
<td>Agent Communication Language</td>
</tr>
<tr>
<td>KQML</td>
<td>Knowledge Query and Manipulation Language</td>
</tr>
<tr>
<td>FIPA</td>
<td>Foundation for Intelligent Physical Agents</td>
</tr>
<tr>
<td>COUGAAR</td>
<td>Cognitive Agent Architecture</td>
</tr>
<tr>
<td>JADE</td>
<td>Java Agent Development framework</td>
</tr>
<tr>
<td>ARES</td>
<td>Agent Rescue Emergency Simulator</td>
</tr>
<tr>
<td>CASA</td>
<td>Collaborative Agent System Architecture</td>
</tr>
</tbody>
</table>
List of Definitions

**Utterance** A sequence of words that make grammatical ‘sense’.

**Performative** A kind of performance in regards to speech. An utterance’s representational verb: commanding, requesting, etc.

**Constative Utterance** An utterance aiming to be either true or false with no performative value.

**Performative Utterance** An utterance that has a certain type of performative value.

**Speech Act** An act of speech, see Performative and Constative Utterance.

**Template** A pattern used by text with a certain purpose.

**Conversation** A sequence of message exchanges.

**Protocol** A rule for responding to a condition/event. The rule’s antecedent matches the condition/event and its consequent defines what should take place.

**Policy** Similar to a rule but these separate the antecedent’s condition/event into the precondition (what must currently hold true) and the antecedent (an event descriptor that matches incoming events). The consequent remains defined as what to do after the antecedent and precondition have been shown to match, however, the postcondition has been added reflecting expected outcome (which may or may not hold).

**Speech Act Template** A template for code that builds a kind of conversation using a kind of performative.
**Action Template** A template for code that defines a concrete conversation. Depends on a Speech Act Template (that has been loaded into memory) and defines a name for the action and (sometimes) an initial reaction.

**Concrete Conversation** An action conversation that has been executed and defined as a *known* (available) conversation in memory.

**Instantiated Conversation** A concrete conversation that has been cloned and supplied values in response to an instantiating event and can itself respond to events.
1.1 Agents and Multi Agent Systems

To define an *Agent* is both a complex and simple thing to do. As pointed out in many works [Denzinger and Hamdam, 2004] [Tweedale et al., 2007] [Ferber, 1999] [d’Inverno and Luck, 2001] [Uhrmacher and Weyns, 2009], the definition varies depending on who is talking. Some sources hold that it is often satisfactory for the author in question to provide their own [d’Inverno and Luck, 2001] while most draw light to the common abilities that should be associated with an Agent. The following [Uhrmacher and Weyns, 2009] strives to be a general (though still somewhat prescriptive) list of features defining an Agent as “a software entity situated in a virtual or a real environment”:

1. that is capable of acting in an environment
2. that is driven by a set of tendencies
3. that possesses resources of its own
4. that has only a partial representation of this environment
5. that can directly or indirectly communicate with its environment
6. that may be able to reproduce itself
7. whose autonomous behaviour is the consequence of its perceptions, representations and interactions with the world and other agents
Mathematically, Dr. J. Denzinger (Denzinger and Hamdam, 2004) formally defines an Agent with the tuple Agent=(Sit,Dat,Act) where Sit is the set of situations the Agent can be in, Dat is the set of possible values the Agent’s internal data structures can have (sometimes referred to as the Knowledge Base), Act is the set of all actions the Agent can perform. The term fAgent describes the function “Sit × Dat → Act” which determines what action the agent will perform based on the current situation versus the Agent’s Knowledge Base.

These definitions show how very much the term agency changes to suit the purpose. For this thesis consider agents to be defined simply as software entities meeting some definition similar to those above with the purveyance that agents can interact directly with other agents especially in terms of addressable message exchange.

A Multi Agent System (or MAS) is, as the name suggests, a system in which multiple agents operate. Besides real world activities, the MAS paradigm has been recognized as a great test bed for simulations in both social (Uhrmacher and Weyns, 2009) (Axelrod, 1997) and scientific disciplines ranging anywhere from computation to biology. MASs can be distinguished from other methods of computation/simulation as they focus on indirectly creating and observing emergent behaviour of macroscopic systems (Uhrmacher and Weyns, 2009) by emphasizing the actions and interactions of the agents modelled (Ferber, 1999) (Taha et al., 2007). The environment in which the agents act is also considered to be a key component as it usually, either explicitly or implicitly, defines the actions agents can perform which can influence their behaviours and social structures. (Uhrmacher and Weyns, 2009) Simply put, the environment is something that either is part of (or a model of) a real or imaginary world and serves as an area facilitating Agent actions and/or interactions. Examples of MASs in computation ranges anywhere from web applications (BBN Technologies, 2004) (Tweedale et al., 2007), to speech recognition systems (Taha et al., 2007) (Erman et al.,...
1980], and, of course, multi agent robot systems [Ota, 2006] and their simulations [Denzinger and Kidney, 2005] [Uhrmacher and Weyns, 2009].

1.2 Design Considerations of Multi Agent Systems

Concrete applications of a MAS can be seen in expert systems such as BIICS [Linkens et al., 2000] which worked towards intelligently optimizing control of a cryogenic cooling plant. These types of systems implement various kinds of specialized programs that can work independently and usually behave collaboratively. Using expert systems we have achieved solutions to many complex problems as far back as the 1970s as with HEARSAY-II in the domain of speech recognition [Erman et al., 1980].

The kinds of problems that we give a MAS is an important point. It should be noted that the problem in question may or may not even be solvable, but in some cases it is expected that the agents eventually reach some sort of conclusion in regards to this. That aside, since the problem in question can sometimes be divided into smaller problem portions and later reassembled into a completed solution (or macroscopically observed as with biological simulations [Uhrmacher and Weyns, 2009]), a MAS can truly be useful in that the many Agents can go off in their own direction and discover possible solutions to the problem pieces (or simply interact with the environment for macroscopic observation). Additionally, we should be aware that the method in which the Agents interact can be anywhere along the spectrum from competitive to cooperative to collaborative [Ferber, 1999] [d’Inverno and Luck, 2001] [Uhrmacher and Weyns, 2009].

Competition has been defined in a variety of ways with the overlap being driven by some consideration of the agents’ goals. Should two agents have goals that conflict (i.e. are mutually exclusive such as with competitors running a race, only one can win) the agents are said to be in competition [Ferber, 1999] [d’Inverno and Luck, 2001].
Competitive agents usually compete to find the best solution to their given problem portion by whatever means they can employ, including delegating the work to other agents. The aim of such a scenario is that the end system can elect to keep only the best agents for future work or learn some classification of the agents such as which agents are better at solving which jobs.

Collaboration tends towards the other end of the spectrum where some (or all) of the goals held by two (or more) agents are not conflicting. When enough of the goals are not conflicting (through minimization or other bases) there is the option to form coalitions but only when the skills one agent (or agent group) already has are insufficient in dealing with the problem at hand. Examples of this can be found in the coalitions formed by political parties and biological systems when autonomy is given up by an Agent in exchange for a better chance at having some of its goals realized.

Cooperation occurs somewhere in between the last two. Some sources characterize cooperation through goal adoption implying that it is a kind of collaboration where others consider it to be the other way around. It has also been pointed out that some definitions only require a shared awareness of ‘acting together’ which fails to consider whether or not the cooperation is actually beneficial (and of course, means for measuring such a thing are presented) or even if the awareness (at least from the observer’s point of view) is really necessary to characterize this type of interaction. Like collaboration, entering a cooperation is considered to be a voluntary act (sometimes defined by a client-server hierarchy) where both agents stand to gain from the mutual possession of some goal.
Finally a mention of coordination should be provided. This notion applies whenever there are insufficient environmental resources [Ferber, 1999] such as limited network access time, when only one Agent’s solution can be submitted and evaluated at a time, or when the space in the physical/simulated environment constrains Agent movement (i.e. the space must be shared). A simple and general definition is provided in [Tweedale et al., 2007] as “the managing of dependencies between activities”. This type of interaction can be combined with any interaction scheme falling along the spectrum containing the three mentioned above and may of course not even be necessary if ‘resources’ are plentiful. In all cases, coordination requires some kind of knowledge exchange either through direct or stygmeric communication (such as leaving and reading signs in the environment) [Uhrmacher and Weyns, 2009] [Tweedale et al., 2007].

1.3 Why Agent Communications?

The outline provided regarding Agent interactions provides some insight into the design choices one must make in designing a MAS. For example, a leader Agent could be required in order to define how the problem will be divided, along with what parts are assigned to which contributors. It could also become evident later on that some of the parts are dependent on others, and so must wait for those dependent solutions before they can be completed. An example of such dependencies is seen in logical proofs, as later parts of a proof rely on the already proven lemmas for their solutions. In addition to this, some parts of the problem pieces assigned to agents may require some agreement in regards to what form the smaller solutions should take, so that they are compatible when forming the complete solution. Such possibilities outline the necessity of communications for Agents in a MAS at least in those cases where the interaction scheme requires it.
The focus of Agent Communications as an area of study is the investigation of how we might get the most out of communications between agents (whenever direct communication is used). Can we improve the knowledge exchange in our messages or even enforce better control amongst the agents? Already there are several competing paradigms for agent communications and these can be found in a diverse assortment of environments like banking transactions, email platforms and even in Massive Multiplayer Online Role Playing Games such as “The World of Warcraft”.

In addition to the competing paradigms are the competing message formats; how the messages are represented. The two most popular forms of messages in use are the Knowledge Query and Manipulation Language, KQML, and that developed by the Foundation for Intelligent Physical Agents known as the Agent Communication Language abbreviated FIPA-ACL [Ferber, 1999] [d’Inverno and Luck, 2001] [Uhrmacher and Weyns, 2009] [BBN Technologies, 2004] [Telecom Italia Lab, 2015] [Labrou et al., 1999] [Chaib-Draa and Dignum, 2002].

1.4 MAS design platforms

Given that these systems have been an active area of research across multiple disciplines for much of the modern age one might wonder what sort of Multi Agent System development environments exist. While there are a multitude, the following are worth noting:

1. **COUGAAR [DARPA, 2007]**. The Cognitive Agent Architecture “is a Java-based architecture for the construction of large-scale distributed agent-based applications”. This system focuses on what they call the Cougaar Component Model which allows a Cougaar agent to be tailored through the use of plugin components allowing the system to be run as anything from a single host application to a highly distributed one. Cougaar abstracts all message transport from the plugins with the intent that
the developers are free to concentrate on the domain-specific issues of their applications. Therefore Cougaar is less a platform for developing agent communications than it is for developing agent application systems.

2. JADE. The Java Agent Development framework is an open source platform for peer-to-peer agent-based frameworks. This system aims to provide a graphical interface for creating agent type hierarchies in which agents that are lower in the hierarchy implement all of the behaviours of their parent agents, as well as any additional ones defined by the user. The form in which agents communicate is entirely through messages implementing FIPA-ACL with no way to do otherwise. This means that Jade agents are incapable of implementing third party communication schemes. Jade is therefore more focused on developing FIPA-style agents and so (as with Cougaar) it is not a viable development environment for Agent Communications.

3. ARES. The Agent Rescue Emergency Simulator is a project providing a “simplified rescue scenario” simulation of a disaster event in which a group of agents explore a 2 dimensional grid environment rescuing as many survivors as possible. The goal of the project is to aid in teaching students the concepts of inter-agent interactions, agent-environment interactions, social hierarchies, and agent communications by providing a simple platform with which to develop the control logic of the simulated agents. However, the inter-agent communications is not the focus of this tool which is evident in the fact that the simulated environment is designed to challenge programmers to weigh all actions (cell-to-cell movement, communication delays, etc.) against the benefit to their team score. The desired platform for Agent Communications as a field of research does not require a real world simulator environment for its agents nor does it need a game aspect of saving survivors. Instead, the agents should have tools in place that already utilize as many varieties
and concepts in Agent Communications as possible in order for it to be considered a good medium for experimentation.

4. CASA [Kremer, 2009]. The Collaborative Agent System Architecture is a Java based development and testing environment for use by researchers and experimenters alike to create agent based systems. The system is highly customizable and can easily become unwieldy given the amount of flexibility available but by carefully balancing the design against ease of use any Multi-Agent organizational structure using any interaction scheme can be obtained. In that interest CASA supports all known Agent Communication languages/paradigms and endeavours to provide basic and reliable communication services such that the Agents developed are free to use any communication scheme they wish.

Given the flexibility and high degree of customizability that CASA offers, it is a clear choice for development in agent communication schemes. In CASA, communications are achieved through the use of conversations between agents. Conversations are the specification of shared social conventions used when exchanging messages such as acknowledging that a message was heard. There is, however, one downfall: In the beginning CASA implemented all communications in the Java language, which made analyzing them a difficult task due to the many components of a conversation being so distributed. Later on, the communication scheme’s implementation was moved to the programming language Lisp which brought all the components into one localized code base, but as can be seen in Appendix A they are still long, and perhaps not very intuitive to a new user that is not very familiar with Lisp.

I intend to provide a visual interface for observing, creating, and modifying conversations used in agent communication paradigms.
1.5 Contribution

I have provided a Graphical User Interface (GUI) with which one can observe, create, and modify conversations in CASA. This tool aids in the education, propagation, and development, of the Agent Communication concepts used in CASA and elsewhere. This motivation is supported by the fact that Agent communication protocols in CASA are notably difficult to read. The actions agents should perform based upon these communications are abstract, and the code in which they are written is complex and often verbose. This modelling tool helps facilitate the education and adoption of advanced communication paradigms by students, developers, and in the future industry.

To accomplish this goal I ensured that the tool I developed was both *useable* and *useful*. To be *useable* means that the tool can actually do what it promises. This simply means that the tool must have the ability to create, modify, and observe the conversations used by the Agents. To be *useful* is to be a tool that users will want to use; it must be easier to build Agent communications with it rather than by any other means. Thus, the graphical representations must be meaningful and the tool should provide methods for debugging any problems. In order to accomplish these requirements the following conditions were met:

1. For the graphical aspect we selected a representation that is both intuitive and familiar to the people using it. If it is not intuitive than it is unlikely to be adopted since the scheme is no better than any other representation. If it is not familiar then it is unlikely to be adopted since the users would have to learn this new representation scheme on top of the ones they already know, leading to a steeper learning curve. Meeting these two requirements provides a better guarantee that the tool is at least tried out.

2. The tool conveys as much information as possible with respect to the commu-
nication paradigm while not bombarding the user with an overcomplicated representation. Sources of information for visualization schemes includes colours used by the various elements, labelling, parenting (or encapsulation), and relationships between elements. Further, by keeping the number of information sources to a minimum we guarantee that the tool remains simple and thus intuitive as in (1).

3. The user is provided with all the tools necessary to create, modify, and observe conversations. Thus, every aspect that is used to control and define a conversation is configurable and the changes are verifiable through an editor window and/or through testing in an animation window.

4. The tool is compatible with a Multi Agent System Implementation. Since we have already said that CASA is the best choice for experimentation in this field and the problem therein is the basis for this research we leave the use of CASA as an assumption.

Keeping the above points in mind will lead to a visualization tool that is both easily adopted and speeds up the development process for users in the field of agent communications.

1.6 Objectives

Having listed in the previous section points that make this tool useable and useful allows for a list of the objectives to be given:

**Literature Review** I will study the existing communication system (CASA) in order to flesh out the important aspects of the Agent Communication paradigms being used.

**Develop Requirements** Accomplishing the Literature Review allows for the development of the system requirements. These describe possibilities for the graphical
representation: what it must represent, possibilities for the necessary user interactions, and what it should accomplish (such as making and saving changes).

**Develop Design** Next, I will provide, and in some cases expand upon, aspects of the design supporting the requirements. This includes selecting an appropriate graphical representation based on other projects that interface a graphical representation with code.

**Develop Implementation** Following declaration of the design, I will provide an implementation for all of the points made therein. I select a tool, based on simplicity, from those that allow for the creation of the selected graphical representation through code.

**Testing the Implementation** I will then test the implementation to validate its inner workings. Methodologies for testing are described along with how they were used and how they were helpful.

**Use Case Analysis** Once the system is in place and has been tested, I will give an example of its successful use along with an analysis of its performance.

**Conclusion** Finally, I will make a conclusion with regards to the work’s value. This includes a reiteration of the project’s goals and how they were met, a description of the benefits provided to CASA, drawbacks that were made due to compromises, and future work that could be done to improve the tool.

Thus when considering that these objectives have all been met we can deem the project complete and the goals accomplished.
1.7 Summary

In this Chapter, Agents and MASs were both defined and a basic overview of Agent Interactions in MASs was provided. The intended area of study selected is Agent Communications. It was noted that many MAS design platforms exist and that CASA [Kremer, 2009] has been selected as the Agent Communication design platform of choice. Finally, the objectives to be met were provided reflecting the overall outline of this work.
Chapter 2

Background

The concept of an Agent communication system is clearly necessary but it may seem trivial to initial inductees. After all, what does a communication paradigm need outside of message passing? To answer this I will first point out that the area of Agent Communications and in fact all Computer Communications as a whole goes as far back as the 1960s.

2.1 The Basis of all Communications

In 1955 John Langshaw Austin visited Harvard and delivered the William James Lectures. In 1962 the notes and tape recordings of attendees were posthumously collected into the book ‘How To Do Things With Words’ [Austin, 1962]. Presented in Austin’s work is the idea of performative utterances. Originally, philosophers of Linguistics held the belief that any sentence could be thought of as a statement that described some state of affairs which it must do either truly or falsely [Austin, 1962, Lec. 1]. Unfortunately, such a definition limits our understanding of certain sentences such as nonsensical ones, interrogatives or questions, and directives which express a command or wish. Austin describes the existence of performative sentences like ‘I promise to do such and such’ or ‘I nominate so and so for president’ which rather than attempting to describe the world aim to change it through the utterance. Further, sentences that appear to commit oneself to certain actions (such as saying ‘I do’ in response to ‘do you take this woman to be your lawfully wedded wife’) can be done so in bad faith but this hardly makes the utterance true or false. Instead, it

---

1White’s Professor of Moral Philosophy at Oxford
is the action associated with the speech that is important.

The type of action these sentences perform is usually a verb and what Austin refers to as the performative verb or simply the *performative*. Austin later encountered difficulties in describing performative verbs: tenses, first vs third person, tone of voice, usage; to distinguish them from constative sentences which, theoretically, do not seek to perform actions but are merely true or false statements. He therefore reanalyzed the acts of asserting something versus the acts of doing something versus its consequential outcomes and created three terms:

**Locution**

A sequence of phonetic tones or ‘noises’ (phones) together having grammatical meaning with respect to a vocabulary (phemes) while also having some ‘sense’ or ‘reference’ to something (rhemes). The Act of saying something about something else. [Austin, 1962, Lec. 8]

**Illocutionary Act**

The act of using a locution with a certain force; ordering, asking, promising, asserting, etc. i.e. in what way are we saying it. The Act in saying something. [Austin, 1962, Lec. 8]

**Perlocutionary Act**

The resulting consequential effects in saying something upon the feelings, thoughts, or actions of the audience and which may be done with design or intent; persuading, scaring, enlightening, etc. A locution’s actual consequences. [Austin, 1962, Lec. 8]

He eventually concludes that constatives (at least in most cases) are just an abstraction of illocutions since by making statements about facts in speech they usually aim to perform some act such as informing or declaring belief. He then states that all
utterances should be though of as an act of speech or a *speech-act*. Thus the definition of a speech-act in Austin’s theory demonstrates that locutions are usually better described by the terms illocution or perlocution. Therefore most utterances in language aim to do something as much as say something with respect to some appropriateness.

When it comes to Agent Communications, the Illocutionary force must be accounted for separately from the actual message (or utterance) because synthetic messages have no tone (as opposed to humans where the *way* something is said can be used to distinguish a request from a command). Additionally, since agents may engage in many conversations simultaneously a conversation identifier (sometimes numerically based) is specified. As a final example, since agents prefer electronic communication to sound waves (from which one could determine the speaker’s voice or location) there has to be a way to determine which Agent is talking in order to avoid errors. Thus, normally, sender and receiver fields are also supplied. This demonstrates some of the hurdles one must overcome for message design in MAS. The KQML language (and most others founded on message exchange) accomplish these (and other) necessities of error-free electronic communication through the use of key-value pairs. Thus, a message (in such a language) would contain keys like ‘performative’, ‘sender’, ‘conversation-ID’ appended by something like an integer, and ‘content’ each paired with a value.

The Perlocutions have to do with social norms. A finite number of things can happen after an Agent performs an utterance each of which can have its own meaning with respect to the context. In Agent Communications the types of responses expected and given can be modelled as a part of the communication scheme. While the actual response is determined by the responding Agent’s internal workings it still falls into some category that the receiver can use for future communication decisions or plans.
Finally, locutions are usually thought of as the actual message or message content. Assuming performatives are specified it can be noted that the locution does not need to contain sentence fragments like ‘I request’ or ‘I command’ nor anything else present as a key. This allows for smaller and simpler message contents.

Thus, any communication scheme can be seen as far more than just the exchange of messages as not only does a message need to give it’s content and context, but it also has certain desired results.

2.2 Applications to Agent Communications

Now the area of Agent Communications, specifically, has been developing since the 1980s (as stated in Chapter 1 §1.1) and has led to several competing paradigms. Generally, the goal in designing these paradigms has come down to two things which will now be discussed:

1. Keep the schemes flexible so as to encourage their adoption by popular MAS projects.
2. Maximize the effectiveness of the Agents’ communications.

Indeed good communication schemes should allow Agents made by different software teams and companies to be compatible. To facilitate such compatibility Don Mckay and Robin McEntire at Unisys Corp and Tim Finin and Richard Fritzson at the University of Maryland collaborated in 1994 to create the Knowledge Query and Manipulation Language (KQML)[Finin et al., 1994]. In KQML every message passed contains a number of key-value pairs. Such keys include the sender, the intended recipient, the performative, the actual message content as well as any others the developers wish to include, exclude, or define when using the scheme. Similarly the actual interpretation of such a message can be left up to the recipient Agent’s
developers meaning it may or may not evaluate all key-value pairs in every situation. The flexibility and versatility this scheme offers presents us with the simplified compatibility we require.

However, in consideration of the effectiveness of communication, we should expect that in any message exchange there is a certain amount of predictability as to what will happen. For example, social norms dictate that when people receive messages they also acknowledge them (note that aside from verbal feedback this can also just be a gesture like a nod, a facial expression, a movement, or nothing when the person behaves rudely). Furthermore, recall that, in regards to the performatives we use, when we perform utterances of the ‘request’ performative the number of possible responses should be finite. In this example, agree, not-understood, refuse, or please-wait are valid responses; however, keep in mind that both the types/numbers of valid responses and semantic meaning of the performatives is up to the developer. We therefore expect that agents follow some sort of protocol.

In Chapter 1 we mentioned FIPA-ACL as a message format in competition with KQML. Upon their creation in 1996, the Foundation for Intelligent Physical Agents built their scheme on top of KQML specifying a single additional parameter, the ‘protocol’. Though they offer little indication for the specific values this parameter should take they do point out that not specifying nor using it makes reliable communication a fairly difficult thing to achieve. This parameter can simply be viewed as a way to specify and control conversation flow by naming the kind of conversation the message containing it is a part of along with any other conditions such as when it is allowed to occur [FIPA, 2002].

This, however, may not make much sense until an analyses of the three main agent communication paradigms is described and how conversation protocol (or flow) can be achieved:
Figure 2.1: A Finite State Machine with a Start state, End state, Error state, and intermediate states. Any kind of event can be used to define the machine’s transition between states. These diagrams typically take a lot of space and should account for all possible events.

1. The ‘Ad-Hoc’ paradigm is one where each kind of conversation available is carefully scripted with the aim of accounting for every possible eventuality. That is, any message that initiates a conversation has a set of defined responses that either causes expected rebuttals, themselves having pre-defined responses, or results in the termination of the conversation. In reality however, the developers are only able to account for every conceivable eventuality rather than every possible one making errors very likely to occur. For example, if an Agent is switched off in the middle of a conversation and the other Agent’s script does not account for such a scenario then the system could crash and any currently generated solution or solution pieces could be lost. In addition, since these scripts can become very long and very complicated their use in large scale projects isn’t very desirable as it can become a great undertaking to make changes. An example of such scripted responses are the common state diagrams used by finite state automata like in Figure 2.1.

2. As mentioned earlier the desire to develop software standards for MAS resulted in the creation of the Foundation for Intelligent Physical Agents [FIP, 2014].
Figure 2.2: The general form of a Belief Desire Intention (BDI) engine based on the algorithm from [Wooldridge, 2000, Ch. 7]. If all desires have been met with respect to the Agent’s beliefs then the Agent can terminate, otherwise beliefs are updated and new desires may be created followed by new intentions that aim to make beliefs contain all desires. Lastly the actions reflecting the intentions are performed on the environment which is once again checked to see if all desires have been met.

These standards include the way Agents communicate. In that interest they adopted and modified the pre-existing KQML and developed Interaction Protocols (IPs) for communicative acts (CAs) to help guide the whens and whys agents communicate [FIPA, 2002] [FIPA, 2003]. The CA’s are formally specified by a modal logic based around the Beliefs Desires Intentions (BDI, see Figure 2.2) rule engine [FIPA, 2003]. In essence, BDI describes agents as holding beliefs regarding the world they are in and that they have desires they wish to make a reality. Out of the beliefs and desires the Agent generates a sequence of actions or ‘intentions’ that should hopefully lead to a world where its beliefs match its desires [Wooldridge, 2000 Ch. 7]. This system is, in theory, able to deal with unexpected occurrences. For example, if all Agents but one were suddenly destroyed the remaining Agent could take this new knowledge and adjust its set of intentions to match.

Unfortunately, mentalistic models for communication have problems and introduce concerns. In BDI it is required that all the beliefs a given Agent holds should always be true making any malfunctioning sensors or viral agents who lie (a discussion too great to go into here) a problem. Also, and perhaps most
relevant to this work, is that the state representing an Agent’s intentions can become extremely long and verbose. For example, an Agent can have an intention due to the fact that it holds the belief that another Agent holds a belief in regards to a third Agent’s beliefs and so on and so forth up to any number of agents. The FIPA Communicative Act specification [FIPA, 2003] overcomes this by defining a short hand for such ‘back-and-forth’ belief scenarios but intractability can still be introduced when considering how beliefs and intentions should be managed, how the semantic language’s defining code base should be implemented, or even in considering how to add relationships with third parties/societal groups [Poslad, 2007]. Thus, under this model, developers are not only bombarded by far too much information but they can also incur a significant amount of overhead (despite some loose suggestions made throughout FIPA’s specifications [FIP, 2014]).

3. This now brings us to 1991, where Munindar Singh [Singh, 1991] developed Social Commitment Theory and in 1995 Cristiano Castelfranchi [Castelfranchi, 1995] refined it further. A ‘social commitment’ (SC) can be seen as a contractual obligation generated by Agents as they interact. For instance, when one Agent sends a second Agent a request, the second agent is now committed to at least respond, whether it be affirmative or negative. In the affirmative case the second Agent becomes indebted to attempt to fulfill the request. In the negative case the first Agent understands that the second is no longer committed to taking any action. Social Commitments Theory has the intention of ensuring collective activity and cooperation amongst the Agents and has succeeded in this regard as it has provided a facility for guiding conversation flow by considering “communicative acts as part of an ongoing social interaction” [Chaib-Draa and Dignum, 2002]. Finally, the concept itself is notably simple and can be
added to any communication scheme with a relatively small amount of work.

2.3 Conversations in CASA

Now that an outline of the various communication control paradigms has been provided we will take a closer look at the MAS development environment we intend to use, CASA, and provide greater insight as to how communications are therein defined via a top-down approach.

To start, CASA has specified IPs as the behaviours for conversation types. The current types are listed in Table 2.1. These have been added in an Ad-Hoc style over the years as CASA has progressed. Of particular interest is the request conversation which has combined the notions of discreet requests (the ask conversation) and proposals (the offer conversation) due to a shared convention in their use. Reliability desires that after the other Agent agrees to our request (or our offer) their should be some verification that the requested action has been satisfactorily completed. In either case, a conversation regarding a proposal to discharge takes place. This is an example of how, in CASA, conversations can have their functionality combined with those of other conversations’ to create reliable communications. This modularity also allows for simpler code to be written [Kremer and Flores, 2015].

Each one of these conversation types is primarily geared toward a certain kind of performative. That is, the kinds of performatives it deals with are distinct in one

<table>
<thead>
<tr>
<th>Conversation Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ask</td>
<td>ask that some action be taken or whether some condition holds</td>
</tr>
<tr>
<td>offer</td>
<td>offer to perform some action on behalf of the receiver</td>
</tr>
<tr>
<td>discharge</td>
<td>ask to be relieved of some commitment to some action</td>
</tr>
<tr>
<td>request</td>
<td>combination of ask, offer and discharge useful for actions</td>
</tr>
<tr>
<td>subscribe</td>
<td>ask to be notified when or whenever some condition holds</td>
</tr>
<tr>
<td>query</td>
<td>ask for the value to which some parameter is assigned if any</td>
</tr>
</tbody>
</table>

Table 2.1: Conversation Types Defined in CASA
Figure 2.3: The subsumption lattice introduced by R. Kremer and R.A. Flores in [Flores, 2006] that arranges performatives by similarity of meaning and generality. The performatives of concern are taken from the FIPA standard on Communicative Acts [FIPA, 2003] and though they are already numerous there may be others yet realized. The ones currently used by CASA are defined in Table 2.2.

Dr. Robert Kremer, University of Calgary, and Dr. Roberto A. Flores, Christopher Newport University, collaborated to organized these into a subsumption lattice in [Flores, 2006] employing various realizations such as a ‘propose’ being a kind of ‘request’ and so on as seen in Figure 2.3. This allows the number of conversations required to cover these performatives to remain relatively small. Also, the amount of work required when a new performative must be named and added (for whatever reason) can be minimal when no new behaviour needs to be added.

Having adaptable behaviours aims to support developers in creating concrete conversations (discussed further in §2.4) with the required functionality in an error free way. For a conversation to be concrete is for it to name a user-friendly action (like ‘open the door’) and a reaction (like ‘wait for the door to open’) which is all performed
with some behaviour’s notion of social norms. Notice how this implies that a concrete conversation may be needed for both sides (client and server) of some conversations. This is because each side may need to perform different reactions. As in the example, a guard Agent (to whom another Agent has made the ‘open the door’ request) should now react by ‘checking security clearance’. Of course, there is more that can be customized with respect to employing a certain behaviour. Parameters can be modified such as what to do when there is a refusal (the guard ‘declines’) or a time-out (the guard ‘never responds’). Or perhaps the developer wishes that another performative be used for the initiating event in place of the default one (like ‘ordering’ the guard as opposed to ‘asking nicely’).

When a server Agent receives a message, it may be labeled as part of a currently executing concrete conversation. If the object to which it refers can be found, the message is pattern-matched against the policies in that conversation template and those matching policies are executed. If, however, the message is not labeled by an existing conversation, the system uses pattern matching against the agent’s global policies and executes those policies; possibly one of those policies will instantiate a new conversation. Assuming that a conversation is instantiated, the structure defined by its conversation type now controls the message exchange through tools like SC instantiation, fulfillment, and removal. However, in the case that no matching responses can be found, the default ‘not-understood’ message is returned as a last resort option.

When considering conversation behaviour as the controlling structure for message exchange consider a conversation concerning a requested action. Such a request can yield several results each of which requires some action to be performed. For instance, the server will notify the client when the requested action is successful, if it fails, or if it is deemed unreasonable. In these different scenarios the server should, preferably,
Figure 2.4: A sequence diagram showing possible message exchanges between a client and server. Possible outcomes include the server’s refusal (resulting in termination) or the server’s agreement followed by a proposal to terminate (to which the client either agrees or refuses).

give different kinds of messages each having some (though not necessarily specific) effect on both the client and server side conversations. See Figure 2.4 for a look at this message exchange and possible results (or branches).

This idea of different events affecting conversation progression in different ways now allows for the definition of a policy. Generally, policies control the conversation’s progression depending on the sorts of messages received though other events can cause their execution as well. They are mainly defined by four components: The antecedent, the event that the policy responds to, the precondition, statements that must currently hold true before the policy is executed, the postcondition, statements that (might\(^2\)) hold true after the policy is executed, and the consequent, the action(s) that the policy actually executes. The precondition is usually concerned with the current state that the conversation is in but could depend on any boolean expression coded.

\(^2\)The consequent(s) may not succeed or are incorrect
Policies come in two forms, local and global. *Local* policies operate locally within a given conversation as described in the previous paragraph. *Global* policies on the other hand are policies that can be seen and executed without the need of a conversational context, that is, there does not have to be an active instantiated conversation in order for the policy to be applied. For example, the ‘not-understood’ global policy can be applied when a client attempts to start a conversation the server doesn’t know how to engage in.

Another function of global policies is to act as a ‘first responder’ in which they *instantiate* the appropriate concrete conversation (clone it and supply the event) when the event’s details match the antecedent. While these global policies do not require an active conversation instance in order to execute they are usually defined within the conversation type describing the concrete conversation’s behaviour. Furthermore, note that when more than one instantiating policy is present the conversation can take on alternate forms. For example, the request conversation can be used to make either a discrete request, or a proposal as discussed earlier regarding the conversation types in Table 2.1.

Lastly, there are three labels that can be given to global policies: always apply, last resort, and ghost. The first is simply a policy that is always applied (like message logging) while the second is a class of policies that should only be used as a last resort option (like when having to send the ‘not-understood’ response). The third option, however, can be used in conjunction with either of the other two and identifies policies that are hidden from execution. So, although ‘ghost’ policies execute they do not identify as a policy that actually *handles* the relevant event. For example, whenever a message is received the contents could be logged but that is a very different thing when compared with responding to the message.
2.4 CASA Conversation Lisp Definitions

Before concluding this chapter, an in-depth look at how these conversational types, templates, and policies, are defined in Lisp will be given. Note that ‘Lisp’ stands for ‘List Processing’ in that its goal is to evaluate lists and lists within lists etc. A list is denoted by a pair of rounded brackets with some number of symbols inside it (i.e. \((\textit{symbol} \ \textit{symbol} \ \textit{symbol} \ldots)\) where the rough meaning here is to apply the first symbol to the remaining symbols as a function). Lisp has many built-in functions that can handle an arbitrary number of symbols (i.e. \((+ \ 1 \ 2 \ 3)\) returns 6) and a Lisp coder can, of course, write similar functions with a relative amount of ease. The topic of coding in Lisp will for the most part not be covered and any relevant cases required by later points of discussion will be kept minimal.

2.4.1 Lisp Overview

There are two things in regards to Lisp that will need to be understood before continuing: function definitions and function calls.

A function definition in Lisp (**defun**) has the format shown in Figure 2.5.

First, consider the parameter list occurring at the start of the function definition immediately after the **function-name** parameter. The following outlines the parameter types defined by GNU Emacs Common Lisp Emulation [GNU, 2015].

1. Required: All parameters that occur before the **optional** flag are required to invoke the function it defines.

2. Optional: All parameters that occur after the **optional** and before the **rest** flags are optional when invoking (they do not have to be supplied).

3. Rest: One parameter can occur after the **rest** and before the **key** flags. This parameter holds a copy of everything else passed to the method invocation after
(defun function-name
  (RequiredParam1
   RequiredParam2
   &optional //optional parameters
   opt1
   &rest //remaining parameters after optional
   rest
   &key //overridable parameters
   key1
   (key2 key2defaultValue)
   &aux //auxiliary parameters
   (aux1 aux1defaultValue)
   (aux2 aux2defaultValue)
)
  "documentation regarding what this defun does"
  ...List of commands...
)

Figure 2.5: Lisp Code for a Function Definition: bolded text in these figures indicates keywords while quoted text are strings

all of the required and optional parameters.

4. Keyed: All parameters that occur after the key and before the aux flags are keyed parameters. These parameters may have default values associated with them otherwise Nil (Lisp for null and False) is selected. These can also be overridden by naming the key to override and supplying the desired value.

5. Auxiliary: All parameters that occurs after the aux flag are auxiliary. These parameters should have default values associated with them as none can be provided by the method invocation. Failing to provide default values causes Nil to be selected. These parameters tend to either depend on other parameters passed by the invocation or simply serve to hold important values (such as PI or Euler’s Number).

Notice that (when such variables are declared) the rest parameter would contain all of :keyed-parameter-1, value-for-keyed-parameter-1, :keyed-parameter-2, and override-value-for-keyed-parameter-2 as defined by the Lisp language (method invocation is
(defun recursiveAdder
  (a &rest b)
  (+ a (if b
       (apply #'recursiveAdder b)
       0))
)

Figure 2.6: A Recursive Adding Function in Lisp

(recursiveAdder 1)
=> 1

(recursiveAdder 1 2)
=> 3

(recursiveAdder 1 2 3)
=> 6

Figure 2.7: Recursive Function Call Example

Calling the Recursive Adding Function

discussed next). Recall that the intent of the rest parameter is to contain all the remaining parameters aside from the required and optional ones. This can allow one to write tail recursive functions like the one in Figure 2.6 or simply to keep a record of all the symbols passed (aside from those evaluated as required or optional ones of course).

Now consider the string of text present after the parameter list. This is simply a documentation string that informs users as to the purpose of the function definition. Immediately after the documentation, however, is the actual list of commands that the function performs when invoked.

To invoke the function given in Figure 2.6 one needs only specify the name given to the function definition and the list of parameters as shown in Figure 2.7.

To demonstrate the use of all parameters types that are available to a method call, refer to Figure 2.8 as it applies to the general defun given by Figure 2.5.
(function-name
  parameter-for-required-parameter-1
  parameter-for-required-parameter-2
  parameter-foroptional-parameter-1
  parameter-foroptional-parameter-2
  :key1 value-for-keyed-parameter-1
  :key2 override-value-for-keyed-parameter-2
)

Figure 2.8: Lisp Code for a Function Call

Notice that the name of the defun (the ‘function-name’ parameter) is, again, supplied first followed by the required, optional, and key parameters. A common error in Lisp (as a consequence of specifying both optional and key parameters) is that when one intends to specify the key parameters they must specify the optional parameters as well. Failing to do so will cause both the key names (prepended by a colon) and their values to be evaluated as the optional parameters. Also notice that in the conversation call given by Figure 2.8 no auxiliary parameters are supplied values. This is because these parameters cannot be overridden in a function call and that there values may instead depend on other variables. These parameters are presumably used elsewhere in the function.

Now that the Lisp code has been defined for function definitions and how to call them a look at how CASA uses Lisp to create Conversation Types will be given.

2.4.2 Conversation Types

Now that a short explanation on function definitions and function calls has been given, an overview of CASA’s Conversation Types can be given. Recall that these are the kinds of conversations agents engage in and that they are designed to provide behaviour for certain types of performatives or speech acts. These are defined using Lisp function definitions (defuns) and generally have the format shown in Figure 2.9.

Notice that the commands executed when invoking such a function definition
(defun type-name
  ( ... parameters ... )
' ' documentation regarding the behaviour'
(agent . put-policy
  (policy
   // a policy handling conversation instantiation
  )
)
... other global policies...
(conversation     // the actual declaration
  conversation-name     // the runtime name
  (list     // included elements
    (policy ...)
    (conversation call ....)
    ... more policies/conversation calls ...
  )
:bind-var ...     // variables declared at runtime
:bind-var-to ...   // declare variable bindings
:bind-state ...    // declare state bindings
)
)

Figure 2.9: Lisp Code for a Conversation Type: bolded text in these figures indicates keywords while quoted text are strings
begins with the creation of any global policy(s) that instantiate the conversation (clone it and supply the event). These will be discussed later in §2.4.4.

The last section is the section that actually creates the concrete conversation as a clone-able object instance using the parameters supplied by the method call (discussed next in §2.4.3). Furthermore, any defun that does not contain this section of code is not considered to declare a conversation type. The contents of this block of code defines the name to use for the conversation (perhaps dependent on event details or supplied parameters), any controlling policies (again, to be discussed in §2.4.4), any delegate conversation calls (covered next in §2.4.3), as well as three types of bindings:

1. :bind-var. Declare a variable at run time. When the conversation object has become instantiated, parameters from the instantiating event can be stored by these variables and used by their policies or delegate conversations. (see Figure 2.10)

2. :bind-var-to. Declare that a parent conversation’s variable be bound to a delegate conversation’s. The delegate and the parent will both maintain the same value for the bound variable. (see Figure 2.11)

3. :bind-state. Declare that a parent conversation’s state be bound to a delegate conversation’s state. A change in state in the delegate corresponds to the state change in the parent that respects this binding. (see Figure 2.12)

Now that the Lisp code for a conversation type has been described we will now discuss the two relevant applications for a call to such functions.
2.4.3 Conversation Calls

Notice that the main point omitted by the previous section was an explanation as to what it meant to call another conversation or create a policy from within the context of a (defun)’s conversation code block. These are, of course, function calls that create more instances of clone-able objects. However, the objects created are contained by the concrete conversation object the defun creates when invoked. This section will cover both function calls that create delegate (or child) conversation objects as well as those that create top-most concrete conversations. Note that ‘top-most’ is preferred to the term ‘parent’ here since delegate conversations can have their own children, which can be delegate conversations that have their own children, etc. The Policy objects created by method calls will be discussed next in §2.4.4.

Conversation calls create what has been referred to as concrete conversations when
their code (a function call) is executed. These conversations serve a particular means depending on the context in which it is executed. Thus, a concrete conversation can have two functions:

1. Stand Alone. A stand alone conversation is a known conversation that has been created by performing a Lisp function call outside of any other conversation type’s context. Such a conversation call usually defines the name (a user friendly action-name the conversation performs like ‘open-the-door’) an action to perform when activated (a java method call, Lisp code, or nothing at all) as well as any other parameters that are either required, optional, or overridable. When an event triggers this conversation to be activated (by matching to one of its global policies) the object is cloned (along with its child objects) and instantiated with the event.

2. Delegate. A delegate conversation is a part of a known conversation. The parent conversation determines all parameters passed to the delegates and performs bindings to its variables. When the parent has been created as a known conversation its children are also created as known conversations that the parent can engage in. When a parent clone has been activated so too are the cloned child delegates (or sub conversations).

Refer to Figure 2.8 for the form of a function call. For the remainder of this thesis, code that creates a ‘Stand Alone’ concrete conversation through a function call will be referred to as a Conversation Template.

Now that an explanation of function calls (concrete and delegate conversation calls) has been provided an explanation of the two types of policies will be given.
(policy
    (antecedent code)
    (consequent(s) code)
    "documentation on what this policy does"
    :precondition (conditions that should be met before execution)
    :postcondition (conditions that should be met after execution)
    :name "policy-name"
    :ghost T or nothing
)

Figure 2.13: Lisp Code for a Make Policy Call

(agent.put-policy
    // a single Make Policy call
    :last-resort T or :always-apply T or nothing
)

Figure 2.14: Lisp Code for a Make Global Policy Call

2.4.4 Policies

Figures 2.13 and 2.14 contain the two kinds of policies we may encounter. Notice the
difference between a policy (which itself is just a function call) and a global policy
which simply wraps the policy in a specialized agent.put-policy command. The global
policy wrapper is what allows the policy to be instantiated as a ‘first responder’ even
though the conversation it is a member of has not been instantiated (though it is
present in memory as a known/concrete conversation). Thus, when such policies are
declared within a conversation type (a defun) it is usually the case that they are what
respond to the event(s) that cause the type’s concrete conversation object to become
cloned and instantiated. When occurring outside of any conversation type definition
however, the policy usually responds to events for purposes such as logging (as with
‘ghost’ policies) or when an event cannot be applied to any known conversation (as
with ‘last-resort’ policies).

Next, a look at a misleadingly simple piece of Lisp code will be given: the condi-
tional.
2.4.5 Conditionals

Conditionals like the one shown in Figure 2.15 may seem unimportant at first, however when they are used within the context of a conversation type (defun) they can affect both the structure of, and the access to, the concrete conversation.

For instance, they affect the structure when present in the list function of Figure 2.9 like in a request conversation (defined by Table 2.1) which can be executed in one of two ways: as an ask (discreet request) or an offer (a proposal). If it is desired that the concrete conversation disallows proposals then the conversation template must set the parameter nopropose to true. Within the request defun’s list function there is a conditional with regards to that parameter:

\[
\text{(if nopropose} \\
\quad () \quad // do nothing if True \\
\quad (offer .....) \quad // allow a sub conversation ‘offer’ if false \\
\text{)}
\]

Such a conditional can specify that another conversation call be present (as is the case here in the request conversation) or can also specify different policies be available under different conditions. Since this condition affects the resulting structure of the concrete conversation it is also included as an important piece of code when it occurs within the context of a conversation type’s function definition.

Next, in considering conversation access, note that such a conditional can also be used to control the available instantiating policies. The request defun also does this, specifying that if nopropose is true then the global policy that instantiates the conversation as an offer will not be available.
Any other uses of a conditional will be ignored and considered simply as code fragments that do not affect conversation structure nor access.

2.4.6 Stray Lisp Code

Code fragments that do not affect conversation structure/access must be accounted for. Some of these fragments are simply pieces of other code sections already mentioned: the parameter values used by conversation types in Figure 2.9, the precondition, postcondition, antecedent, and consequent codes of a policy in Figure 2.13, the actual condition used in Figure 2.15 which could be any combination of logical conjunctions or disjunctions. However there may, of course, be other code fragments that do not occur within any of these contexts nor fits any of the given definitions. The purposes of these may be outside the realm of CASA’s communications schemes and so they will not be modelled in any specific way and instead left as code fragments.

2.4.7 Comments

In Lisp there are two types of comments: single and multi line. Single line comments are comments that indicate that an entire line of symbols is not considered code and are preceded by the semicolon ‘;’ character. Multi line comments state the same thing for multiple lines and are preceded by the symbols hash and bar ‘#|’ and succeeded by the symbols bar and hash ‘|#’.

Comments can occur anywhere in Lisp code both outside of and within the definitions provided. Comments have the intention of documenting sections and/or fragments of code for those users who wish to code in Lisp directly or simply as an additional source of reference.
2.5 Summary

What has been presented here is the origins of Agent Communications from Austin’s theory of performative utterances followed by an overview of the different strategies used to create protocols for and conversations within Agent Communications. The strategies mentioned included the Ad-Hoc scripting methodology, FIPA’s mentalistic BDI semantics, and Singh and Castelfranchi’s Social Commitment theory. A brief introduction to Lisp functions definitions and an overview of CASA’s communications framework was provided which drew attention to its conversation and policy objects (along with other relevant pieces of code) and how they are defined in Lisp.

This provides the reader with an outline of the type of system this project models, the capabilities it must have, and the challenges involved in adequately presenting it in a graphical, and modifiable, form.
<table>
<thead>
<tr>
<th>Table 2.2: List of the FIPA communicative acts</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Accept Proposal</strong></td>
</tr>
<tr>
<td><strong>Agree</strong></td>
</tr>
<tr>
<td><strong>Cancel</strong></td>
</tr>
<tr>
<td><strong>Call for Proposal</strong></td>
</tr>
<tr>
<td><strong>Confirm</strong></td>
</tr>
<tr>
<td><strong>Disconfirm</strong></td>
</tr>
<tr>
<td><strong>Failure</strong></td>
</tr>
<tr>
<td><strong>Inform</strong></td>
</tr>
<tr>
<td><strong>Inform If</strong></td>
</tr>
<tr>
<td><strong>Inform Ref</strong></td>
</tr>
<tr>
<td><strong>Not Understood</strong></td>
</tr>
<tr>
<td><strong>Propogate</strong></td>
</tr>
<tr>
<td><strong>Propose</strong></td>
</tr>
<tr>
<td><strong>Proxy</strong></td>
</tr>
<tr>
<td><strong>Query If</strong></td>
</tr>
<tr>
<td><strong>Query Ref</strong></td>
</tr>
<tr>
<td><strong>Refuse</strong></td>
</tr>
<tr>
<td><strong>Reject Proposal</strong></td>
</tr>
<tr>
<td><strong>Request</strong></td>
</tr>
<tr>
<td><strong>Request When</strong></td>
</tr>
<tr>
<td><strong>Request Whenever</strong></td>
</tr>
<tr>
<td><strong>Subscribe</strong></td>
</tr>
</tbody>
</table>
Chapter 3

Requirements

Now that background has been provided and the Lisp code that facilitates communications in CASA has been defined, the system can be described.

Note that all the Lisp code in figures 2.9 to 2.15 is what this project aims to represent. It is desired that users can visually edit the code, and save the visual representations back to the original file as these kinds of code. The previous chapter also mentioned that there may be other pieces of code that do not fit any of those representations but must still be accounted for.

Lastly, to facilitate debugging, we desire that the system provides an animation of the active, or instantiated, concrete conversations showing their current state, their execution flow, and the events that caused the flow of execution.

This allows us to now explore this system’s requirements. We, however, do not intend to change how any of the communication concepts are represented by Lisp. This requirement will be left as an implicit one and will be discussed later on in Chapter 8§8.1.3 on future work.

Note that what is presented in this Chapter will be kept brief and that the full break down of the requirement points can be found in Appendix E.

3.1 Visualization Requirements

This section lists the requirements that the ideal visualization tool should have.

The Graphical Representation shall provide a visual representation of Lisp code that is easy to understand by new users, particularly those with a background in Computer Science. This can be accomplished using any 2D/3D models such as graphs,
trees, UML diagrams, or anything else familiar to Computer Scientists. (see Vis - 1 Graphical Representation).

Different Diagram for Different Uses means that we provide a clear disassociation between the various activities one engages in when using the software. Requirement Vis - 2 Different Diagrams for Different Uses and its sub points dedicate one diagram for editing Speech Act level elements (where conversation types can be created, modified, deleted, and saved into a Lisp file), one for editing Action level elements (where templates utilizing conversation types to create concrete conversations can be created, modified, deleted, and saved into a Lisp file), and an Instance Viewer for observing the progression of conversations an Agent has participated in or is currently participating in.

The Objects to be Represented are defined by Requirement Vis - 3 Objects to be Represented which explains how to represent the various aspects of the Lisp code in a distinguishable manner. This includes explicit pieces of Lisp code that define things like conversation types, policies, conversation templates, conditionals, and stray comments/Lisp code, all of which shall contain the relevant pieces of Lisp code that makes up their definitions (parameters, documentation, accessibility, etc.). This also includes the abstract ideas implied by the code including state transitions, equivalent variables (shared between parent and child conversations), containment (when a piece of code is part of another’s definition), declaration order, and file membership. Also, the visibility of these objects must be configurable so as to not bombard the user with possibly irrelevant information. To achieve this, any combination of colours, relations, emphasis typography, pop-up dialogues, numbering, labelling, content text, and mouse over text can be used.

The User Interaction requirements are defined by Vis - 4 User Interaction and simply state that of the objects represented, the user is free to: create, modify and
remove them; and change their position of declaration. These points can be achieved through pop-up dialogues, menus, or mouse/keyboard interaction.

3.2 Error Reporting Requirements

The *Error Reporting Requirements* defined by Appendix B §B.2 simply states that the user is alerted to any errors present in the code sections of diagram elements. This includes context checking such as undefined/redefined variables as well as malformed Lisp code (as with unmatched parentheses). However, if the error is due to something like an undefined variable (as opposed to malformed Lisp code) the user will only be alerted to the fact and still allowed to save the code. This is because the error reporting mechanism is meant to inform the user with regards to utilizing the concepts rather than imposing them in any strict way.

3.3 Saving Requirements

*Saving Requirements* are defined in §B.3 of Appendix B. As mentioned in the previous section, there is the requirement that the visual representation of the code be transformable back to code and saved for future use by CASA. For this, a user-friendly file saving mechanism needs to be in place that simply prints the code to the desired filename. The code it prints shall come from a transformation that uses knowledge of the kinds of Lisp code to create consistent and human-readable code.

3.4 Animation Requirements

Lastly, the *Animation Requirements* are defined in Appendix B §B.4. Since the tool shall also aid in testing as part of conversation development it is therefore necessary that it provides an animation of the conversations that agents are engaged in.
Therefore a mechanism needs to be in place that monitors conversations that an
Agent has engaged in, collects the sequence of changes to each conversation as a
list of events or ‘keyframes’ [Blender Foundation, 2015], and illustrates each change
between keyframes in a time-delayed manner.

3.5 Summary

This Chapter gave an outline of the top level requirements of the system such as
the requirement that the visual representation shall be recognizable to computer
scientists. The key requirement of the representation was, however, that it must
visualize each piece of Lisp code used in a Lisp file as well as make obvious the
abstract concepts associated with CASA’s use of said code.

Different uses for the representation were mentioned including template creation
and editing, concrete conversation creation, and instantiated conversation animation
where the first two required file reading/writing and error reporting for when the user
creates code that appears inconsistent or malformed. The last point merely required
a notion of ‘keyframes’ popularly used by animation modelling tools like Blender
[Blender Foundation, 2015].

All of these requirements were expanded into sub points and possibilities for each
one’s design were given, vaguely, leaving room for flexibility in Chapter 4: Design.
Chapter 4

Design

This Chapter describes designs that meets all of the points in Chapter 3 while leaving the actual implementation open for possibilities to be discussed in Chapter 5.

4.1 Selecting an Appropriate Scheme

To start, I will first attempt to address requirement Vis - 1 Graphical Representation which states that the representation shall be easily adoptable by persons using the software, namely, those working in the area of Computer Science. Graphs and trees are tempting as they can represent connected nodes in a hierarchical format (to satisfy requirements Vis - 3.6 File Membership Indication and Vis - 3.8 Child Containment Order) and their relations can be labeled (satisfying requirements Vis - 3.9 Conditional Representations and Anim - 3 KeyFrame Markups) like the transitions in Finite State Machine diagrams. Recall Figure 2.1 for Finite State Machines and see Figure 4.1 for hierarchical diagrams.

These could work since a conversation and it’s child policies can be distinguished through the use of different nodes and their relationship can be made visible with simplified contains relations. Additionally, the notion of transition relations aids in describing the transitions between the conversation states. Further, modifying such diagrams can be accomplished by allowing for the addition and removal of child nodes and the modification of their contents.

However, we also desire that the nodes can easily display pieces of information associated with an object in the software engineering sense. This would include the class of the object and key data members. For example, a policy would indicate that
it was a policy and would have a section to display its precondition, postcondition, antecedent, and consequent. The most common and widely used type of diagram in software engineering meeting these purposes is the Unified Modelling Language or UML. In particular UML class diagrams are well suited for containing all the relevant information of a class object. On top of this the format is just as well known to developers in the field of Computer Science and has been used in a similar way to generate code from UML [Vera et al., 2012] and as a debugging aid through augmented reality interfaces [McIntosh and Hamilton, 2010]. Therefore I will choose to use UML based diagrams but combine them with the hierarchical structure of tree diagrams and the state transitions of Finite State Machines.

This selection gives the design choices presented in Appendix C which here will be briefly described.

4.2 Design Choices

Design **Diagrams - 1 Different Diagrams** supports all requirements in **Vis - 2**

**Different Diagrams for Different Uses** simply by having separate diagrams for each of the three cases: speech act editing (for creating new behaviours), action editing (for creating templates for concrete conversations), and animation viewing (for past or presently engaged in conversations).

Design **ElemDefs - 2 Element Definitions** supports all requirements in **Vis - 3**
**Objects to be Represented** by describing how each kind of UML element in each diagram will be represented through either UML class elements/relations or some other means available such as mouse over text. Thus it describes the labelling, contents, and colour constraints of each element that reflects code, objects, and concepts. This includes all pieces of code and objects: conversation types, policies, instantiated conversations, sub conversations/concrete conversation templates, the agent (when running within an agent context), the relevant files that elements are a member of, conditions that affect the accessibility of concrete objects, stand alone comment sections and lisp code fragments. This design also includes the elements that represent both abstract and implied relationships/concepts as well as the methods by which they do so: containment by a parent, order of containment by parent, state transitions, equivalence of bound variables, edit-ability of nodes, and documentation versus documenting comments. Lastly, this design also specifies how any of the UML elements can be removed from view so as to assist the user in keeping the diagrams clear.

The specific menu designs for each diagram are given by **TempDiagMenus - 3**, **Speech Act Diagram Menus**  **ActDiagMenus - 4 Action Diagram Menus** and **InstDiagMenus - 5 Instance Diagram Menus**. These simply outline the various tasks the user can perform in each diagram. In the speech act editor one can: open/add/remove conversation types, stray comments, and stray Lisp code blocks; collect and save all the elements to a file; import other files in order to reference their conversation types as sub conversations; change the types file being edited/added to; and add/remove/reorder nodes that are contained by the file or other nodes. The action level editor allows the user to: add/remove/hide/reorder concrete conversation templates, stray comments, stray Lisp code blocks, nodes representing files; and open/create/save the files that contain these kinds of Lisp code. The instance viewer
simply allows the user to play and reset animations and change the speed with which
the animations play.

Next, design [EditNodeContents - 6 Editing Node Contents] describes the
differences in how one can edit nodes in the three diagrams. All nodes in the speech
act editor can have all of their contents modified with the exception of sub conver-
sations and their child nodes. Sub conversations may only have the values passed to
their parameters modified as they represent a function call rather than a definition
(despite the fact that the complete function definition is shown for these conversa-
tions). Similarly, the templates that define concrete conversations in the action level
editor may only have their parameter values modified instead of the parameters them-
selves added to or deleted. The instance viewer does not respond to any changes made
to its nodes as these nodes represent concrete conversation clones that have either
already terminated or have simply already been instantiated and thus are already
‘underway’.

Design [ParseChanges - 7 Parsing Content Changes] aims to fulfill the re-
quirements when evaluating changes to the contents of nodes. Simply put, the changes
are updated in the data structures of the objects whenever the error checking that is
performed does not reveal errors.

Design [NodeVis - 8 Node Visibilities] describes the utilities available to the user
such that the diagrams do not bombard them with two much information. Parent
nodes (nodes that contain other nodes) are expandable, collapsible, and closeable.
Closing a node removes it from the diagram and from memory however, in the action
level editor, it is only hidden in order to relieve the user of having to reopen files that
they may have accidentally closed. All three diagrams access the same content filter
in which the user can choose to hide certain kinds of UML elements (either a specific
UML class, UML relation, or mouse over text) so that hiding a type of object in one
The design for Animation given by [Anim - 9 Animations] expands on the animation requirements by specifying how the keyframes are defined, created, and used. The Observer Design Pattern [Lethbridge and Laganiere, 2005, Sec. 6.6] will be used to observe events. The keyframe objects will simply contain the event the conversation responded to at that frame of execution, as well as which policies responded. The policies will use their containment relation to display the event’s message content and the next state that the conversation should transition to (provided a transition takes place). The states listed in the conversation node will receive some kind of emphasis typography (bolding, highlighting, etc.) when the conversation is in the corresponding state.

The points in design [ParseNCapture - 10 Parser And Comment Capturing] define the parser and comment capturing heuristics in which they specify: the high level details associated with parsing Lisp code and how the parse trees should form; the methods used for recognizing the types of code (where anything not matching these types is considered ‘stray’); and how comments become associated either with nodes, code fragments, or nothing (in which case they too are ‘stray’).

Finally, the points in design [PrettyPrint - 11 Pretty Printing] define the methods for outputting human readable code. The code is assembled by recursively descending the object trees associated with the diagram elements, wrapping the code of child nodes within those of the parents’. Some simple rules are obeyed such as: inserting a space to separate all symbols; specifying a maximum line length that also imposes indentation whenever a newline is inserted; following code patterns for the recognized types of Lisp code; correctly placing comments so they associate with the correct nodes/code fragments; ensuring that every single line comment is followed by a newline so as to avoid commenting out any code occurring afterward. This concept
is similar to code style formatting used in IDE’s like Eclipse whenever new code needs to be generated automatically.

4.3 Summary

In this Chapter, a visualization medium was chosen based on its simplicity, recognizability to students of Computer Science, and its successful use in the projects User Interface and Navigation Modeling Methodology for Mobil Hypermedia Systems [Vera et al., 2012] and X3D-UML: 3D UML Mechatronic Diagrams [McIntosh and Hamilton, 2010]. What was selected is a conglomeration of hierarchical and Finite State Machine diagrams centred around UML class diagrams.

Next, a brief overview was given on how all of the requirements listed in Appendix B have been met and expanded upon in Appendix C. For example, what the UML forms of Lisp code objects would contain along with the various relationships between them and how to add/edit/remove them. This chapter then described the menu options for the various diagrams and how the diagrams treated certain objects and actions. Next, how file parsing for Lisp to UML along with the comment capturing heuristic would be achieved were both identified as points for the design that needed to be addressed. Finally, this Chapter finished off with how the design facilitates the output of human readable code.
Chapter 5

Implementation

As determined in Chapter 4, we will be using UML diagrams for the graphical representation of the Lisp code and want to be able to use mouse and keyboard event capturing on the elements represented. Therefore we must select a tool that is good for these purposes and is at least interface-able with Java, if not written in Java, since CASA is a Java based platform.

5.1 Implementation Options

There are several options available to accomplish this implementation which can be broken down into the following two examples:

**JOGL**

Java Binding for the OpenGL API [Gothel et al., 2013] is a wrapper library for the standard Open Graphics Library Application Programming Interface. It allows one to draw any shape within the window, write any text anywhere, supports mouse event capturing, and can integrate with Java’s application development frameworks. The downside of such an approach is that there is a lot of overhead that must be done in order to create the desired representations reliably (putting text in the desired place, capturing events on specific elements, etc). This would be the same for any library that gave us direct control over the graphics.

**UMLet**

UMLet [Auer et al., 2015] is an open source UML drawing tool that interfaces
well with Eclipse (a popular Java IDE). The fact that it is open source means that the API (all of the actual code used) is exposed to us and editable should we desire to change any behaviour. UMLet provides a simple means with which to create custom UML elements through simplified draw functions and also allows for drag and drop addition of elements from palettes to the diagram space. These elements extend Java’s `Components` class, meaning that mouse interactions and position settings can easily be manipulated and extended. Additionally, one can have several named diagrams open simultaneously and with an exposed API distinguishing between them and changing the contents programmatically is relatively easy to do. Most UML creation tools would most likely have many of the same features but without an exposed API it would be nearly (if not completely) impossible to interface their elements with our own code.

Thus we have chosen to use UMLet, which I will now provide an overview of, followed by a discussion on how the design choices from Chapter 4 are supported using this system. I will then follow with a description of the various classes I am using and how each one interfaces with UMLet.

5.2 Overview of UMLet

As mentioned in 5.1 UMLet is an open source java based platform aimed at the design and sharing of UML diagrams for software engineering purposes. What follows is a discussion of UMLet features as seen in figures 5.1, 5.2 and 5.3.

UMLet provides a window for the editing of multiple diagrams simultaneously. The Extensible Markup Language (XML) is used to represent these diagrams and the corresponding XML files are held in a location of the user’s choosing. The user can adjust the zoom level of these diagrams and can search for elements that contain a piece of text (resulting in their becoming highlighted).
UMLet’s palette panel (see Figure 5.1) allows the user to select between multiple kinds of UML elements and from there to drag and drop new ones into the diagram. XML is also used to represent these palettes however those XML files are expected to be located in the same directory as the UMLet program in a folder called ‘palettes’.

UMLet also has a text editor panel which shows the contents of a selected element and allows the user to modify it.

Finally, UMLet provides the creation of custom elements through an activate-able panel. The user can override the `paint()` method of these elements in order to give them a customized appearance.

UMLet has some other features that are perhaps unnecessary and so these were deactivated and removed from the display panel:

File Menu
Figure 5.2: An overview of the custom element interface, do not try to read any of the text. This allows users to create elements with custom printing behaviour. To the bottom right is a demonstration diagram area for this new element. The bottom left is the text editor in which the user could modify the contents of the element. The bottom middle is the custom display code, here the element’s developer can specify custom printing behaviour based on the text the user has entered.

This is a menu that allows the user to open diagrams, create new diagrams, save diagrams, create custom elements, and get help with using UMLet.

However since the goal of this tool is to design Agent Communications through UML diagrams and not UML diagrams themselves this component of the User Interface (UI) was removed.

Mail Panel

This enabled users to share the UML diagrams they have created with other users via email.

As with the file menu, this feature was not (at the moment anyways) of any relevance to this project and so it was deactivated.
Figure 5.3: An overview of the mail interface. This allows diagrams to be sent to other users through email.

We specifically chose three elements for our project out of the many provided by UMLet: Class diagrams, Relations, and Custom elements.

We chose Class elements (Figure 5.4) since in Software Engineering they represent the various elements associated with a given Defined Class or Class Object. When one draws such a class one also declares that the class will exist in the system and that instances of it can be created [Lethbridge and Laganiere, 2005, Sec. 5.2]. Typically the Class’s variables, methods and the accessibilities thereof (public vs. privately accessible) are also shown in some way though the actual implementation of UML varies widely [Lethbridge and Laganiere, 2005, Sec. 5.2].

Relations (Figure 5.5) in UMLet are numerous but for simplicity of programming only one actually needed to be referenced since they are all based on the same element.

Lastly, we use custom elements (Figure 5.6) to help in simplifying the amount of visualized data and represent the containment relationships of the hierarchical
Figure 5.4: A class element used to refer to objects, their parameters, and methods. Note the properties pane (right) is the text editor through which a user may modify the Class node’s contents.

Figure 5.5: Various relation elements used to indicate relationships between UML objects. Notice that relations can have differing colours specified as well as text. The property pane refers to the bottom most relation.

structures. We accomplished this by allowing the parent elements to be minimized and maximized. Aside from that, we model such objects to be like regular class diagrams.

Note that UMLet makes extensive use of the singleton design pattern which poses issues using it in a multi agent design platform. In some cases, multiple agents may run on the same Process and so there can only be one instance of the UMLet project shared amongst all the agents. However, since the project provides an exposed API we were able to modify all singleton based classes so that this issue is no longer a problem.

Figure 5.7 shows a stand alone implementation of this project and its use will be discussed throughout the remainder of this Chapter.
Figure 5.6: A custom element (left) and its XML code (right) built through UMLet’s custom element interface. These are useful for adding custom display behaviour to UML objects.

Figure 5.7: An overview of this project at startup. Notice the removed and disabled menu options, the three diagrams defined at startup (visible as tabs, currently the right most is selected), and the palette panel contents reflecting the default speech act file.
Figure 5.8: A new conversation type displayed in the speech act diagram named ‘new-conversation’ with default states ‘init’ and ‘terminated’ and a default required variable ‘conversation-name’. This variable labels the concrete conversation object that use this conversation type.

5.3 Meeting the Design Objectives

In this section, I will give brief descriptions on how the implementation meets all design objectives from Chapter 4. The complete list of implementations can be found in Appendix D. Where appropriate, images will be provided.

Since UMLet allows for the use of multiple diagrams, implementation **DiagsImp - 1 Different Diagrams** simply describes: the naming convention used for the three diagrams, how/where their respective files are created, and how they are destroyed.

Implementation **ElemImps - 2 Element Implementations** meets all design points concerning how the various UML elements are actually rendered in UMLet. This includes how the relevant information is displayed along with the colours used. These will all be described next.

5.3.1 Element Implementations

Figure 5.8 shows the implementation of Conversation Types. These are only used in the speech act diagram and have been coloured light blue. (see **ElemImps - 2.1 Conversation Types Implementation** in Appendix D)

Figures 5.9 and 5.10 show the implementations of Local and Global Policies respectively. Note that local policies are coloured yellow while global policies are pale
Figure 5.9: A policy that occurs locally to a conversation’s context added by the user containing default ‘Nil’ values for all fields.

Figure 5.10: A policy that is globally visible performing some action when the antecedent matches the event.

green to help distinguish their accessibility to the Agent. (see [ElemImps - 2.2 Policies Implementation](#) in Appendix D)

Figure 5.11 shows the implementation of Instance Conversations. These are only present in the instance level diagram and notice that these nodes (as well as all other nodes representing cloned instances of concrete objects) share the default colours for conversations and policies used in the other diagrams. However, the text is left black rather than grey (as with conversation templates and delegate conversations) despite the fact that they cannot be edited. This is to assist the user when debugging as black text is easier to read than grey text. (see [ElemImps - 2.3 Instance Conversations Implementation](#) and [ElemImps - 2.4 Instance Diagram Nodes Implementation](#) in Appendix D)

Figure 5.12 shows the implementation of Concrete Conversation Templates and
Sub/Delegate Conversations both using the same colour as Conversation Types and Instance Conversations. Notice that the text is greyed out to indicate that the node is not editable (parameters cannot be removed or added and the name cannot be changed) however the values passed to the parameters can be changed (provided the node is parented by a top level conversation type or is a Concrete Conversation Template). (see [ElemImps - 2.5 Templates And Sub Conversations Implementation](#) and [ElemImps - 2.6 Node Edit-ability Impelmentation](#) in Appendix D)

Figure 5.13 shows the implementation of Agent Nodes. They are coloured white and simply contain the name of the Agent. Recall that these nodes are only present when the project is run within the context of an Agent and that they are only used in the action level diagram (to show what concrete conversations were loaded from what files) and in the instance level diagram (to show what conversation instances the Agent is or has engaged in). (see [ElemImps - 2.7 Agent Nodes Implementation](#) in Appendix D)

Figure 5.14 shows the implementation of File Nodes. These are coloured black and are labeled by the file name that is currently associated with them. Changing
Figure 5.12: Left: An offer-client conversation called by ‘new-type’ in the speech act diagram. Right: An ask-client conversation implemented as a known conversation by the file node ‘package.class.init.lisp’ (see Figure 5.14 for the naming convention) in the action diagram. The called conversations use grey text to show that only the values supplied to their variables may be changed.

The label changes the default file name given. These are only used in the action level diagram to help manage collections of Concrete Conversation Templates. The files that contain Conversation Types are stored behind the scenes in an array of parsed Lisp code. Their filename (as well as the filenames of other imported types files) can be seen in the mouse over text of the palette area they populate. (see ElemImps - 2.8 File Membership Implementation in Appendix D)

Figure 5.15 shows the implementation of Containment Relations. These are unidirectional arrows that are black with white arrow heads. These indicate containment of child nodes by a parent node by pointing from the parent to the child. These are used in all diagrams to indicate either file or code containment, however, the instance level diagram also adds labels to these nodes during animation which will be demonstrated later on. Child nodes always appear below the containing parent and their declaration order is shown left to right for first to last. (see ElemImps - 2.9 Parent Node Membership Implementation, ElemImps - 2.10 Position)
Figure 5.13: Above are two images associated with an Agent within who’s context the project is running. The top image is the action diagram showing all the files that the Agent has loaded concrete conversations from. The bottom image is the instance diagram showing all the conversations the Agent has engaged in.

of Child Elements Implementation and ElemImps - 2.11 Order of Child Elements Implementation in Appendix D

Figure 5.16 shows the implementation of Instantiation Relations. These have the same meaning as Containment Relations but use a dashed arrow to add additional informative help to the user concerning the global accessibility of the policies to which they point. (see ElemImps - 2.9 Parent Node Membership Implementation)

Figure 5.14: A file node that aims to contain known conversations, stand alone global policies, comment nodes, or other Lisp code blocks. Note that the naming convention is based on the Agent’s class name and package name in order for CASA to apply it to the correct kind of Agent.
Figure 5.15: Nodes that are part of (or parented by) another node indicate this using containment relations like the ones shown here. Note that transition relations have been removed from view for clarity but are shown/discussed in Figure 5.18 in Appendix D.

Figure 5.17 shows the implementation of Conditionals. These are coloured pink and are labeled by the code fragment the condition depends on. Notice that there can only be two containment relations for these nodes and that the relations are labeled ‘true’ and ‘false’ and their arrow heads are coloured green and red respectively. This is to help the user know which case applies to which node. (see Conditionals Implementation in Appendix D)

Figure 5.18 shows the implementation of Transition Relations. These are blue unidirectional arrows with orange arrow heads. Transition Relations are used in all three diagrams and serve to indicate what policies can be executed after others. The arrow begins at a ‘set state’ command and points to the preconditions of any policies...
Figure 5.16: Similar in meaning to the contains relation however in case a global policy is hidden below a conditional (as shown here) these relations make its existence clear.

Figure 5.17: A conditional node that executes one of two possible Lisp commands based on the result of ‘some-condition’. Notice that the relations are both labeled and coloured to help distinguish between the two cases.

Figure 5.19 shows the implementation of Equivalence Relations. These are green bidirectional arrows with yellow backgrounds. They are bidirectional in order to indicate that the variables or states held between two conversations are in fact the ‘same’ or ‘equivalent’. These, in fact, are only set between a parent and child conversation using binding commands (discussed earlier in Chapter 2 §2.4.2). However, if a parent has more than one child conversation and it has a variable/state that is bound to a variable/state in both children then implied equivalence relations are inserted between them as shown in the Figure. (see ElemImps - 2.14 Equivalent States)
Figure 5.18: A transition relation between two policies, do not worry about the cut off code. The policy on the left sets the state in its consequent to one that the policy on the right requires in its precondition.

Figure 5.19: States or variables that are shared between a conversation and its child conversation as well as between sibling conversations use equivalence relations like the ones shown here.

and Variables Implementation in Appendix D

Figure 5.20 shows the implementation of Stray Comments. These are coloured green and are only used in the speech act and action level diagrams. Recall that these comments did not appear to be associated with any code fragments nor did they appear to be part of a node’s documentation as per the heuristics described in Chapter 4 §4.2 and defined in Appendix C under design point ParseNCapture - 10.4.4 Stray Comment Detection (see ElemImps - 2.15 Stray Comments Implementation and ParseNCapImpl - 10.4.4 Stray Comment Detection in Appendix D)
Figure 5.20: A comment node added by the user containing a default single line comment ‘;comment’.

Figure 5.21: A general Lisp code node added by the user containing default empty Lisp code ‘()’

Figure 5.21 shows the implementation of Stray Blocks of Lisp Code. These are coloured blue and are only used in the speech act and action level diagrams since Instance Conversation objects would not contain such objects as part of their definition (these are already evaluated when the clone-able Concrete Conversation is initially created). Recall that these pieces of code are not part of any known node’s definition as per the heuristics described in Chapter 4 §4.2 and defined in Appendix C under design point ParseNCapture - 10.2.6 Stray Lisp Code Detection (see ElemImps - 2.16 Stray Lisp Code Implementation and ParseNCapImpl - 10.2.5 Stray Lisp Code Detection in Appendix D).

The Lisp code defining Policies and Conversation Types contains a string field representing the item’s Documentation listing its intended function/use. Figure 5.22 shows the implementation of this as mouse over text. The text only displays when the user’s cursor is over the UML class object’s label. Note that the text may also contain any Lisp comments that appeared to document the node. (see ElemImps -
Figure 5.22: Any conversation types or policies that make use of the documentation parameter display it in mouse over text as shown here. The yellow popup also displays any comments that appear to be associated with the node. Both of these can be edited through the popup menu options Edit Documentation and Edit Comments.

2.17 Documentation Information Implementation, ElemImps - 2.18 Documenting Comments Implementation, and ParseNCapImpl - 10.4 Comment Capturing Heuristic in Appendix D

Figure 5.23 shows the implementation of Lisp Comments associated with a node's code fragments as mouse over text. The text is displayed when the user's cursor is over the section of the UML class object that contains the code fragment to which it refers. (see ElemImps - 2.19 Smart Comments Implementation and ParseNCapImpl - 10.4 Comment Capturing Heuristic in Appendix D)
Figure 5.23: Comments associated with or occurring within the Lisp code of a variable or section are displayed in mouse over text as shown here. The top image shows comments associated with the conversation’s ‘request-performative’ parameter. The middle image shows a comment that occurred within the antecedent code of a policy. The bottom image shows a default display message for a policy’s postcondition.
5.3.2 User Interactions

This section lists the implementations for all of the high level user interactions.

Design [ElemDefs - 2.18 Minimized Complexity] is the part of the design that allows the user to remove any top level nodes that are no longer of concern. In the speech act diagram such nodes are removed from the list of open and editable nodes and unsaved changes are lost. In the instance level diagram they are removed from memory and can no longer be animated. If the conversation has not terminated, however, then the next policy the conversation responds to will cause the conversation to be re-added to the diagram and its animation will only consist of that policy and any future ones executed. (see [ElemImps - 2.20 Minimized Complexity] in Appendix D)

TempMenuImps - 3 Speech Act Diagram Menus Implementation of Appendix D describes the available menu options in the speech act diagram. Figure 5.24 shows an example of such a popup. These simply allow the user to create top level objects, import Lisp files (in order to reference their Conversation Types), save the collection of top level nodes to a file, switch the active Conversation Types file or create a new file, parse the changes to node contents, clear the diagram of all objects, as well as select visibility settings for the types of UML elements. Other options exist that allow the user to delete top level objects from the active Types file as well as to add child nodes and edit documentation/comments (see Figure 5.22 for an example), however, the options available depend on the type of node. Lastly, the ‘clear error nodes’ option has been added as a temporary fix to UMLet. When a user double clicks a UML element a copy is placed in the diagram so this option simply scans for such occurrences and removes them. This has been diverted to future work. (see TempMenuImps - 3 Speech Act Diagram Menus Implementation and ParseChangesImpl - 7 Parsing Content Changes Implementation and
Figure 5.24: A popup dialogue for the speech act diagram’s background supporting various operations.

**NodeVisImps - 8 Node Visibilities** in Appendix D

**ActMenuImps - 4 Action Diagram Menus Implementation** of Appendix D describes the available menu options in the action level diagram. Figure 5.25 shows an example of such a popup. These simply allow the user to create files for containing Concrete Conversation Templates, open existing ones, import Lisp files (in order to reference their Conversation Types), parse the changes to node contents, show all open nodes, switch view so that all conversation templates appear as lisp code blocks (and back again), as well as select visibility settings for the types of UML elements. Figure 5.26 shows the options available for a Concrete Conversation Template in which the user may elect to delete the node from the containing file node, change its position in the file, or change the file it is contained by. These options are available to all the top level nodes that file nodes contain while file nodes themselves can only be closed or saved under a user defined filename. (see **ActMenuImps - 4 Action Diagram Menus Implementation** [EditNodeImps - 6 Editing Node](#))
Figure 5.25: A popup dialogue for the action diagram’s background supporting various operations.

Contents Implementation | ParseChangesImpl - 7 Parsing Content Changes Implementation | NodeVisImps - 8 Node Visibilities in Appendix D) of Appendix D describes the available menu options in the instance level diagram. Figure 5.27 shows an example of such a popup. These only allow the user to speed up and slow down animation as well as select visibility settings for the types of UML elements. Figure 5.28 shows the options available for an Instantiated Conversation in which the user may elect to play or reset the animation (a step in such an animation is given by Figure 5.29). The option to set as focus simply means to hide all other conversations under the Agent node. (see InstMenuImps - 5 Instance Diagram Menus Implementation and AnimImps - 9 Animations in Appendix D)

5.3.3 Input and Output Implementation

ParseNCapImpl - 10 Parser And Comment Capturing Heuristic in Appendix D explains how all pieces of Lisp code are interpreted and identified
as one of the forms (or none of them) that are of concern to CASA (regarding the
types of code shown in Chapter 2 §2.4). It also notes how comments become as-
sociated with code and when they do not (see Figure 5.30). All code is assembled
into a parse tree that supports various internal operations such as updates to specific
sections of the tree.

Implementation PrettyPrintImps - 11 Pretty Printing Implementation
describes the algorithms used to print the parse trees to file in a human readable
fashion. These simply must obey implicit design points (space separated symbols,
newlines after single line comments), explicit ones (maximum line length, comment
placement), as well as those that actually concern how the code shall aim to be printed
(through code patterns enforcing newlines in specific locations and accumulation of
indentations). Figure 5.31 shows an example of a Conversation Type’s defun that
has been printed to a file.
Figure 5.27: A popup dialogue for the instance diagram’s background supporting various operations.

5.4 Classes and their Operations

Since all elements of concern for our diagrams are elements defined by UMLet it is useful to wrap all Class elements, Custom elements, and relations within a single class, namely the UmlElement class. This class provides default functionality for drawing our diagrams such as setting the positions and sizes, issuing repaint commands, and removing/adding to the diagram. This class is extended into two kinds of elements, Nodes and Relations.

UmlNode is the class that all nodes extend including policies, conversations, conditionals, comments, code blocks, file nodes and agent nodes. It adds additional functionality such as the ability to contain child nodes, the Relations that point to and in between them, and the ability to update its Lisp code with respect to whatever changes have been made in the text box. Objects that extend this class must define how their text is printed, interpreted, what sort of diagram element they use (Custom or Class) and the colour they should use. These nodes also carry their defining Lisp representations (or a default one if not resulting from a code parse) which can be updated and integrated into their parent node (if a parent exists). In case errors are
found during an update the user is informed and the update is canceled.

UmlRelation is the class that all relations extend including contains, instantiations, transitions, and equivalences. The functionality it adds is the ability to reference two UmlNodes that it connects, where on the nodes it connects to, as well as the colours and forms it should take depending on its kind. Aside from the equivalence relations which represent variable and state bindings (see Chapter 3 Figure 2.9) relations are not actually reflected by any Lisp code. Instead, relations are created out of the code defined by their linking UmlNodes and/or other class data. Such relations are automatically generated and removed when changes are made to the UmlNodes or through popup menus on the UmlNodes (as with linking equivalence relations).

5.5 Diagrams and their operations

The following subsections describe the three diagrams used by this project for user interaction. This includes what each diagram is meant to be used for, the meanings
Figure 5.29: The second keyframe of the ask-server conversation receiving a request that the arithmetic ‘1+1’ be performed. The first keyframe placed the event type, message content, and the resulting new conversation state on the relation to policy ‘ask-server-000’ which was executed in response to the event. This keyframe acts instead on policy ‘ask-server-003’. As these keyframes are shown the parenting conversation marks its current state in bold.

for the elements that are present in the diagrams, and how the palette and text panels interact with the elements contained therein.

5.5.1 The Speech Act Editor

The diagram that is currently selected in Figure 5.7 is what we refer to as the ‘speech act’ diagram since the conversations it displays are primarily concerned with the occurrence of certain kinds of performative utterances (or speech acts) occurring in a particular sequence.

When this diagram is selected and made visible through the tabbed pane it is meant to be an editor and creator for the elements contained within the palette. Note that the palette represents the elements contained within the conversation types file,
Figure 5.30: Top: A portion of the original code being parsed (the enclosing conversation definition is omitted). Bottom: The resulting UML children of the omitted conversation. Notice that the non-indented comment ‘;comment’ has been given its own node while the indented ones occurring before the policy’s definition (as well as the trailing comment occurring immediately afterwards) are collected in the policy’s mouse over text.

A file that contains defuns whose form matches that of a conversation type. This is a file that will we be loaded by CASA at startup in order to form the concrete conversations defined in the Agent startup files by executing the templates within (see 5.5.2). The current loaded file path and name is visible through mouse over text in the palette area.

The current working file can be changed to another through the diagram’s background popup menu. Defuns from other files can be referenced by including them as imports through the diagram’s background popup menu though they do not appear in the palette panel.
Figure 5.31: The complete recursively constructed Lisp code of Figure 5.30 showing the conversation type’s parenting code as well. Notice that increasing levels of indentation has been applied and thus that the stand alone comment ‘;comment’ will be associated with the policy’s mouse over text in the future.

Aside from the conversation types (shown as red nodes in the palette area of Figure 5.7) the palette also includes any sections of comments found throughout the file (as green nodes) and any occurrences of Lisp code that do not match the form of a conversation type (as blue nodes). Since the declaration order matters in some cases (such as when a block of Lisp code makes some decisions on what value a variable should take before it is needed) the user can select the elements in the palette and drag them above or below other elements to change the order. Additionally the user can hide the comment nodes in the palette since they provide no control in the file and can be so numerous that they are a nuisance to the user. However if a node’s declaration order is changed while comments are hidden the moved node will be moved either immediately after/before the next higher/lower visible node. This means that comments aiming to document some behaviour with respect to that point in the file may be in the wrong place if they are even still accurate to begin with.
To actually edit any of the nodes in the palette the user only needs to click and drag the element from the palette onto the diagram. When this happens a lookup is performed for the particular piece of code the element represents and the code is parsed and displayed on the screen. The user can then edit the contents of the node through the text editor, perform binding operations if it can and does parent another conversation, and add/remove children where appropriate (again, assuming it is a node that can contain children). The user can then order either a re-parse of the new contents or an update to the logged representation both of which will perform error checking first.

If the user wishes to add more elements to the current working file (comments, code blocks, or conversation types) the popup menu of the diagram’s background has options for these.

### 5.5.2 The Agent Startup File/Action Level Editor

The middle tab in the project’s overview (Figure 5.7) is what we refer to as the ‘action’ diagram since the conversation templates and global policies it displays are primarily concerned with the execution of specific actions which are carried out by the sequence of speech acts defined by the conversation types the templates use.

In this diagram there are two possibilities for its contents at startup. If this program is run stand alone (non Agent context) then the diagram is blank. If the program is, however, run within the context of an Agent then the action Agent node is present along with all the files it has included (represented by File nodes) containing all of the known actions each file defines (see the top portion of Figure 5.13).

In either case, this diagram aims to allow the user to create and develop agent initialization files (startup files) with the naming convention “agentPackage.agentClass.-init.lisp”. The user is to add or remove comments, code blocks, conversation templates, and global policies to File nodes within this diagram and save the File nodes
so that they may be used during an Agent’s startup.

The palette can only be used to drag conversation types into the diagram while everything else is disallowed. Adding a conversation type results in a popup dialogue in which the user is to fill out the required and desired parameters of the `defun` such that a conversation template may be created. This route however, requires that the user then inform the template which file it is to be a member of. This behaviour may be desirable as the user might just wish to look at the form the template takes or its effect on the structure of the conversation type used before actually adding it to a file (though removal is a trivial operation as well). The alternate route is for the user to interact with the File node’s popup menu and select to add a conversation template under a given type that way.

This diagram also supports the importing of alternate conversation type files through it’s popup menu whose types can then be used for the construction of templates. However for that to work, an import statement for the types file in question will need to be supplied as a Lisp code node. This code node must appear in the action file node before any of the imported types are used otherwise failure to execute the templates will occur.

Modifications to the various nodes in the diagram can be accomplished through the text editor and a request to re-parse the changes can be made through the diagram’s popup menu. Any elements that have errors and thus cannot be safely parsed meaning no correct Lisp representation can be created for them will result in a user notification. Should the user choose to save the file anyway then such error nodes will not be saved to the file.

5.5.3 The Animation Viewer

The left most tab in the project’s overview (Figure 5.7) is what we refer to as the ‘instance’ diagram since the conversations and policies it displays are the result of
actual instances of the class objects that have been defined due to some message exchange amongst Agents.

This diagram does not interact with the palette panel as it only adds nodes of conversations that the project’s Agent Observer class has witnessed when running within the context of an Agent (see the bottom portion of Figure 5.13). Thus if no Agent is defined because the project is running stand alone the diagram will remain empty.

To add a conversation the user needs to make the Agent engage in some conversation that he/she wishes to observe. Once the conversation has been created (typically within a few seconds) it will be available as a child of the Agent node in the diagram. The user can then use the conversation node’s popup menu to play the animation.

Playing an animation causes the conversation to move through a list of keyframes. Keyframes are pairings of the policy that was executed and the event that caused the policy to be executed. The keyframes have a delay of 2 seconds between them which the user can increase or decrease through the backgrounds popup menu.

Animating a single keyframe has three effects on the conversation and it’s children:

1. **Message Executions**: Policies that are executed due to their antecedent matching the message will have the message’s contents posted on the contains relation that links them to the parent conversation. The state that the policy moves the conversation into (if their is such a transition) is also posted.

2. **Update Current State**: Conversations will be notified of the state they are in, search through their state definitions for the matching state and wrap it in bold text. If the conversation is a child conversation then the state binding is used to translate the parent conversation’s current state which gets searched for and wrapped in bold text as well.

3. **Execution Sequence**: Finally, the sequence of policies executed is written to the
conversation node’s mouse over text. This numbers the execution order and writes the name of the policy and the message/event it captured.

Note that in this diagram no interaction with the text panel has any affect on the elements therein. That being said, it can be useful to use it in order to more easily browse the contents of the animated containment relations.

5.6 Other Utilities and Methodology

There are several utilities that do a large amount of work in the background that were not directly required by Chapter 3 nor specifically mentioned as part of the design in Chapter 4 but were necessary and are used often.

5.6.1 The Layout Manager

As the name suggests this class merely lays out the elements in the diagram in a recursive manner.

Recall that all child nodes need to appear below their parent node and so they are told to appear at a distance of half the parent’s height below the parent. The parent is told to centre itself amongst its children. Neighbouring nodes, or neighbouring hierarchies of nodes, are told to separate by 100 pixels.

5.6.2 Popup Menu Maker

The popup menu maker was important as it allows nodes to create their popups upon instantiation while providing them with an interface to the main part of the program. Additionally the popup maker filters the actual content of the menu options based on the kind of the node being created.
5.6.3 Listeners

Of the many listeners the code uses, two are of particular interest.

**Diagram Container Listener** This allowed us to detect whenever a component was added to the diagram from the palette area. If it shouldn’t be there (as when an element is added to the instance level diagram) remove it. If the move was valid however (like when a conversation type is added to the speech act or action level diagrams) the component (which would then be a red node containing the type’s name) is removed and swapped for the appropriate representation.

This is how the drag and drop functionality in the three diagrams was achieved.

**Tool Tip Text Listener** This listener allowed us to place mouse over text whenever the user’s cursor was above a node. However the text place was the appropriate text. Meaning it identified the specific position of the cursor on the component and, with respect to the section of the node at that position, it displayed the comments associated with that section as opposed to the comments that documented the entire node.

5.6.4 Abstract Lisp File Class

This class simply provides an interface for dealing with the parsed forms of Lisp files such as the current working conversation types file or any open Agent initialization files.

This made accessing elements by name, kind, or index much easier and allowed for the swapping behaviour we desired when editing such a file. Though saving could have been done rather easily using any other method (such as an ArrayList field) it was still helpful to have a common object with which to localize the notion of a *parsed file* and its contents.
This section outlined the various implementation details for the UMLet wrapper project that interfaces with CASA agents and their communications dubbed ‘CasaUmlet’.

5.7 Summary

In this Chapter the open source UML design tool UMLet [Auer et al., 2015] was selected for this project to wrap. Some notes were given in regards to how the diagram area, palette area, and text editing area were overridden to have the desired behaviour. Additionally, all of the design choices from the previous Chapter were given an implementation in this Chapter involving the class objects associated with the conversational elements and those class objects dedicated to parsing Lisp text.

Lastly, some supporting utilities were noted that made keeping track of parsed information easier (the AbstractLispFile class), keeping the diagram layout clear (the LayoutManager class), the popup menu construction class (the PopupMaker class), and the mouse over listener which allowed us to display the comments and documentation associated with code fragments and the UML nodes (the ToolTipListener class). There are, of course, other supporting classes but their purpose is geared toward simplifying the use of Java’s Swing and AWT frameworks while having little to do with the implementation that supports the design and requirements. As such, these have been omitted.

The goal was to show how all design details from Chapter 4 have been met by the implementation and now a discussion of how this project was tested will be given in Chapter 6.
Chapter 6

Testing

Testing is one of the most (if not the most) important aspects of software development as it is also considered to be both useful and highly beneficial when combatting human error \cite{Lethbridge and Laganiere, 2005}. Through testing we can potentially expose bugs that may not have been thought possible and verify that the software behaves according to our specifications.

Testing is usually divided into two paradigms: white box and black box testing. \cite[Sec. 10.1]{Lethbridge and Laganiere, 2005} White box testing is where the tester aims to provide tests that cover all possible paths code may take (excluding infinite loops). Black box testing is where the tester provides inputs to the application and observes the outputs for correctness.

In addition to these are strategies for testing larger systems such as integration testing specifically incremental testing where after each subsystem of the application has been tested individually the testing of their combinations are performed, that is, they are integrated with each other then tested \cite[Sec. 10.9]{Lethbridge and Laganiere, 2005}. Doing this, one can employ either a Top-down, Bottom-up, or hybrid strategy. Top-down testing involves first testing the user interface with stubs, pieces of code that have the same API as the integrating layer but do nothing. When complete, the next layer is integrated and testing of that layer is performed. This process continues until all layers have been integrated and tested. Bottom-up testing is the opposite where first the lowest layers are tested by employing drivers which are simple programs that make calls to the lower layers similar in concept to stubs. Sandwich testing is the hybrid which just employs both strategies until the system
is completely integrated and only the middle layer remains on which to perform the final set of tests.

Finally, we should mention the two major forms of testing, **Automation** and **Test Cases**. [Lethbridge and Laganiere, 2005, Chap 10] Automation is perhaps the most widely known as it can quickly perform many complex checks. Automated tests are essentially suites of methods that perform operations programmatically on the application verifying the results. Test Cases on the other hand depend on **Use Case** scenarios. A Use Case describes one way that a user can interact with the system and its possible outcomes. Test Cases are simply the tests that describe the sequence of actions that should lead to a given outcome. This includes actions the user performs with the application and the actions the underlying software does in response. These are verified through live demonstration of the application.

CasaUmlet employs testing through a conglomeration of these methodologies as will be seen in the next section however what should also be noted is **what** is being tested. The important aspects of the application are broken down and tested as follows:

1. Reading of Lisp Files: Test that the Lisp files are read in correctly by the ‘Parser’ causing the correct data structures to be created.

2. Writing of Lisp Files: Test that the data structures are correctly written to the file through their `toString()` in a human readable fashion (i.e. correct commands, correct indentation, carriage returns, etc.)

3. Supporting Logic: Test that the various methods throughout the application which provide non state changing functionality behave correctly.

4. Diagram Interface: Test that the various UmlNode objects initialize within their respective diagrams properly.
5. User Interaction: Here we test that the various user interactions provide the correct changes to the state of objects in the application, whether it be an update to the underlying Lisp representations, an update to the graphical representations, or that an error/warning message is displayed in some way.

Testing these should guarantee that a ‘useable’ application is provided to the user though not necessarily one that is completely bug-free.

6.1 Performing the Tests

This was implemented through automated testing using Junit, a free and easy to use java testing service that allows the tester to implement drivers for the code under test. Testing was performed from the bottom up using automation however when the full system was tested a mixture of automated and user test cases was used to verify the results (discussed in §6.1.5).

6.1.1 Reading of Lisp Files

For all of the known kinds of Lisp code, we created and ran dummy Lisp code through the parsing utilities to ensure that the correct data structure trees were created. This means that the Objects created were of the expected kind, had the correct commands, and that their child elements were also correct in the same way. In addition their are also tests to verify that modifications to a command data structure could be performed correctly as well as tests to ensure that malformed (as well as non existent) Lisp code fails to be parsed.

Note that while all these tests passed there is no guarantee that the parsing subsystem is entirely bug free due to the infinite possibilities in which Lisp code may be written. In testing this is referred to as combinatorial explosion [Lethbridge and Laganiere, 2005, Sec. 10.2] and is avoided by testing representative values (values
that should extend to all cases) and boundary conditions (such as null or incorrect inputs).

6.1.2 Writing of Lisp Files

For all of the known kinds of Lisp code tested in the previous section their `toString()` method was also tested. The method was invoked to ensure that the Lisp code that would be written to a file matched what was defined as ‘human readable’.

These tests were implemented before the actual implementation of the `toString()` methods in a form of test-first development. Lethbridge and Laganiere, 2005, Sec. 10.8] Test first development is aimed at ensuring that the desired behaviour is being created by verifying that the number of tests passed increases. However, as with automated testing, there is no guarantee due to the infinite possibilities.

6.1.3 Supporting Logic:

Since ‘CasaUmlet’ takes text and produces Lisp data structures there is no doubt a lot of code used throughout the various classes and methods that do not strictly require the object to be instantiated. This includes methods that take an array of objects and some descriptor and returns the first object that matches the description. Such methods have all been tested through automation with Junit along with their relevant boundary conditions.

6.1.4 Diagram Interface:

Here automated tests created objects in the UML diagram and verified that they had the correct properties (colours, default content, default Lisp, number of children, etc.) and that they appeared on the diagram through their `isVisible()` command.
6.1.5 User Interaction:

I tested user interaction in two steps in order to avoid testing the User Interface directly. The main reason for this is that the goal of testing this system is not to test and verify that Java’s event handling system is working correctly since Java’s maintainer, Oracle, is already testing this. Instead the goal is simply to test that the added functionality performs the intended operations correctly.

Thus testing for user interaction was done in the following manner:

1. Automation Testing:

   I performed automation testing of the user interactions by calling the same methods that would result from using the Java User Interface (popup menus, input dialogues, etc.). However wherever input was required, default behaviour was assumed (such as pressing ‘Ok’ in cases where ‘Cancel’ would do nothing or by assuming that some kind of default input string was given).

2. Test Cases:

   The Test Cases in Appendix E represent the entire system’s integration test. Since the Java User Interface has been tested (by Oracle) and the logic that the system’s UI uses has been tested (as mentioned above in ‘Automation Testing’) these then tested the remaining integration. Thus the test cases describe sequence of actions, such as selecting a menu item through a mouse-click and the results (the appropriate methods being called) but do not test the registration of the mouse click nor that the correct mouse event is generated correctly in the Java system (though this can be implicitly reasoned through the integration testing).
6.2 Bugs

Testing was indeed invaluable to the software’s implementation. Early on, when testing the Lisp parsing subsystem, bugs revealed that certain kinds of Lisp code were not consuming their respective child Lisp elements properly to form the correct data structures (associated comments in particular went missing).

The output to file tests were part of a test-first development ideology and as such didn’t find pre-existing bugs but instead identified them whenever created.

The automation testing on the UML diagrams verified that the correct changes were made to the supporting data structures of the UML nodes in the diagram. That is, when new child nodes were added they were confirmed to be present in the parent node’s child list (or not present in the case of a removal) and all their parameters were set appropriately.

Integration of the UML diagram system with the Lisp data structure subsystem verified that changes to a node’s contents either performed the correct changes to the diagram, the node’s Lisp tree representation, or showed the correct error message in the node’s contents. This also tested the File outputs of these nodes.

Additionally, the automation testing of the UML tree helped when methods and concepts needed to be reconstructed. For instance, at one point it became apparent that the various UmlNodes were implementing the same methods in different ways for different reasons. In order to bring their implementations in line with each other, methods were added to meet the different needs. After this was performed it became apparent (through reruns of the tests) that some of the original algorithms needed to be changed or augmented before the final phase of testing could take place.

Once all integrations had been tested, the ad-hoc tests for UI needed to be formalized in order to verify that all possible operations were in fact being performed correctly. In that interest the integration of the UI with the UML and Lisp tree
systems was performed through the various test cases in Appendix E. Though this
wasn’t expected to find any bugs their was a minor one that was caught: when a new
conditional node was added to a conversation type it didn’t have any default children
associated. However its default Lisp representation defined two empty lists. This was
based on an assumption that no children should be created since their was no actual
parsing going on. This was easily fixed.

6.3 Summary

This Chapter discussed some of the forms of software testing available and in what way
this project utilized them in its testing. Generally, there exists white box testing and
black box testing where the former tests functions with the knowledge of their inner-
workings to expose potentially buggy edge cases while the latter tests the software
in the same way that a user might use it. Both use representative test cases when
the testing space is very large with the hope that these cases imply that all (or at
least most) uses of the software behave correctly. Also there is a methodology known
as test driven development which simply requires that we write the tests first then
develop the software to meet the test cases. Lastly, there is integration testing where
major components of the software are tested separately followed by their integrations
before finishing with the testing of the entire project.

White box testing tested any supporting logic in the software that does not depend
on application state. This simply included any methods that didn’t require an object
to be present.

White box testing verified that objects were created correctly when reading Lisp
files and that they could be manipulated with the correct results. The user interface
was tested with this methodology in both an automated and manual form so as to
encapsulate the layer of functionality introduced by Java’s Swing and AWT APIs.
Test driven development ensured that the Lisp code produced by runtime objects matched what was desired starting with simple cases followed by the more complex ones.

Integration testing occurred throughout development in a bottom up fashion as certain aspects of the project depended on others. First, it was desirable that correct file reading and writing was achieved before ensuring that the correct UML objects were created from the parsed code objects and that they produced the correct output. Second, the interactions with the UML objects was tested now that we knew that they reflected and produced the correct Lisp code. This included all the manipulations that could be performed on the UML objects programmatically. Lastly, the user interactions was tested now that the UML objects had the desired affects when manipulated. This simply checked to see that the activities a user performed had the desired affects through manual test cases.

Given the amount of bugs found and avoided the conclusion can be drawn that the testing paradigms and methodologies employed were adequate in assuring a certain level of usability of the software in question.

Chapter 7 will explore an actual demonstration of the software’s use followed by a verification of the results, again, through the use of the software and an argument will be made for the software’s usefulness.
Chapter 7

Analysis

To argue the utility of this work I will go through a single extended use case in detail. To that end, I will give an attempt to use the project to develop the request-when conversation followed by the results. This chapter will then make mention of some related work in regards to debugging Agent Communications in CASA before concluding with the summary.

Request-when means to request that some action takes place when an event has occurred. Note the difference between this and a request-whenever conversation or a subscription which requests that some action take place every time an event occurs.

The key difference between a request-whenever and a request-when is therefore seen in how the two conversations terminate. The request-whenever is a continuous subscription that only terminates when the subscription terminates or communication fails between the two agents. The request-when however is a single use subscription in that the conversation terminates after the first execution.

7.1 Creating A Single Use Subscription Conversation

As shown earlier in Chapter 5 the ‘CasaUmlet’ tool can be run stand alone or within the context of an Agent. Since the process of creating the new conversations does not require an Agent context (and a CASA Agent cannot as of yet add new conversations at run time anyway) we will begin this process in stand alone mode as shown in Figure 5.7 in Chapter 5.
7.1.1 Making the Client & Server-side Conversation Types

Figures 7.1 to 7.3 show how we begin the process of making the client side of the request-when conversation. Since a strong similarity has already been drawn between the subscription conversation and the request-when conversation this example proceeds by modifying the former and saving it as the latter.

In order to reduce the complexity of what the user sees in Figure 7.3 the user may want to hide the comment nodes through the popup shown in Figure 7.4 or (since we are modifying this conversation) outright delete them. Both methods achieve the results shown in Figure 7.5 such that the amount of visible information is reduced.

Now the user may begin exploring the conversation’s components and make the
necessary changes. Figure 7.6 shows that the user has selected the global policy that causes the conversation’s instantiation at run time. The text editor shows the contents of that policy which has a naming convention based on subscription and the performative of the antecedent requires that a subscription be sent. Figure 7.7 shows the necessary modifications to name this policy after the new request-when notion being developed and to make this policy instantiate the parent conversation when a request-when message is sent rather than a subscribe. The necessary changes have been highlighted for clarity.

Figures 7.8 and 7.9 show partial views of the first local policy. Similar to the modifications made previously to the global policy the naming and relevant performatives of the policy were changed. In addition, since waiting for a subscription isn’t very relevant anymore, we change the state the policy moves to from “waiting-subscribe”
Figure 7.4: The speech-act diagram’s background popup menu option allowing the user to hide all comment nodes.

to “waiting-agree” indicating that we are waiting for the server to agree.

Naming, performative, and state modifications are made throughout all the policies however the policy that must be modified in order to achieve the necessary change in behaviour is policy number “007” shown in figures 7.10 and 7.11. This is where an inform message has been received as per the client’s subscription. The client side policy performs its local inform action and remains in the “active” state. Thus the postcondition of the new policy will instead expect that that conversation moves to the “terminated” state and the consequent will perform the state change.

Figure 7.12 shows the subscription conversation node’s contents. To create the new request-when type requires renaming the conversation as well as the parameter ‘conversation-name’ that generates its runtime name. Figure 7.13 then shows the user performing a save to types action followed by Figure 7.14 which asks if the name change means that a new type is being created. The user selects ‘No’ (meaning to create a new conversation type).

Finally the user wishes to save these new conversations so that they may be referred to during an Agent’s initialization. Figure 7.15 shows the speech act diagram’s
Figure 7.5: A simplified overview of the client side of the subscription conversation. Notice that comments have been omitted from view through the visibility menu.

popup menu that leads to the types’ file save window. There exists two methods to do this which depends on how the user wants to treat these new types. If the user wishes to include these with the other types and overwrite the types file in their CASA distribution then they can navigate to the file’s location and save over it as seen in Figure 7.16. The other option is illustrated in Figure 7.17 where the user has removed all other types except for the two new ones. These will be saved under some new file name which an Agent’s initialization file must load before the types can be used by the file’s concrete conversation templates.
Figure 7.6: The instantiating policy (left) of the client side of the subscription conversation and a partial view of its contents (right). Notice that selecting the node in the diagram has caused its contents to be opened for editing in the text editor. The highlighted code pieces are what must be changed.

Figure 7.7: The modifications to the instantiating policy of the client side of the subscription conversation such that a request-when conversation is created instead.
Figure 7.8: The contents of the first local policy of the client side of the subscription conversation.

Properties

"subscribe-client-000"

--
bg=orange
fg=black

__antecedent:__

`'(msgevent-descriptor event_messageSent :performative subscribe :act (act ,the-act) :sender
--
__precondition:__

')(equal (conversation.get-state) "init")

--
__postcondition:__

')(equal (conversation.get-state) "waiting-subscribe")

__consequent:__

`'((sc.add :Debtor server :Creditor client :Performative reply :Act (act subscribe ,the-act) (conversation.set-state "waiting-subscribe"))

Figure 7.9: The modifications to the first local policy of the client side of the subscription conversation such that conversation state and social commitments control and refer to a request-when conversation instead.

Properties

"request-when-client-000"

--
bg=orange
fg=black

__antecedent:__

`'(msgevent-descriptor event_messageSent :performative request-when :act (act ,the-act
--
__precondition:__

')(equal (conversation.get-state) "init")

--
__postcondition:__

')(equal (conversation.get-state) "waiting-agree")

__consequent:__

`'((sc.add :Debtor server :Creditor client :Performative reply :Act (act request-when) (conversation.set-state "waiting-agree"))

96
Figure 7.10: The contents of the local policy that responds to inform-events from the server. This is part of the client side of the subscription conversation.

```
Properties
"subscribe-client-007"
  __
  bg=orange
  fg=black
  __
  _antecedent:_
  "(msgevent-descriptor event_messageReceived"
  __
  _precondition:_
  '(equal (conversation.get-state) "active")
  __
  _postcondition:_
  '(equal (conversation.get-state) "active")
  __
  _consequent:_
  "([]inform-action)"
```

Figure 7.11: The modifications to the local policy that responds to inform-events from the server. This includes renaming the conversation and (most important) terminating the conversation. Notice the addition to the consequents facilitating termination.

```
Properties
"request-when-client-007"
  __
  bg=orange
  fg=black
  __
  _antecedent:_
  "'(msgevent-descriptor event_messageReceived ; performative inform-ref ;<act"
  __
  _precondition:_
  "'(equal (conversation.get-state) "active")
  __
  _postcondition:_
  "'(equal (conversation.get-state) "terminated")
  __
  _consequent:_
  "([]inform-action ,terminate-action (conversation.set-state "terminated"))"
```
Figure 7.12: The subscription conversation node’s containing text. Note that the type is named `subscription-request` and that the conversation-name parameter also contains the word ‘subscription’.

Figure 7.13: The conversation type node’s popup menu where the user is electing to save the changes.
Figure 7.14: Conversation type rename warning asking the user whether or not a new conversation type is what’s intended.

Figure 7.15: The speech-act diagram’s background popup menu option allowing the user to save the current list of conversation types to a Lisp file.
Figure 7.16: The file saver dialogue. The actual directory contents are irrelevant. The user is choosing the file name ‘defs.process.lisp’ most likely with the aim to overwrite the file CASA loads at startup.
Figure 7.17: The file saver dialogue. This time the user is choosing the file name ‘requestWhen.process.lisp’ which will only contain the two new request-when types. The file will be separate from the default types file loaded by CASA at startup. An Agent’s initialization file will therefore have to specify that the file be loaded before attempting to use the types in concrete conversations templates.
7.1.2 Making the Concrete Client & Server-side Conversations

Now that the types have been saved to a file the user must create the concrete conversation templates that will test the functionality of the new types and save them to a file. This is done in the action-level diagram and starts with creating a file node that will contain the concrete conversation templates through the popup menu shown in Figure 7.18. A popup dialogue allows the user to specify a file name to use as the file node’s label, however, the file name can of course be changed at anytime as well as in the file saver dialogue.

Figure 7.18: The action-level diagram’s background popup menu option allowing the user to create a file node reflecting the Agent initialization file that will contain the new concrete conversation templates.

Now assume that the user has chosen that the two new types will be contained within a separate file named ‘student.lisp’. Therefore the user must first declare that the file be loaded. This is done by adding a code block node (Figure 7.19) and filling it with the ‘load-file-resource’ command shown in Figure 7.20.

Figure 7.20 also shows the beginning of the next step, that is, to add a concrete conversation template that implements the client side of the request-when conversation type. A popup dialogue like the one in Figure 7.21 is shown and the drop down
Figure 7.19: The file node’s popup menu option allowing the user to add a generic Lisp code segment.

Figure 7.20: The file node’s popup menu option allowing the user to add a new concrete conversation template based on the available types.

is where the user selects the name of the type to use and presses “OK”.

Now that the desired type has been selected the user must complete the necessary parameters for the concrete conversation template. A popup dialogue like the one shown in Figure 7.22 will appear. This dialogue requires two parameters: ‘the-act’ and ‘subscribe-action’. The first parameter is simply the name of the act such as “open-the-door” or “delete-the-log-file”. The second parameter is what the Agent should do once the conversation has been instantiated. This is usually the execution of a function defined in the Agent’s declaring java file such as “AttemptToOpenDoor()” or “AttemptToDeleteLogFile()”.

103
Figure 7.21: A popup dialogue in which the user must select the type of concrete conversation template to create.

Figure 7.23 shows the completed template in which all fields marked REQUIRED have been completed. The act is unimaginatively named “request-when” and the action to be performed is a function call in the Agent’s declaring java file named “sendRequestWhen”. The KEYed parameter ‘agree-action’ has also been overridden. The Agent will print to both its log file and console “RECEIVED AGREE” when it receives the server’s ‘agree’ to the request. When the user clicks “OK” the concrete conversation template is added to the file just after the file load command as shown in Figure 7.24.

Since it may be useful to document the behaviour of the new client side conversation the user adds a comment node similar to how the code block was added in Figure 7.19. The user then places it just before the conversation, and modifies its contents to something informative for future developers. Figures 7.25 to 7.28 reflect these operations.

Next the user adds the server side of the request-when conversation. The parameters given are shown in Figure 7.29. Note that it is important that the first parameter ‘the-act’ receives the exact same name as in the client side of the conversation (namely “request-when”) such that the client and server know they are talking about the same thing.
The second parameter ‘the-event’ indicates what kind of event the server should be watching for. For ease of testing we will let the type be as generic as possible and simply require that an “event” occurs. The third parameter ‘the-watched-event’ allows the server conversation’s agree action to list the types of events that will be observed for logging purposes. Since the type is “event” it is unrealistic to provide this information so the parameter is left “NIL”.

The fourth parameter ‘request-when-action’ is that action that the server should perform when the conversation has been instantiated. Again the user opts for a method call into the server Agent’s declaring java file named “receiveRequestWhen”.

The last required parameter ‘inform-action’ is an extra action the Agent can perform when informing the client Agent. Here we have again declared that a print statement should take place though this could have been left as “NIL” as with the client’s ‘agree-action’ parameter.
Figure 7.23: The popup dialogue in which the user has completed all the REQUIRED fields to make the client side conversation template.

Again, the user adds a comment that documents the meaning of the template and adjusts its position to just before as shown in Figure 7.30.

Finally, Figure 7.31 shows the user electing to save the file with the commands in the given order (left to right) as a Lisp file through the file node’s popup menu. Figure 7.32 shows the actual file name to be used and the location for the file. The file is being placed in a java projects sub folder called ‘dataFiles’ which CASA knows to look in, however if the file is not found then CASA will search in the project’s directory. If it is still not found and is not found anywhere in the CASA distribution’s file set then the Agent will only have access to the default Agent conversations defined in the file ‘casa.TransientAgent.init.lisp’ along with whatever files that one refers to.

The naming convention of these files must obey the Agent’s java project settings. Thus if the Agent’s java class file is named ‘Tristan’ and the package it is a member of is named ‘auriniAgents’ the file name must therefore be ‘auriniA-
Figure 7.24: The client side template has been added as a child to the file node just after the file load command. Notice the greyed text indicating that the conversation cannot be modified aside from the values given to its parameters.

agents.Tristan.init.lisp’. Since the Agent extends CASA’s ‘TransientAgent’ class the file ‘casa.TransientAgent.init.lisp’ will also be loaded along with whatever files that that file references and any other ‘...init.lisp’ files complementing the entire class hierarchy.

In order for the Agent to use the two templates defined the user must make one final modification to the file ‘Tristan.java’. Recall that the two concrete conversations perform actions reflecting java method calls: ‘sendRequestWhen’ and ‘receiveRequestWhen’. We must ensure that both of these methods are actually defined otherwise the method lookup will fail and errors will be logged to the Agent log files.

Figure 7.33 shows both of these methods which take a CASA MLMMessage object as
Figure 7.25: The comment node’s popup menu option allowing the user to change its position within the file. This option is available to all child nodes of the file node.

their sole parameter. From these parameters we can obtain the sender’s information, the message content, the language the content should be interpreted with, as well as many others as defined by the general KQML message syntax. The user uses these methods to do work within the Agent when the conversation begins. However for the most part we have just used print statements rather than method calls since we are only interested in testing the new types.

As a final note, the defined CasaLispOperator ‘AGENT_REQ_WHEN’ allows the user to send the initiating ‘request-when’ message from the Agent’s command line interface as will be demonstrated. This merely allows the user to call the operator
Figure 7.26: The change position popup dialogue. Here the user selects “index 1” so that the comment will move ahead of the template but after the file load command located at “index 0”.

![Change Ordinality Dialogue](image)

Figure 7.27: The results of the change position popup dialogue in which the comment has moved to index 1.

![Diagram](image)

with the desired receiver URL as opposed to writing the complete message on the command line. Though it isn’t shown the CasaLispOperator actually takes a named parameter with which this operator can be accessed from Lisp: ‘AGENT.REQUEST-WHEN’.
Figure 7.28: The text panel reflecting the contents of the comment node. The user has changed the comment section to include documentation as to what the next command is doing.

Figure 7.29: The popup dialogue in which the user has completed all the REQUIRED fields to make the server side conversation template.
Figure 7.30: Overview of the file node with all necessary child nodes. From left to right, the types file import command, the client side request-when preceded by its doc, and the server side request-when preceded by its doc. Do not try to read any of it, this just illustrates that the Agent initialization file is ready to be saved.

Figure 7.31: The file node’s popup menu option allowing the user to save the Lisp representations of the child nodes in the order shown. Left to right is first to last respectively.
Figure 7.32: The file saver popup dialogue allowing the user to save the Lisp representations of the child nodes to a file defined by the user.

Figure 7.33: The methods that must be defined in the ‘Tristan’ Agent’s java file in order for the conversation to take place.
7.1.3 Testing the New Conversations

Now that the conversation templates’ java methods have been defined and all files are in place, the user runs two ‘Tristan’ agents within CASA with the runtime names ‘TJ0’ and ‘TJ1’ and with Agent URLs ‘/TJ0’ and ‘/TJ1’ (Figure 7.34).

Figure 7.34: Two ‘Tristan’ agents started in CASA named ‘TJ0’ (top) and ‘TJ1’ (bottom).

The user now adds an instance of the ‘CasaUmlet’ plugin to each Agent as a frame through the Agents’ ‘Tools’ menu (Figure 7.35). Figure 7.36 then shows that each Agent has a copy of the project running within their contexts. Figure 7.37 shows the ‘instance-level’ diagrams which will contain the observed concrete conversations the agents have engaged in.

Recall that to test these types and the concrete conversations whose templates use them we must first send a ‘request-when’ to one of the agents so that the conversation can start taking place. Figure 7.38 shows ‘TJ0’ acting as the client using the CasaLispOperator “agent.request-when”, specifying “/TJ1” as the server.

The conversation takes place very quickly and has completed before the user is aware. However the instance diagrams have logged the conversations and their
Figure 7.35: Opening an instance of the ‘CasaUmlet’ project as a frame of Agent ‘TJ1’. The project therefore runs within the context of that Agent.

complete progression accessible as an animation. Figure 7.39 shows that both Agent nodes in the instance diagrams have captured and logged the conversation instances.

Figures 7.40 to 7.42 show the progression of the client side conversation as seen by Agent ‘TJ0’. (Note that the unused and global policies were moved off screen to make the images clearer.) First the sent message causes the client conversation’s state to move to “waiting-agree” as it is waiting for a subscription confirmation from the server (Figure 7.40). Next the agreement is received and the client conversation moves into the “active” state (Figure 7.41). The client then receives an inform message and the conversation terminates (Figure 7.42).

Initially the mouse over of the client side conversation showed no event execution sequence (Figure 7.40). Now that the conversation has completed however, the mouse over text in Figure 7.43 shows the sequence of policies that were executed as well as message event details including the message event type, the performative of the message, and the actual content if any is present.

Figures 7.44 to 7.46 show the progression of the server side conversation as seen by Agent ‘TJ1’. First the reception of the request message causes the client conversa-
Figure 7.36: Both ‘Tristan’ agents are populated with an instance of the project.

tion’s state to move to “waiting-agree” as it is waiting for its subscription confirmation to be sent (Figure 7.44). Next the agreement is received by the client and so the server conversation moves into the “active” state (Figure 7.45). The server then detects the reception of an inform event which matches the conversation’s observed type (since a message event is an event). Thus the server informs the client, performs its local ‘inform-action’ and sets the conversation to terminated (Figure 7.46).
Figure 7.37: Both ‘Tristan’ agents showing their conversation instance diagrams.
Figure 7.38: Agent ‘TJ0’ initiating the ‘request-when’ concrete conversation.
Figure 7.39: Both ‘Tristan’ agents showing the captured client (left) and server (right) conversation instances in the instance diagrams.
Figure 7.40: Client Agent sends the message and moves to ‘waiting-agree’ state. The mouse over was initiated before the relation was animated so the animation sequence is still empty.
Figure 7.41: Client Agent receives agree message and moves to ‘active’ state.
Figure 7.42: Client Agent receives the requested inform message and moves to ‘terminated’ state.
Figure 7.43: Mouse over text of the client side conversation after termination. Policies that are executed as part of the conversation instance are listed under “Animation Sequence” with an execution number and the event details.

Figure 7.44: Server Agent receives request-when message and moves to ‘waiting-agree’ state.
Figure 7.45: Server Agent sends agree message and moves to ‘active’ state.
Figure 7.46: Server Agent detects requested event and sends the inform followed by termination of the conversation.
7.2 Results

As illustrated, the implementation of a new conversation is fairly trivial when the user understands the functionality of the available types. Implementing a new conversation type from scratch would however be more taxing. The reason for this is that the method shown here didn’t necessarily require that the user had memorized all the Lisp commands that the policies and conversations depended on. On the other hand this tool does provided other conversation types and their code for reference so it would be possible to make new ones even for new, inexperienced, users.

Creating the Agent initialization files is also fairly easy to do especially since the user can work off of the already implemented conversation templates in CASA. In fact, the java methods used here were based on the TransientAgent class’ “Agent.Tell” method meaning that there can be a low amount of overhead for new users in this area as well.

Finally the animation window is especially easy to use as the user simply monitors for new conversations and can play their animation. In the event that an animation doesn’t proceed as expected the execution of policies can help determine where the conversation got stuck. Lacking that, the user is free to use their debugger and the Agent log files to determine why their policies were not executed as expected (though for the purposes of this demonstration we left any occurrences of errors out).

The tool therefore does indeed relieve the user of most responsibilities concerning development of Agent communications in regards to the technical aspects though there are still features to be desired which will be discussed in §8.1.3.

7.3 Related Work

As mentioned earlier there has been one other project concerned with visually debugging conversations in CASA. The Project was called MASVIS [Yee, 2010] and was
the thesis topic of Ryan Yee in 2010 (see Figure 7.47).

Figure 7.47: An overview of Ryan Yee’s MASVIS project taken from his Thesis [Yee, 2010].

The visualization in MASVIS used a mix of connected node diagrams with UML action diagrams employing concepts similar to this project’s information filtration and keyframe creation. The governing purpose was to facilitate debugging conversations in CASA through observation. For example, in it an error is described in which subscriptions did not persist properly. When a new Agent joined the simulation the other agents were not informed. This was evidenced by the lack of fan like message passing amongst agents after the new Agent had registered. The developers were thus able to find and fix the coding error regarding subscription conversations.

This approach is similar to other MAS visualizations in which agent groups and their interactions are the primary focus (see ARES [Denzinger and Kidney, 2005] and
the Agent Visualization System (Mostafa and Bahgat, 2005). These kinds of tools generally focus on using graphical and textual information to distinguish between different agents and their activities in order to confirm or disconfirm assumptions made regarding the macroscopic behaviour of the system.

The main difference between these projects and CASA UMLet is that the diagrams in CASA UMLet are focused on observing the progression of conversations and not the interactions of agents themselves. For instance, policies are not shown in MASVIS and the system does not aim to represent the actual code being used. Instead, the intent of MASVIS is to help developers ensure that the conversations have the desired macroscopic effects within agent groups rather than ensuring that the conversation has the desired local effect in an individual Agent.

7.4 Summary

This Chapter presented the use of the software in developing a new kind of conversation: request-when. This kind of conversation alerts a subscribing Agent when a certain event occurs then terminates.

To do this, the subscription conversation type was reworked to simply specify that termination occurred after the inform message was sent. The new types were then saved to a separate file of the user’s choosing.

Next, the action level diagram editor created Agent initialization files that contained concrete conversation templates using the new types. These simply specified that any event was what was watched for along with what Java methods to call in the Agent code base when a request-when was sent or received.

The two agents were then started in CASA and, using the tool, the user witnessed the message exchange and conversation termination.

Finally, this chapter provided a comparison to some other approaches to visualiz-
ing agent communications outlining their similarities and why they differ from what CASA UMLet provides.

Given the ease with which these new conversations were created, the amount of work the user is relieved of, and the benefit delivered by the animation for easy debugging we can conclude that the project is indeed useful for developing Agent Communications in a M.A.S design platform, namely CASA.

The next Chapter concludes this by discussing any drawbacks, unintended benefits, and future work.
Chapter 8

Conclusion

8.1 Goals and Implementations

The goal of this project was to provide a graphical interface through which users could develop and implement agent communications in CASA. The reason for this was that representing communication schemes purely in code tends to make the paradigms involved difficult to understand for new inductees. It was also mentioned that for such a tool to be useful the representations used had to be familiar to the intended user demographic. The tool also needed to omit certain details and allow the user to see and work with simplified versions of the presented interface. All these conditions needed to be met while allowing for the output of the graphical designs to CASA’s Lisp code base.

We designed the graphical interface around the open source project UMLet. The primary reason for this was that UMLet enables the graphical design and content management of UML diagrams. UML diagrams are used throughout various areas of software engineering to model such projects and systems. The second reason UMLet was chosen is that it is open source meaning that the code base is completely open for perusal and modification. Therefore it was easy enough to extend the project’s functionality as the brunt of the work needed for the interface had already been done.

Modifications to UMLet include the population and representation of various Lisp code pieces as UML diagram elements. These representations contain simplified views of the ideas the Lisp code pieces are meant to convey mixed with smaller sections of the code. The visible pieces of code can be modified through UMLet’s text editor and updated through modified popup menus. Updating can also provide the user with
warnings and/or error messages if the added parsing and logic utilities revealed any issues/errors. However, if the code is error free then it can be saved back to a correct Lisp representation through the parsing utility.

We simplified the code into a graphical representation in several ways. One way is that certain pieces of information were omitted in favour of the graphical representation given. For example, a conversation type is represented by a function definition or `defun` as was shown in Figure 2.9. The child elements of the function definition are wrapped in a call to the function ‘conversation’. This function call is not shown nor is it configurable without making changes to the ‘CasaUmlet’ project’s source code. Its presence, however, is implied by the fact that the node is a conversation node in the first place.

Another way we simplified code as graphics is that some pieces of information are represented through alternate means. For instance, the binding code relationships are instead represented by green relations between parent and child conversation nodes.

A third way we simplified code is actually seen in how extra information has been provided to the user. For instance, the blue state transition relations are computed by the project and do not actually occur anywhere in the code, rather, they are implied by the preconditions and postconditions of policies. Since this adds information rather than hides it (albeit with the goal that it assists the user) there also needs to be a way to remove such (as well as other) pieces of information from view. Thus, various visibility selectors were implemented to avoid overburdening the user with too much information.

Thus the project is as it is. Now follows a discussion on some of the benefits presented by the project, some drawbacks, and finally future work that can be done.
(defun foo
  (a a) ; two parameters by the same name
  (list a) ; list the value stored in the parameter
)

Figure 8.1: Function Definition with Duplicate Required Variables

8.1.1 Unintended Design Choices and Benefits

During development it became apparent to us that a conversation type could be defined recursively. This would be possible by allowing the conversation to have as a child either a direct call to itself or an indirect call via a call to some other conversation type. This presented some layout problems for the diagram that, although they could have been avoided, the question of whether a conversation instance should call itself recursively was brought up. An example of such a case is in a negotiation where an agreement is trying to be reached and the condition(s) for the agreement keep changing until either the negotiation succeeds or there is no more room for flexibility. On the other hand, some might argue that the negotiation logic should be part of the agent rather than the conversation. For instance, there is a chance that the negotiation is taking place between a creditor and many debtor agents simultaneously. If one debtor has already agreed to the original condition(s) then many cancelations must now be sent. This can be an issue when there are communication delays. It is possible that a creditor might agree to some debtor’s relaxed condition(s) before another debtor agrees to the original one(s). In addition, such a conversation would need to be given some knowledge of how to apply changes to the condition(s) as well as the condition(s) themselves. Such a solution may be seen as too complicated or out of scope for what should be a relatively simple task, namely a request. For all of these reasons (and perhaps others) conversations are forbidden to refer to themselves recursively either directly or indirectly.

Development of the parameter objects used by Lisp defuns revealed another issue.
In Lisp, it is perfectly legal for function definitions to have multiple parameters by the same name. Consider the function shown in Figure 8.1 where two identical parameters are named ‘a’ and the value of ‘a’ (whatever it may be) is displayed in a list. An invocation of this function such as ‘(foo 1 2)’ returns ‘(2)’ as the result since the number 2 was the last value applied to ‘a’. Now consider the function shown in Figure 8.2 where instead a keyed parameter is given the matching name. This means that we do not have to provide a value for the keyed parameter ‘a’ and note that 10 has been specified as the default. Thus a correct invocation of ‘(foo 1)’ has a similar result of ‘(10)’ as the number 10 was the last value applied. This last case is particularly dangerous as the error is present but the Lisp compiler does not throw any error or even a warning message. There are, of course, some cases where this behaviour is desirable. For instance, the parameter ‘a’ could be updating itself using some of the other parameters that have been read in. However since this tool is meant to make development as easy as possible duplicate parameters are forbidden and the user informed.

The last unintended benefit came from automated testing of the UML-Lisp integration which aimed to confirm the file outputs. This helped drive some of the design for the underlying theory in Lisp conversation type definitions. Recall Figure 2.9 in Chapter 2 §2.4. The code shows a parameter called ‘conversation-name’ as part of the ‘conversation’ command. As mentioned there, this is the runtime name of the
conversation when it is instantiated. Unfortunately this parameter is not defined in the parameter list shown and the section of code which requires the parameter is intended to be hidden from the UML representation. Thus if the parameter was not defined this project could create non-functional conversation code. This realization caused the final design to force the system to have this parameter defined as part of the ‘correct’ representation. It was then found that some of the conversation types already defined by CASA indeed had the ‘conversation-name’ parameter but this was used for naming policies rather than for the ‘conversation’ code section. Instead a parameter simply named ‘name’ was being used. Such cases were updated and now all conversation types use the same parameter by the same name.

8.1.2 Drawbacks of the Implementation

One of the biggest drawbacks of this project is that it is indeed possible for the user to write incorrect Lisp code as this system just checks syntax. For instance, if a function definition refers to variables that are not locally defined in the same file or any of the imports the user is not alerted. The reason for this is that it is perfectly legal in Lisp. Refer to figures 8.3 and 8.4. The first figure shows the contents of a Lisp file named ‘File1.lisp’. The file defines the function ‘foo’ which takes no parameters but depends on two variables ‘A’ and ‘B’ which have not been declared. The second figure shows the contents of a Lisp file named ‘File2.lisp’. This file first imports ‘File1.lisp’ then sets the values of A and B. The second file can be imported by a Lisp interpreter (or the lines it contains could be inserted verbatim) and the function ‘foo’ can then be called resulting in ‘3’ being returned as the result of 1+2. Therefore requiring that a variable or function is defined before they are used in a function definition is not desired as it would unnecessarily bombard the user with errors. A solution would of course be to highlight the values ‘A’ and ‘B’ in red (which would require some sort of working file suite) but at present UMLet has no support for
(defun foo  ;; define function foo
  ()       ;; no parameters
  (+ A B)  ;; add A and B
)

Figure 8.3: The contents of File1.lisp

(load 'File1.lisp)  ;; load file 1
(set a A 1)       ;; set value of A
(set b B 2)       ;; set value of B

Figure 8.4: The contents of File2.lisp

highlighting individual characters or words in its contents. Such a system is more
in line with an Integrated Development Environment like Eclipse then a Graphical
Interface like CasaUmlet. Therefore such cases cannot (at present) be controlled for
and will instead be deferred to future work in the following section. Thus, at present
we require that the user has a certain knowledge of the rules associated with Lisp and
programming in general (which is an implicit assumption of this project anyway).

Another drawback is that mutually exclusive sub conversations are not indicated
by conversation representations. To understand this, recall the definition of the request
conversation type from Table 2.1 where a request depends on a combination
of calls to ask, offer, and discharge. If a discreet request is performed then ask is
invoked, however if an offer is performed then offer is invoked. Both of these cases
set the conversation to different states and once either is set it is impossible for the
other to be used. However, the animation of such a conversation does not animate
state transitions in the unused sub conversation which, for now, is the only indication
of mutually exclusive sub conversations this project has.

Back in Chapter 4 the section regarding design point Editing Node Contents -
6 Editing Node Contents in Appendix C mentioned that conversations could
not be modified in the animation diagram. Time delays could of course be added
to CASA to ensure that the user has the time to modify the data structures of conversations or even if a conversation were ‘stuck’ and we wished to force termination the ability to modify the object through this project’s interface would of course be useful. Furthermore, the definition of new conversation types as well as new concrete conversations during an Agent’s lifetime (as opposed to terminating, designing, and restarting) could have a large impact on user efficiency. Unfortunately the system was not designed with these thoughts in mind as this was the first iteration therefore this is also deferred to future work in the following section.

A final drawback is that the consequents of policies do not show any information regarding social commitments nor are they evaluated. This means that the user cannot see what social commitments are created and destroyed and the system does not ensure that social commitments are destroyed (and perhaps this is desirable in some cases). This last drawback mostly describes a possibility for future work which now follows.

8.1.3 Future Work

As mentioned at the end of the last section social commitment addition and removal could be visualized in the diagrams as well. One method would be for rounded squares to appear below policies that caused social commitments to be added within their consequents. The contents of these could just be text with either a + or - along with the text SC paired with a number generated by the system to associate with the social commitment. This would be a fairly trivial implementation but was not necessary for the overall system to function.

In the interest of getting CASA UMLet more in line with a traditional IDE (in that variable and method ‘scope’ errors can be supported) there are two routes to consider. One is to wait for a future release of UMLet (which may or may not be open source) to add support for modifying pieces of the text contained by UML objects
(highlighting, underlining, etc.). The other is to delve further into UMLet’s source code and add the required functionality ourselves. The drawback of waiting is of course that it may never be added by the UMLet development team. The drawback of modifying the source code directly is that certain classes may become deprecated (which actually happened during the move to Java 8 requiring an upgrade to the latest UMLet and this project). In the end, it may be best to either leave these kinds of tasks as unsupported, choose another method in which ‘scope’ errors could be monitored, or attempt to to write a new graphical framework that met this (rather minor) purpose and perhaps others. In the end, there is nothing to stop this from being done provided errors were shown through alternate means like mouseover error text that indicates its presence by colouring nodes red. The ‘scope’ check can be achieved fairly easily since similar functionality is already in place in which one can import other files in order to use their conversation types. All that really needs to be done to meet this criteria is to consider the best way to represent/achieve these aspects.

In the last section it was also noted that users were not made aware of mutually exclusive conversations. By analyzing state transitions (which the system already generates) it would be possible to detect these cases but a method with which to make this visible is not yet developed. One way would be to insert a red relation in-between the label sections of the child nodes in question but there could still be better ways to visualize it as well.

In regards to modifying active conversations in the animation diagram one needs only to implement the update routine to update the values of data structures as well as make a slight modification to CASA that enforces a minimum time delay between the evaluation of events. Also, in order to create new conversations during an Agent’s runtime all that needs to be considered is where to save newly created conversation
types and templates (or perhaps not even bother) and execute the Lisp code.

Another addition that could be valuable has to do with the code generated by the system. For instance, a policy must use a Lisp function of the same form shown in Chapter 2 Figure 2.13. If changes were ever made to CASA’s definition of policies then this project’s code base must be updated. On top of this, the project could, hypothetically, write to any other code base instead. To do this we would only need a configuration file defining the code for each element. This file could also specify containment rules, how to treat the sections of the code, and alternate output languages. Unfortunately the project has not been designed to make this particularly easy to do but it is still feasible and would only require swapping out the behaviour used by node objects and the file read/write utilities.

A final point for development has to do with code and comment generation. It may be beneficial that when a user adds a policy some of the sections could be populated by default code with automatically generated comments to match. This is actually a fairly trivial thing to do but would best be implemented after the codebase configuration files are in place.

Lastly, it was noted in Chapter 5 that there is a known bug with a workaround that is a result of using UMLet. When UML elements are double clicked they create copies of themselves which may or may not be desired. It could be that such a mechanism could be useful and as such the code responsible has not been removed since if UMLet is ever updated there will be less code to change.

8.2 Concluding Remarks

In the end, we meant for the project to create a simplified way to develop agent communications in CASA, which it does as demonstrated by Chapter 7. The project: provides an intuitive, familiar, and interactive visualization for the viewing and editing
of CASA’s Lisp files; enforces certain coding practices to help avoid errors; alerts the user to the most relevant kinds of errors; and creates code that is easily read by future users. On top of these points the project was also developed through good practices such as the many forms of testing employed throughout its construction ensuring that it is useable as well as useful. Also, the fact that UMLet has been modified minimally makes further extensions and changes easy for future developers. However a quick discussion and defence of its consequences, drawbacks, and future is warranted.

The drawback that noted technical consequences of using Lisp (§8.1.2 on multiple files and in §8.1.2 on duplicate variables and function recursion) shouldn’t be of much concern. As these conversations are fairly straightforward, users aren’t required to understand or use the more advanced programming practices Lisp has to offer anyway. Note that we are concerned with communicating the ideas behind these communication schemes rather than educating all of the ways Lisp can be used and therefore such drawbacks and caveats are somewhat exaggerated.

The last two drawbacks continued into the future work sections. Of those, the most beneficial would be social commitment visualization, however the note on configuration files would definitely come in handy should CASA change. As noted in that section the suggested changes are indeed possible and for the most part trivial, though not in general necessary. Recall that the project successfully demonstrated its ability to design and implement communication in Chapter 7.

Finally, the design practices this project enforced could be seen as both useful and harmful to CASA. As mentioned before, they do cause the entire communication scheme to become standardized so that all code is consistent. On the other hand such restrictions also limit the flexibility with which these conversations can be written (though not by much). If however a user wished to disobey these conventions they can still of course write the code themselves without the use of the project. Note that
such a user is typically considered to be an *advanced* user (since they presumably know of and can see a better way) and not a *new inductee* to the field of Agent Communications and CASA for whom this project is primarily intended.

Thus the project performs as expected. Options for continued development are available and the results of its use contain few (if any) drawbacks/consequences.

### 8.3 Summary

This Chapter concluded this work on designing Agent Communications through a visual medium employing UML diagrams within the CASA framework. Drawbacks of the software were discussed along with some of the benefits its development has provided. Finally, areas of future work were mentioned before concluding that all goals delivered in Chapter [1] were met and that a useable and useful tool has been developed.
Bibliography


141


Appendix A

CASA Client-Side Request Conversation

The following images are CASA’s request-client conversation defined in Lisp:

```
(defun request-client
  ( ; will be named: the-act+"-request-client"
    the-act
    result-action ; when the server responds with propose/discharge
    execute this
    &key
    (request-performative request)
    (request-act the-act)
    (base-name "request-client")
    (conversation-name (concatenate 'string the-act "-" base-name))
    ;; for the ask-client conversation
    ; setting this gives the agent a chance to AGREE or REFUSE a proposal
    (propose-decision '((performdescriptor 0 :performative refuse))
    ; the action to perform when we receive a AGREE reply to our REQUEST
    agree-action
    ; the action to perform when we receive a REFUSE reply to our REQUEST
    refuse-action '((exception-handler ,conversation-name
                     "received unexpected refuse"))
    ; the action to perform when we receive a NOT UNDERSTOOD reply to our
    REQUEST
    (not-understood-action '((exception-handler ,conversation-name
                             "received unexpected not-understood"))
    (timeout-action '((exception-handler ,conversation-name
                      "received unexpected timeout"))
    ;; for the discharge-client conversation
    (failure-action '((exception-handler ,conversation-name
                      "received unexpected failure"))
    (timeout-discharge-action '((exception-handler ,conversation-name
                                 "received unexpected timeout-discharge"))
    nopropose
    optional-negotiation
    ;; the content language in the message-to-match
    (content-language NIL)
  &aux
    (ask-name (concatenate 'string conversation-name "-ask"))
    (offer-name (concatenate 'string conversation-name "-offer"))
    (approver-name (concatenate 'string conversation-name "-approver"))
    )
)```

Figure A.1: Request Conversation Lisp Representation - Parameters
"Creates a Conversation for a client side request, supporting:
1. an outgoing REQUEST while in state not-started (→started)
   and incoming messages AGREE, REFUSE or not-understood when in state
   started (→terminated);
or
2. an incoming PROPOSE while in state not-started (→started)
   and an outgoing AGREE or REFUSE when in state started (→terminated)
"

; outgoing request (agent−GLOBAL policy): create this kind of
conversation
(agent.put−policy
 (policy
   '{(msgevent−descriptor
      event_messageSent
      ::= performative , request−performative
      :act (act , the−act)
      :language , content−language
   )
   '{((agent.instantiate−conversation , conversation−name (event . get)))
   (concatenate ' string "Conversation2 global policy: For an
      outgoing request/"
      the−act ", instantiate a client conversation, request−
      client−001")
   :name (concatenate ' string "request−client−000 (" base−name ")."
      the−act)
   ); end policy
 )
)

; incoming propose (agent−GLOBAL policy): create this kind of
conversation
(if nopropose
 ()
 (agent.put−policy
 (policy
   '{(msgevent−descriptor
      event_messageReceived
      ::= performative propose
      :act (act , the−act)
      :language , content−language
   )
   '{((agent.instantiate−conversation , conversation−name (event . get)))
   (concatenate ' string "Conversation2 global policy: For an
      incoming propose/"
      the−act ", instantiate a client conversation, request−
      client−002")
   :name (concatenate ' string "request−client−001 (" the−act)
   ); end policy
   )
 )
)

Figure A.2: Request Conversation Lisp Representation - Documentation and Global Policies
; the actual composite conversation
(conversation conversation−name
(list
 (ask−client
  ask−name
  the−act
  :request−performative request−performative
  ; ;propose−decision propose−decision
  :agree−action agree−action
  :refuse−action refuse−action
  :not−understood−action not−understood−action
  :timeout−action timeout−action
)
; ;transformation transformation
;optional−negotiation optional−negotiation
)
(if noPropose()
 (offer−client
  offer−name
  the−act
  :propose−decision propose−decision
  ; ;agree−action agree−action
  ; ;refuse−action refuse−action
  ; ;not−understood−action not−understood−action
  ; ;timeout−action timeout−action
  ; ;transformation transformation
  )
)

(discharge−client
 approver−name
 the−act
 result−action
 :request−performative request−performative
 :failure−action failure−action
 :timeout−action timeout−discharge−action
 ; ;transformation transformation
 ;optional−negotiation optional−negotiation
)
)

Figure A.3: Request Conversation Lisp Representation - Delegate Calls
:bind-state (append
  (("init", ask-name "init")
   ("waiting-request", ask-name "waiting-request")
   ("terminated", ask-name "terminated")
   ("waiting-discharge", ask-name "terminated-pending")
  )

(if nopropose
  ()
  (("init", offer-name "init")
   ("waiting-propose", offer-name "waiting-propose")
   ("terminated", offer-name "terminated")
   ("waiting-discharge", offer-name "terminated-pending")
  )
)

(("init", approver-name "blocked-init")
 ("waiting-request", approver-name "blocked-request")
 ("waiting-propose", approver-name "blocked-propose")
 ("waiting-discharge", approver-name "init")
 ("waiting-propose-discharge-reply", approver-name "waiting-propose")
 ("terminated", approver-name "terminated")
)

Figure A.4: Request Conversation Lisp Representation - State Binds
Figure A.5: Request Conversation Lisp Representation - Local and Non Local Variable Bindings
Appendix B

Project Requirement Points

B.1 Visualization Requirements Points

This section lists the requirements that the ideal visualization tool should have.

Vis - 1 Graphical Representation

The ideal system shall provide us with a visual representation of the code in a file that is fairly easy to understand by new users, particularly those with a background in Computer Science.

Possibilities include any 2D/3D models such as connected graphs, trees, or UML. They do not include things from other fields such as cells from biology.

Vis - 2 Different Diagrams for Different Uses

We could, hypothetically, implement all types of editing and observing in one diagram but it would most likely be too difficult for a user to easily distinguish between the various activities the tool aims to provide. In that interest, different diagrams will become associated with what they can edit and/or observe.

Vis - 2.1 Speech Act editing

Allows for conversation types to be added to and edited in the current working conversation types file, for other types files to be imported and referenced, and for the working types file to be saved to a Lisp file.

Vis - 2.2 Action editing

Allows for templates (the calls to conversation types that create concrete conversation objects) to be added and edited to a lisp file that agents can
run at startup. Recall that these are the known conversations an Agent can engage in and not the delegate conversations used by defuns.

**Vis - 2.3 Instance viewing**

Allows for the visualization of the conversation the observed Agent has or is currently participating in.

This simply requires that there be three different views for the tool.

**Vis - 3 Objects to be Represented**

Using a representation that meets **Vis - 1 Graphical Representation** we must employ some intuitive means with which to represent the important pieces of information.

**Vis - 3.1 Conversation Type Representations**

Conversation Types shall be represented by elements labeled by their name and containing all of their variable-value pairs and possible states.

**Vis - 3.2 Policy Representations**

Policies shall be represented by elements labeled by their name, accessibility, and ghost status and shall contain their antecedent, precondition, postcondition, and consequent(s) code.

**Vis - 3.3 Instantiated Conversation Representations**

Instantiated conversations (conversations that have occurred within the context of an Agent) shall be represented by elements matching those for regular conversations but shall not be editable at all by the user.

**Vis - 3.4 Conversation Template Representations**

Conversation templates (type calls that make concrete conversations) shall be represented by elements matching those for regular conversations. They shall indicate that they and all of their children are not editable (as they
are simply function calls) but the values of any override-able parameters shall be editable.

**Vis - 3.5 Agent Representation**

If an Agent is defined then it shall be represented by an element labeled by the Agent name and shall indicate what conversations have been executed and what files the Agent has loaded at startup.

**Vis - 3.6 File Membership Indication**

All elements that are members of a file shall indicate which file they are a member of.

**Vis - 3.7 Child Containment Indication**

Similar to file membership, all elements that are children of (or contained by) a parent element’s definition shall have some way of indicating the relationship.

**Vis - 3.8 Child Containment Order**

All elements that are contained by a parent element’s definition have some way of indicating their order of declaration within the definition.

**Vis - 3.9 Conditional Representations**

Conditionals that control the structure of a conversation through the evaluation of a condition shall be represented by elements labeled by the condition. The non comment child nodes (of which their can only be two) indicate which case evaluation they are a part of (i.e. True case or False case).

**Vis - 3.10 State Transition Representations**

If a conversation has any child elements that are policies, some way of indicating possible execution paths with respect to state transitions shall be present.
Vis - 3.11 Equivalence Representations
   If a conversation has any child elements that are conversation calls, than any shared (or equivalent) variables and states shall somehow be made obvious.

Vis - 3.12 Documenting Use Representations
   Documentation which states the intended use and workings of a conversation template or policy shall be visible someplace.

Vis - 3.13 Documenting Comments Representations
   Comments that appear to also document the conversation template or policy shall be visible someplace.

Vis - 3.14 Code Fragment Comment Representations
   Comments that appear to document a conversation template’s variables or policy’s section shall be visible someplace and clearly associated with the given variable or section.

Vis - 3.15 Stray Comment Representations
   Stray sections of comments that do not appear to be associated with any pieces of code are present as elements in the diagram that show the collected comments.

Vis - 3.16 Stray Lisp Code Representations
   Stray pieces of lisp code that do not conform to either policies, conversation templates, nor conversation calls, are present as elements in the diagram that show the corresponding code.

Vis - 3.17 Clear Diagrams
   Diagrams shall be clear of excess information. Thus, elements shall have the ability to be removed from view when they are not of concern to the
user.

Requirements **Vis - 3.1 Conversation Type Representations** to **Vis - 3.17 Clear Diagrams** can use any combination of colors, relations, emphasis typography, pop-up dialogues, numbering, labeling, content text, and mouse over text.

**Vis - 4 User Interaction**

The objects represented shall be manipulatable through user interaction.

**Vis - 4.1 Create, Modify, and Remove**

In the case of conversation types, template conversations, global policies, generic lisp code blocks, and comments the user shall be able to modify/remove existing ones and create new ones.

**Vis - 4.2 Change Position**

Also, the user shall be able to change their position in their containing code base (when parented by a file or conversation type) since execution order can be important.

Possibilities for requirements **Vis - 4.1 Create, Modify, and Remove** and **Vis - 4.2 Change Position** include pop-up dialogues, menus, mouse/keyboard interaction.

B.2 Error Reporting Requirements Points

**ErrRep - 1 Error Reporting Requirements**

User generated errors shall be caught. This could include anything from malformed Lisp generated by the user to warning that supplied postconditions are not possible (as far as the system can tell at least). These errors, however,
shall not prevent the user from having the final say and so shall not enforce any decision making rather, they shall aid and inform the user.

B.3 Saving Requirements Points

Once the modifications to conversation types have been visually completed, or the desired template conversations have been created, it is of course a requirement that we shall be able to save this new representation such that it can be used by CASA and CASA agents.

Save - 1 Save To File

Save newly created conversation types or template conversations to a file of the user’s choosing.

This can be a popup file saver or configuration variables supplied at startup. In the case of editing an already existing file the default save path is the path to that file, otherwise the user’s working directory shall be used.

Save - 2 Visualization to Lisp

When saving, the graphical representations shall be transformed back into Lisp code.

This involves having a transformation mechanism that uses the current Lisp code representations of each graphical object and their relevant parameters.

Save - 3 Readable Generated Lisp Code

The Lisp code resulting from the transformation shall be readable by anyone who wishes to edit the code directly.

Save - 3.1 Pretty Printing Lisp Code

The Lisp code generator shall employ ‘pretty printing’ to ensure that the
produced code is human readable.

The Lisp code generator shall simply use some heuristic to insert newlines and tabbing whenever appropriate.

**Save - 3.2 Correct Comment Placement**

The Lisp code generator shall take into account user comments associated with various sections of code preserving their position and not resulting in commenting any additional code.

To achieve this, some kind of tracking for comments needs to be in place in which comment sections are associated with code fragments.

**Save - 3.3 Consistent Code**

A general trend in the representation of Lisp code shall be preserved so that certain sections appear consistent with each other such as policies, variables, etc.

This is similar to Save - 2 Visualization to Lisp but with more constraints as to how the default representations shall be applied. For example, a policy’s consequent shall always appear on their own line (and tabbed appropriately) immediately after the antecedent has finished printing. Though the antecedent may be longer or shorter depending on the policy (requiring newlines be inserted) the consequent shall always occur after a newline (after the antecedent) for all policies.

**B.4 Animation Requirements Points**

When an Agent is present, the tool shall allow for debugging of conversations in the diagram described by Vis - 2.3 Instance viewing. This means that once we have created new conversation types and provided the Agent with concrete conversations
defined by new (or old) templates using these new (or old) types we shall be able to visualize how these conversations get executed when the Agent engages in them.

**Anim - 1 Conversation Monitoring**

When the program is run from within the context of an Agent, it shall register itself as an observer for any conversations the Agent engages in.

This can be achieved through the well known ‘observer’ design pattern [Lethbridge and Laganiere, 2005, Sec. 6.6] which is supported by agents in CASA.

**Anim - 2 Conversation KeyFrames**

Objects representing observed conversations shall collect the sequence of policies executed during their life cycle (initialization to termination).

This can be achieved through the use of ‘keyframes’ in an animation sequence. Keyframes are simply steps in the sequence containing information about the condition of the object with respect to a point in the sequence. [Blender Foundation, 2015]

**Anim - 3 KeyFrame Markups**

An animation indicates which policies were executed, why, and the resulting state transitions of the conversation.

This could be labelling on the parent-child relationship indicators defined by requirement [Vis - 3.7 Child Containment Indication] colour changes in the objects or relationship indicators, or through the use of emphasis typography such as bolding, underlining or italicizing on any text contained within the objects.
Appendix C

Project Design Points

Diagrams - 1 Different Diagrams

Supporting Vis - 2 Different Diagrams for Different Uses in Chapter 3, B.1 is trivial as we simply need to have separate diagrams for each of the three cases. Requirement Vis - 2.1 Speech Act editing requires a diagram that specializes in editing the speech act conversation types which the concrete conversations’ templates depend on. Requirement Vis - 2.2 Action editing requires a diagram that specializes in editing the templates used to define concrete conversations that an Agent can know about and adding them to a file. Requirement Vis - 2.3 Instance viewing requires a diagram that specializes in displaying an animation of the past conversations the observed Agent has engaged in or is still engaging in. Therefore we shall:

Diagrams - 1.1 Speech Act Diagram

Dedicate a diagram to the speech act level editor for creating conversation types (Vis - 2.1 Speech Act editing).

Diagrams - 1.2 Action Diagram

Dedicate a diagram to the action level editor for creating templates that instantiate concrete conversations (Vis - 2.2 Action editing).

Diagrams - 1.3 Instance Diagram

Dedicate a diagram to the instance level viewer for observing past and presently active conversations (Vis - 2.3 Instance viewing).

ElemDefs - 2 Element Definitions
Before providing a design that meets all of requirement Vis-4 User Interaction in the template and action level diagrams we need to be sure that we can edit the contents of the UML nodes easily and that any changes are reflected in the diagram. To that end we must know what we are editing and therefore we must first provide a design for the various elements listed by requirement Vis-

3 Objects to be Represented

ElemDefs - 2.1 Conversation Type

Requirement Vis-3.1 Conversation Type Representations requires that conversation types appear as their own nodes labeled by their name and containing all of their variable-value pairs and possible states.

ElemDefs - 2.1.1 Conversation Type Labelling

We will insist that the label of this kind of node is the name of the conversation template.

ElemDefs - 2.1.2 Conversation Type Contents

We will insist that the content of this kind of node represents the Lisp code representing all the variables, their values, and the states that the conversation can be in.

ElemDefs - 2.1.3 Conversation Type Colour

We will choose a colour for this node that is not used by other nodes.

ElemDefs - 2.1.4 Conversation Type Diagram Use

The elements used to refer to these types are meant to be editable (changes to their parameters, to their state/variable bindings, to the template names) and so will only be used in the speech act level editor.

ElemDefs - 2.2 Policies
Requirement **Vis - 3.2 Policy Representations** requires that policies appear as their own nodes labeled by their name, accessibility, and ghost status and shall contain their antecedent, precondition, postcondition, and consequent(s) code.

**ElemDefs - 2.2.1 Policy Labelling**

We will insist that the label of this kind of node contains the name, accessibility, and ghost status.

**ElemDefs - 2.2.2 Policy Contents**

We will insist that the content of this kind of node represents the Lisp code representing the antecedent, precondition, postcondition, and consequent(s).

**ElemDefs - 2.2.3 Policy Colour**

We will choose a colour for this node that is not used by other nodes.

**ElemDefs - 2.3 Instance Conversations**

Requirement **Vis - 3.3 Instantiated Conversation Representations** requires that instantiated conversations (conversations that have occurred within the context of an agent) shall be represented by elements matching those for conversation types and templates but shall not be editable at all by the user.

**ElemDefs - 2.3.1 Instance Conversation Form**

Note that the form of these conversations depends on the runtime object’s Lisp code and child object lists so no matching of the conversation type needs to be performed.

**ElemDefs - 2.3.2 Instance Conversation Labelling**

We will insist that the label of this kind of node is the name of the runtime object rather than the conversation type used. The name
depends on the evaluation of the parameter 'conversation-name' used by the Lisp code shown in Chapter 2 § 2.4 in Figure 2.9. This parameter is used to name the runtime object to help distinguish it from other runtime conversations.

**ElemDefs - 2.3.3 Instance Conversation Contents**

We will insist that the content of this kind of node is taken from the runtime object’s Lisp code which has already been evaluated. Thus all the variables, their values, and the states that the conversation can be in are parsed from such a representation and are what’s included as the content.

**ElemDefs - 2.3.4 Instance Conversation Colours**

We will choose a colour for this node that matches the colours in designs **ElemDefs - 2.1.3 Conversation Type Colour** and **ElemDefs - 2.4.4 Template and Sub Conversation Colours**.

**ElemDefs - 2.4 Conversation Templates/Sub Conversations**

Requirement **Vis - 3.4 Conversation Template Representations** requires that conversation calls appear as their own nodes matching those for conversations types while indicating that they and all of their children are not editable. The exception to this is that the values supplied to the override-able parameters (required, optional, and key) in the conversation call can be edited.

**ElemDefs - 2.4.1 Template and Sub Conversation Form**

Note that the form of these conversations will require matching them against one of the conversation types. If no matching type can be found (the type being from a file that is not currently imported) then the conversation call will appear to be stray Lisp code and
shall therefore default to a stray Lisp code node.

**ElemDefs - 2.4.2 Template and Sub Conversation Labelling**

We will copy the labelling from what’s defined in design [ElemDefs - 2.1 Conversation Type](#).

**ElemDefs - 2.4.3 Template and Sub Conversation Contents**

We will copy the content from what’s defined in design [ElemDefs - 2.1 Conversation Type](#).

**ElemDefs - 2.4.4 Template and Sub Conversation Colours**

We will choose a colour for this node that matches the colour of design [ElemDefs - 2.1.3 Conversation Type Colour](#).

**ElemDefs - 2.4.5 Sub Conversations in Speech Act Diagram**

These nodes are used by conversation types in the speech act diagram since types can call other types supplying the delegates with parameters.

**ElemDefs - 2.4.6 Templates in Action Diagram**

These nodes are also used by the action diagram file nodes as these conversation calls are executed by an Agent at startup so that the Agent knows how to engage in the corresponding concrete conversation.

**ElemDefs - 2.4.7 Template and Sub Conversation Editability**

Indicate non-editable status (ie. the removal/addition of parameters) through paler colours.

**ElemDefs - 2.5 Agent Nodes**

Requirement [Vis - 3.5 Agent Representation](#) requires that if an agent is defined then it shall be represented by an element labeled by the agent name and shall indicate what conversations have been executed and what
files (containing conversation templates and/or global policies) the agent has loaded at startup.

**ElemDefs - 2.5.1 Agent Node Labelling**

We will insist that the label of this kind of node is the name of the agent.

**ElemDefs - 2.5.2 Agent Node Contents**

We will insist that the content of this kind of node is empty since it shan’t be doing anything else.

**ElemDefs - 2.5.3 Agent Node Colours**

We will choose a colour for this node that is not used by other nodes.

**ElemDefs - 2.5.4 Agent Node Context**

This node will only be present when the tool is running in an active agent context.

**ElemDefs - 2.5.5 Agent Node in Speech Act Diagram**

Since this node has nothing to do with the contents of the speech act diagram it will not be placed there.

**ElemDefs - 2.5.6 Agent Node in Action Diagram**

Since the action level diagram is what’s concerned with files and their contents the Agent node shall appear there using containment relations to indicate the files (and thus the kinds of concrete conversations) it knows about (or that are contained within its definition having been loaded at startup).

**ElemDefs - 2.5.7 Agent Node in Instance Diagram**

Since the instance level diagram is what’s concerned with executed/executing conversations and their animation the Agent node shall appear there using containment relations to indicate the conversations...
tions that are currently contained within its history or that are current active.

**ElemDefs - 2.6 File Membership**

Requirement [Vis - 3.6 File Membership Indication] requires that all nodes which are members of a file indicate which file.

**ElemDefs - 2.6.1 Instance Diagram Files**

Since the instance level diagram is only for conversations that have occurred or are presently occurring there is no need for file information.

**ElemDefs - 2.6.2 Action Diagram Files**

Since the action level editor is concerned with viewing and building multiple files containing templates for concrete conversations (because at startup an Agent can import many) it is necessary that File Nodes be created and designed:

**ElemDefs - 2.6.2.1 File Node Labelling**

We will insist that the label for a File Node is the filename.

**ElemDefs - 2.6.2.2 File Node Contents**

We will insist that the content of this kind of node is empty since it shan’t be representing anything else.

**ElemDefs - 2.6.2.3 File Node Colours**

For File Nodes we will choose a colour for this node that is not used by other nodes.

**ElemDefs - 2.6.2.4 File Node Members**

Nodes that have membership to a file node shall indicate this using a containment relation (see design [ElemDefs - 2.7 Parent Node Membership]).
ElemDefs - 2.6.3 Speech Act Diagram Files

Since the speech act level editor is concerned with creating/adding and modifying conversation types it is desirable that the UML diagram area remain clear of as many obstructions as possible. (Requirement Vis - 3.17 Clear Diagrams)

Therefore the file that contains the conversation types (whose types are being updated, added to, and removed) shall not be represented in this diagram. Instead, it shall be represented elsewhere. Possibilities include concrete ones (like dedicating another diagram that contains the file(s)) or abstract ones (like UI dialogues).

ElemDefs - 2.6.4 Speech Act Diagram Imports

It is desirable that we can work with multiple speech act files and so some way of representing and having access to other speech act files containing conversation type is needed while obeying the point made in ElemDefs - 2.6.3 Speech Act Diagram Files.

ElemDefs - 2.7 Parent Node Membership

Requirements Vis - 3.7 Child Containment Indication requires that any child nodes of a parent node (contained by) shall indicate parentage.

To do so we shall use a containment relation between the nodes.

ElemDefs - 2.8 Position of Child Elements

Requirement Vis - 3.8 Child Containment Order and Vis - 3.6 File Membership Indication both refer to nodes and representing membership, however, they do not indicate anything in regards to where the child elements are to be placed with respect to the parent (though perhaps possibilities have been implied).

We will therefore commit that child nodes appear below the parent nodes.
in keeping with the hierarchical relationship of tree diagrams as in Figure 4.1.

**ElemDefs - 2.9 Order of Child Elements**

Requirement [**Vis - 3.8 Child Containment Order**](#) requires that nodes which are contained by another node have some way of indicating their order of declaration.

Thus we will order the child nodes from left to right where leftmost occurs first within the parent’s definition and rightmost is last.

**ElemDefs - 2.10 Conditionals**

Requirement [**Vis - 3.9 Conditional Representations**](#) requires that structure affecting conditionals appear as their own node labeled by the condition. The non comment child nodes (of which there can only be two) indicate which case evaluation they are part of (i.e. True case or False case).

**ElemDefs - 2.10.1 Conditional Labelling**

We will insist that the label of this kind of node is the code representing the condition.

**ElemDefs - 2.10.2 Conditional Contents**

We will insist that the content of this kind of node is empty since it shan’t be doing anything else.

**ElemDefs - 2.10.3 Conditional Colours**

We will choose a colour for this node that is not used by other nodes.

**ElemDefs - 2.10.4 Conditional True/False Relations**

True case vs. False case child nodes will have a conditional relation (see design [**ElemDefs - 2.7 Parent Node Membership**](#)) from the parent conditional and will place an indication on the relation in the
form of a label. The relations could also be distinguished by having different colours.

**ElemDefs - 2.11 State Transitions and Possible Execution Paths**

Requirement [Vis - 3.10 State Transition Representations] requires that if a conversation has any child nodes that are policies, some way of indicating possible execution paths with respect to state transitions shall be present.

To indicate possible execution paths between policies we will use a *transition* relation.

**ElemDefs - 2.12 Equivalent States and Variables**

Requirement [Vis - 3.11 Equivalence Representations] requires that if a conversation has any child nodes that are conversation calls, than any shared (or equivalent) variables and states shall somehow be made obvious.

To indicate that two states/variables in two different conversations are considered the same (or equivalent) we will use an *equivalence* relation.

**ElemDefs - 2.13 Documentation Information**

Requirement [Vis - 3.12 Documenting Use Representations] requires that documentation associated with a conversation type, conversation template, or policy shall be visible someplace.

For this we can place the documentation as mouse over text when the user’s cursor is above the node’s label.

**ElemDefs - 2.14 Documenting Comments**

Requirement [Vis - 3.13 Documenting Comments Representations] requires that comments that document a conversation type, conversation template, or policy shall be visible someplace.
For this we can add the comments to the mouse over text in design [ElemDefs - 2.13 Documentation Information].

**ElemDefs - 2.15 Smart Comments**

Requirement [Vis - 3.14 Code Fragment Comment Representations] requires that comments that appear to document a conversation type's variables or policy’s section shall be visible someplace and clearly associated with the given variable or section.

Similar to design [ElemDefs - 2.14 Documenting Comments] however when the cursor is above the variable or section within the node’s content the corresponding comments shall appear as the mouse over text.

**ElemDefs - 2.16 Stray Comments**

Requirement [Vis - 3.15 Stray Comment Representations] requires that stray sections of comments appear as their own nodes with the contents being the comments collected.

**ElemDefs - 2.16.1 Stray Comment Labelling**

We will insist that the label of this kind of node is simply a generic name for comment nodes.

**ElemDefs - 2.16.2 Stray Comment Contents**

We will insist that the content of this kind of node is the collected comments.

**ElemDefs - 2.16.3 Stray Comment Colour**

We will choose a colour for this node that is not used by other node types.

**ElemDefs - 2.17 Stray Lisp Code**

Requirement [Vis - 3.16 Stray Lisp Code Representations] requires that stray pieces of Lisp code appear as their own nodes with the contents
being whatever the Lisp code is.

**ElemDefs - 2.17.1 Stray Lisp Code Labelling**

We will insist that the label of this kind of node is simply a generic name for code nodes.

**ElemDefs - 2.17.2 Stay Lisp Code Contents**

We will insist that the content of this kind of node is the captured Lisp code.

**ElemDefs - 2.17.3 Stray Lisp Code Colour**

We will choose a colour for this node that is not used by other nodes.

**ElemDefs - 2.18 Minimized Complexity**

Requirement **Vis - 3.17 Clear Diagrams** states that elements shall have the ability to be removed from view when they are not of concern to the user. This is achieved in each diagram in the following ways:

**ElemDefs - 2.18.1 Remove Node from Speech Act Diagram**

For the speech act level diagram, if an element is removed from view than it is removed from the diagram’s list of open (and thus editable) elements.

**ElemDefs - 2.18.2 Remove Node from Action Diagram**

For the action level diagram, if a global policy, conversation call, file node, or agent node, is removed from view it is only hidden from view and is thus recoverable through menu options discussed in design **ActDiagMenus - 4.2 Show Hidden Nodes**.

**ElemDefs - 2.18.3 Remove Node from Instance Diagram**

For the instance level diagram, if an executed conversation is removed from view then it is removed from the diagram’s list of past conversations and therefore can no longer be executed.
ElemDefs - 2.19 Node Edit-ability

Note that requirement Vis - 3.4 Conversation Template Representations also indicates that non editable nodes (ones that are or are part of a conversation template/sub conversation call) need to indicate this somehow. To do this something like paler colours shall be used.

ElemDefs - 2.20 Instance diagram nodes

Any nodes that occur in the instance level shall be assumed by the user not to accept any user input and so can just use the conventional colours.

TempDiagMenus - 3 Speech Act Diagram Menus

For the speech act diagram defined by design Diagrams - 1.1 Speech Act Diagram we need to support right click functionality that is specific to the various nodes and the speech act diagram’s background.

TempDiagMenus - 3.1 Open/Add Top Level Node

The background shall have the ability to open/add conversation types, stray comments, and stray Lisp code blocks. (Requirement Vis - 4.1 Create, Modify, and Remove).

TempDiagMenus - 3.2 Save Current File

The background shall have the ability to save the current speech act file to a specific file. (Requirement Save - 1 Save To File).

TempDiagMenus - 3.3 Import File

The background shall have the ability to open other speech act files that may exist so they can be referenced by the current active speech act file (the one that is editable).

TempDiagMenus - 3.4 Change File

The current active file shall be able to change to another file.
TempDiagMenus - 3.5 Reorder Nodes in File

The conversation type collection shall allow the various elements it contains to be reordered. (Requirement Vis - 4.2 Change Position)

TempDiagMenus - 3.6 Delete Nodes from File

The conversation type collection shall allow the various elements it contains to be deleted. (Requirement Vis - 4.1 Create, Modify, and Remove)

TempDiagMenus - 3.7 Add/Delete Child Nodes

Top level conversation types and conditionals (ones that are not members of a sub conversation call) need the ability to have policies, conditionals, sub conversations, stray comments, and stray Lisp code blocks added to them and removed from them. (Requirement Vis - 4.1 Create, Modify, and Remove)

TempDiagMenus - 3.8 Reorder Child Nodes

All child nodes of an editable conversation type (one that is top level) shall be able to change their declaration order within the data structure. (Requirement Vis - 4.2 Change Position)

ActDiagMenus - 4 Action Diagram Menus

For the action diagram defined by design Diagrams - 1.2 Action Diagram we need to support right click functionality that is specific to the various nodes and the action diagram’s background.

ActDiagMenus - 4.1 Open/Save File

The background shall have the ability to open existing conversation template files, create new ones, and save them to a specific file (Requirements Vis - 4.1 Create, Modify, and Remove and Save - 1 Save To File).
Note: Creating/saving shall perform error checking in case of file overwrite or if a file by that name is already open (Requirement ErrRep - 1 Error Reporting Requirements).

**ActDiagMenus - 4.2 Show Hidden Nodes**

In the case that nodes are hidden there shall be a way to recover them.

**ActDiagMenus - 4.3 Add Nodes to File**

File nodes need the ability to have new conversation templates, global policies, stray comments, and stray Lisp code blocks added to them. (Requirement Vis - 4.1 Create, Modify, and Remove)

**ActDiagMenus - 4.4 Delete Nodes from File**

All nodes contained by a file node shall be deletable from the file node. (Requirement Vis - 4.1 Create, Modify, and Remove)

**ActDiagMenus - 4.5 Reorder Nodes in File**

All nodes contained by a file node shall be able to specify their declaration order within the file. (Requirement Vis - 4.2 Change Position)

**InstDiagMenus - 5 Instance Diagram Menus**

For the instance diagram defined by design Diagrams - 1.3 Instance Diagram we need to support right click functionality that is specific to the top level conversation nodes and to the instance diagram’s background.

**InstDiagMenus - 5.1 Playing Animations**

Top level conversations need the ability to get a conversation to play through it’s animation (Requirement B.4)

**InstDiagMenus - 5.2 Animation Speeds**

In the interest of being user friendly the amount of time between animation steps shall have the ability to be increased and decreased.
EditNodeContents - 6 Editing Node Contents

All editable nodes (either top level or directly contained by a top level element) shall allow their contents to be modified. (Requirement Vis - 4.1 Create, Modify, and Remove)

Note that while any UML node could have modifiable text what this really means is that the data structure of editable nodes can be updated through these modifications.

EditNodeContents - 6.1 Speech Act Editor

In this diagram the only UML nodes excluded (i.e. have unmodifiable data structures) are those that are not occurring as a top level element or a child of a top level element. This applies to conversation type calls which will be represented by their corresponding types (Requirement Vis - 3.4 Conversation Template Representations). Conversation type calls cannot have parameters/children added to or removed from them. However, the values passed to a call’s parameters by the parenting top level conversation type can be modified.

EditNodeContents - 6.2 Action Editor

Recall that all conversation templates in the action diagram are conversation type calls so (as described by design EditNodeContents - 6.1 Speech Act Editor) they can only have the parameters passed to them modified.

EditNodeContents - 6.3 Instance Viewer

All nodes in this diagram represent data structure of already instantiated conversations. While a conversation that has not yet terminated (perhaps it is waiting for system resources) could have certain values modified during run time (like forcing a change in state) the intended purpose of this
diagram is (at present) to observe these conversations and not to modify them. This will be discussed further in Chapter §8.1.2 on the Implementation Drawbacks and §8.1.3 on Future Work.

ParseChanges - 7 Parsing Content Changes

All editable nodes shall be able to refresh their data structures whenever content changes are made. Note: This shall perform error checking. (Requirement ErrRep - 1 Error Reporting Requirements)

NodeVis - 8 Node Visibilities

Since part of the focus of this tool is to keep from bombarding the user with too much information nodes/relations shall have the ability to be removed from, and added to, the view in various ways.

NodeVis - 8.1 Collapsible/Expandable Nodes

Nodes that can have children shall be collapsible and expandable. Collapsing removes their children from view while themselves remaining visible. Expanding makes all of their immediate children visible meaning that a conversation call occurring within a conversation type will not also be expanded to see its children though it will also be expandable.

NodeVis - 8.2 Closable Nodes

Nodes (and all their children) shall be removable from the diagram. In case a node is edited but not saved a confirmation warning message shall be given to the user before close is actually performed. (ErrRep - 1 Error Reporting Requirements)

NodeVis - 8.3 Focusable Nodes

Any node shall have the ability to be focused on, minimizing any other nodes that are open.
NodeVis - 8.4 Filtering Content

In case the user does not want to see certain pieces of information such as particular kinds of relations or nodes the diagram shall have an interface available to set visibility flags for the various types of nodes and relations.

Anim - 9 Animations

The following parts of the requirements listed in §B.4 will need to be included within the design. Recall that animations are only used in the instance level diagram as these collect and animate the sequences of message exchanges occurring between agents for conversations that have actually been instantiated.

Anim - 9.1 Observer Design Pattern

The class that represents the agent node defined by design will register itself as the appropriate observer in order to log policy execution events and associate them as keyframes to the appropriate conversations.

(Requirements Anim - 1 Conversation Monitoring)

Anim - 9.2 Keyframes

A list of keyframes will be defined as a list of pairings between policies executed and the event that caused the execution. (Requirement Anim - 2 Conversation KeyFrames)

Anim - 9.3 Policies Executed

The contains relation pointing from the parenting conversation to the executed policy will be labelled by the event’s message content and the resulting next state.

Anim - 9.4 Conversation State

The animating conversation has a list of available states within its contents
as defined by design [ElemDefs - 2.1.2 Conversation Type Contents]

Every state that the conversation moves into during animation shall be wrapped in some kind of emphasis typography such as bolding or underlining.

ParseNCapture - 10 Parser And Comment Capturing Heuristic

Requirement [Vis - 1 Graphical Representation] states that we must be able to generate the graphical representations from Lisp code. This can be done through the use of a parser but we also need to fulfill requirements [Vis - 3.15 Stray Comment Representations], [Vis - 3.13 Documenting Comments Representations], [Vis - 3.14 Code Fragment Comment Representations], and [Save - 3.2 Correct Comment Placement] meaning that comments must be captured and stored appropriately. The actual parsing of Lisp code will simply need to perform bracket and colon pairing.

ParseNCapture - 10.1 Term Identification

Any sequence of characters that do not start with a ‘(’ or ‘)’ or ‘:’ and do not meet any of our special function types (see [Regex - 10.2 Bracket Pairing]) shall be considered a term.

ParseNCapture - 10.2 Bracket Pairing

Every occurrence of ‘(’ shall eventually be followed by a ‘)’

ParseNCapture - 10.2.1 Conversation Type Detection

(defun ... (conversation ...) ): indicates a conversation type.

ParseNCapture - 10.2.2 Conversation Type Call Detection

(type-name ....): indicates a conversation type being called provided a type with the name matching ‘type-name’ can be found.

ParseNCapture - 10.2.3 Global Policy Detection
(agent.put-policy (policy ...) ...): indicates a global policy.

ParseNCapture - 10.2.4 Local Policy Detection

(policy ...): indicates a policy object.

ParseNCapture - 10.2.5 Conditional Detection

(if ...): indicates a runtime structural change to a template when it occurs within the template definition’s list function or after the parameter list and before the conversation function in which it wraps globally accessible policies.

ParseNCapture - 10.2.6 Stray Lisp Code Detection

Anything else that does not fit any of the above and is not a comment is simply a stray Lisp code block.

ParseNCapture - 10.3 Colon Pairing

Every occurrence of ‘:’ shall be followed by a name and an immediate code section. (i.e. :precondition code, :ghost T, etc).

ParseNCapture - 10.4 Comment Capturing Heuristic

The following rules will determine how to recognize comments and how to determine if an individual comment is associated with a block of Lisp code. (Requirements Vis - 3.14 Code Fragment Comment Representations and Save - 3.2 Correct Comment Placement).

ParseNCapture - 10.4.1 Single Line Comments

A semicolon ‘;’ indicates that everything up to the end of the line is part of a comment.

ParseNCapture - 10.4.2 Multiline Comments

A “#|” indicates that everything up to the next “|#” is part of a comment.
ParseNCapture - 10.4.3 Trailing Comment Detection

If the comment appears at the end of a line of code then it is a trailing comment for that code.

ParseNCapture - 10.4.4 Stray Comment Detection

If the comment is on its own line and is not indented, it is a stray comment that does not refer to anything.

ParseNCapture - 10.4.5 Preceding Comment Detection

If the comment is on its own line and is indented then it refers to the next line of code if one exists.

PrettyPrint - 11 Pretty Printing

Saving (Requirement Save - 2 Visualization to Lisp) can be done fairly trivially as we need only provide a default representation for the major pieces of code presented in Chapter 2.4 figures 2.9 to 2.15.

A recursive construction of those code pieces can easily create parseable Lisp code however we additionally wish to provide some heuristic to allow the created Lisp code to be readable by anyone who wants to directly edit the Lisp representation (Save - 3.1 Pretty Printing Lisp Code). On top of this we do not want to unwittingly confuse commented code with uncommented code nor allow for comment migration in case the comments are in regards to a specific section of code (Save - 3.2 Correct Comment Placement). This gives us our final list of design choices:

PrettyPrint - 11.1 Space Separated Fields

All fields shall be separated by a single space with the exception of opening and closing brackets where the first and last field in a bracket pairing does not require a space.
PrettyPrint - 11.2 Maximum Line Length

If a line exceeds a certain length in characters, insert a newline, wherever possible, and indent appropriately.

PrettyPrint - 11.3 Print Using Code Patterns

Code that falls into one of the categories presented in Chapter 2.4 shall closely follow those patterns.

PrettyPrint - 11.4 Owned Comment Placement

Comments that are associated with a given piece of code shall be printed before the code.

PrettyPrint - 11.5 Single Line Comment Newlines

Single line comments shall always be followed by a newline.
Appendix D

Project Implementation Points

DiagsImp - 1 Different Diagrams

As mentioned in §5.2 UMLet allows for multiple diagrams to remain open so satisfying the design objectives in **Diagrams - 1 Different Diagrams** is easy enough. The diagrams created are stored in the user’s temporary directory and are told to delete on exit. If the program is, however, forced to quit (whether from the command line or through a system utility) than the diagrams will remain until the operating system removes them (usually on exit). To make the diagrams distinguishable they are named differently to reflect their function and, if running in the context of an Agent, the diagram names also contain the name of the Agent.

DiagsImp - 1 Speech Act Editor

A diagram has been dedicated to the speech act level editor. (Design **Diagrams - 1.1 Speech Act Diagram**) The diagram is named either ‘speechAct-StandAlone’ or ‘speechAct-agentName’ (see §5.5.1) depending on the context the project is running in.

DiagsImp - 1 Action Level Editor

A diagram has been dedicated to the action level editor. (Design **Diagrams - 1.2 Action Diagram**) The diagram is named either ‘actionLvl-StandAlone’ or ‘actionLvl-agentName’ depending on the context the project is running in.

DiagsImp - 1 Instance Level Editor

A diagram has been dedicated to the instance level viewer. (Design
Diagrams - 1.3 Instance Diagram
The diagram is named either ‘instanceView-StandAlone’ or ‘instanceView-agentName’ depending on the context the project is running in.

ElemImps - 2 Element Implementations
What follows is how the various elements in design are implemented as follows:

ElemImps - 2.1 Conversation Types Implementation
Conversation types are implemented as the one shown in Figure 5.8.

ElemImps - 2.1.1 Labelling Implementation
Template nodes are labeled by the name of the defun that defines them. (Design)

ElemImps - 2.1.2 Contents Implementation
The content of this kind of node is the Lisp code representing all the variables, their values, and the states that the conversation can be in. (Design)

ElemImps - 2.1.3 Colour Implementation
Conversation Type nodes are coloured light blue. (Design)

ElemImps - 2.1.4 Diagram Use Implementation
Conversation Types are meant to be editable so they are only used in the speech act level editor. (Design)

ElemImps - 2.2 Policies Implementation
Policy nodes contained by a conversation are implemented as the one shown in Figure 5.9.
Global policy nodes are implemented as the one shown in Figure 5.10.

**ElemImps - 2.2.1 Policy Labelling Implementation**

Policy nodes are labeled by the name in their Lisp representation.  
(Design [ElemDefs - 2.2.1 Policy Labelling](#))

**ElemImps - 2.2.2 Policy Contents Implementation**

The content of this kind of node is the Lisp code representing the antecedent, precondition, postcondition, and consequent(s). (Design [ElemDefs - 2.2.2 Policy Contents](#))

**ElemImps - 2.2.3 Policy Colour Implementation**

Policy nodes contained by a conversation are coloured yellow however for added clarity, global policies (and thus ones that cause conversation instantiation) are coloured pale green. (Design [ElemDefs - 2.2.3 Policy Colour](#))

**ElemImps - 2.3 Instance Conversations Implementation**

Instantiated conversations are implemented as the one shown in Figure 5.11. These do not have to relate their non-editability status since the concept of animating past conversations already implies that no changes can be made and it is better to keep the conversation as well as its child nodes readable.

**ElemImps - 2.3.1 Form Implementation**

No type matching is necessary as we have the runtime object’s Lisp code and child object lists so these conversations always take the form of conversations. (Design [ElemDefs - 2.3.1 Instance Conversation Form](#))

**ElemImps - 2.3.2 Labelling Implementation**

The label of this kind of node is taken from the name of the run-
time object. (Design \textit{ElemDefs - 2.3.2 Instance Conversation Labelling})

\textbf{ElemImps - 2.3.3 Contents Implementation}

The content of this kind of node is essentially the same as in \textit{ElemImps - 2.1.2 Contents Implementation} and \textit{ElemImps - 2.5.3 Contents Implementation} except that the code is taken from the runtime object whose parameters have already been evaluated. (Design \textit{ElemDefs - 2.3.3 Instance Conversation Contents})

\textbf{ElemImps - 2.3.4 Colours Implementation}

Conversation call nodes are coloured light blue as in \textit{ElemImps - 2.1.3 Colour Implementation} and \textit{ElemImps - 2.5.4 Colours Implementation} (Design \textit{ElemDefs - 2.3.4 Instance Conversation Colours})

\textbf{ElemImps - 2.4 Instance Diagram Nodes Implementation}

Nodes in the instance level use the conventional colours despite their non-editability as shown with the instance conversations in Figure 5.11. (Design \textit{ElemDefs - 2.20 Instance diagram nodes})

\textbf{ElemImps - 2.5 Templates And Sub Conversations Implementation}

Conversation calls/sub conversations are implemented as shown in Figure 5.12 where the only difference from a conversation type is that the text is greyed out to indicate that they are not editable aside from the parameters passed to them.

\textbf{ElemImps - 2.5.1 Conversation Call Form Implementation}

If no matching type can be found within the currently active type file (open in the palette panel and editable) nor any of the imported
type files then a Lisp code node is generated. (Design ElemDefs -
2.4.1 Template and Sub Conversation Form)

ElemImps - 2.5.2 Labelling Implementation

The label for this kind of node is the same as in ElemImps - 2.1.1
Labelling Implementation.

ElemImps - 2.5.3 Contents Implementation

The content of this kind of node is the same as in ElemImps - 2.1.2
Contents Implementation.

ElemImps - 2.5.4 Colours Implementation

Conversation call nodes are coloured light blue as in ElemImps -
2.1.3 Colour Implementation.

ElemImps - 2.5.5 Speech Act Diagram Implementation

These nodes are used by types in the speech act diagram since types
can call other types with parameters. (Design ElemDefs - 2.4.5
Sub Conversations in Speech Act Diagram)

ElemImps - 2.5.6 Action Diagram Implementation

These nodes are also used by action diagram file nodes as templates
that are executed by an Agent at startup to create the concrete
conversations the Agent can engage in. (Design ElemDefs - 2.4.6
Templates in Action Diagram)

ElemImps - 2.5.7 Editability Implementation

To indicate non-editable status we use paler text as discussed in
ElemDefs - 2.19 Node Edit-ability. (Design ElemDefs - 2.4.7
Template and Sub Conversation Editability)

ElemImps - 2.6 Node Edit-ability Implementation

Non editable nodes (ones that are or are part of a conversation call) indi-
cate this by having grey text as shown with the offer-client and ask-client conversation calls in Figure 5.12. Note that all of the children of the conversation calls would also use grey text as they cannot be modified.

(Design **ElemDefs - 2.19 Node Edit-ability**)

**ElemImps - 2.7 Agent Nodes Implementation**

Agent nodes are implemented as the ones shown in Figure 5.13.

**ElemImps - 2.7.1 Labelling Implementation**

The label of this kind of node is the name of the Agent. (Design **ElemDefs - 2.5.1 Agent Node Labelling**)

**ElemImps - 2.7.2 Contents Implementation**

The content of this kind of node is empty. (Design **ElemDefs - 2.5.2 Agent Node Contents**)

**ElemImps - 2.7.3 Colour Implementation**

Agent nodes are coloured white. (Design **ElemDefs - 2.5.3 Agent Node Colours**)

**ElemImps - 2.7.4 Context Implementation**

This node is only present when the tool is running in an active agent context. (Design **ElemDefs - 2.5.4 Agent Node Context**)

**ElemImps - 2.7.5 Speech Act Diagram Implementation**

The speech act diagram does not make use of this node. (Design **ElemDefs - 2.5.5 Agent Node in Speech Act Diagram**)

**ElemImps - 2.7.6 Action Diagram Implementation**

The action level diagram makes use of this node and uses containment relations to indicate the conversation calls, Lisp code blocks, and comments that are contained within it. (Design **ElemDefs - 2.5.6 Agent Node in Action Diagram**)

185
ElemImps - 2.7.7 Instance Diagram Implementation

The instance level diagram makes use of this node and uses containment relations to indicate the conversations that are currently contained within its history. (Design ElemDefs - 2.5.7 Agent Node in Instance Diagram)

ElemImps - 2.8 File Membership Implementation

Design ElemDefs - 2.6 File Membership defines how nodes that are members of a file indicate such along with which file.

ElemImps - 2.8.1 Instance Diagram Files Implementation

No file information is present in the instance file. (Design ElemDefs - 2.6.1 Instance Diagram Files)

ElemImps - 2.8.2 Action Diagram Files Implementation

Since the action level editor requires file nodes they are implemented as the one shown in Figure 5.14. (Design ElemDefs - 2.6.2 Action Diagram Files)

ElemImps - 2.8.2.1 File Node Labelling Implementation

File nodes are labeled by their filename. (Design ElemDefs - 2.6.2.1 File Node Labelling)

ElemImps - 2.8.2.2 File Node Contents Implementation

File nodes have no contents. (Design ElemDefs - 2.6.2.2 File Node Contents)

ElemImps - 2.8.2.3 File Node Colours Implementation

File nodes are coloured black. (Design ElemDefs - 2.6.2.3 File Node Colours)

ElemImps - 2.8.2.4 File Node Members Implementation

Nodes contained by a file node are connected to it via a contain-
ment relation. (see Figure 5.15). (Design ElemDefs - 2.6.2.4

File Node Members)

ElemImps - 2.8.3 Speech Act Diagram Files Implementation

Since the file that conversation types are contained in cannot appear in the speech act diagram itself (as a file node) its contents are displayed in the palette panel. To save on space the elements in the palette panel do not have any content but are labeled by either a type name (if they are a conversation type) or a numbering for which (in declaration order) stray comments/Lisp code block they represent as shown in Figure 5.7. (Design ElemDefs - 2.6.3 Speech Act Diagram Files)

ElemImps - 2.8.4 Speech Act Diagram Imports Implementation

Other conversation type files can be imported per designs TempDiagMenus - 3.3 Import File and TempDiagMenus - 3.4 Change File to be used by other types in the speech act diagram (as sub conversation calls) and by templates in the action diagram (as calls that create concrete conversations). Since mouse over text can be added to the palette panel the text is set to show the name of the active (editable) type file and the list of imports.

ElemImps - 2.9 Parent Node Membership Implementation

Nodes that are children of (or contained by) a parent node indicate parentage using a containment relation as shown in Figure 5.15. (Design ElemDefs - 2.7 Parent Node Membership)

However, sometimes a policy that instantiates a conversation may be present under a conditional node and thus not initially visible to the user
due to our use of minimizing and maximizing (design Collapsible/Expandable Nodes in the custom elements used by parent nodes (Figure 5.6). In such a case the containment relation of the conditional should inform the user of this. Therefore all containment relations that point towards global policies should use a special containment relation as shown in Figure 5.16.

ElemImps - 2.10 Position of Child Elements Implementation

Child nodes appear below parent nodes as shown in Figure 5.20 (Design ElemDefs - 2.8 Position of Child Elements).

ElemImps - 2.11 Order of Child Elements Implementation

Child nodes indicate declaration order from left to right where leftmost occurs first within the parent’s definition and rightmost is last. (Design ElemDefs - 2.9 Order of Child Elements)

However, since the palette panel shown in Figure 5.7 must have the same indication capabilities the order is instead shown from top to bottom where topmost occurs first within the conversation types file and bottommost is last.

ElemImps - 2.12 Conditionals Implementation

Conditionals and their children are implemented as the one shown in Figure 5.17.

ElemImps - 2.12.1 Conditional Labelling Implementation

The label of this kind of node is the code representing the condition. (Design ElemDefs - 2.10.1 Conditional Labelling)

ElemImps - 2.12.2 Conditional Contents Implementation

The content of this kind of node is empty. (Design ElemDefs - 2.10.2 Conditional Contents)
ElemImps - 2.12.3 Conditional Colours Implementation

Conditional nodes are coloured pink. (Design ElemDefs - 2.10.3 Conditional Colours)

ElemImps - 2.12.4 Conditional True/False Relations Implementation

The label on the contains relation that indicates the True and False cases uses the text True and Nil (False). In addition, the arrows of the relations are coloured green and red for True and False respectively. (Design ElemDefs - 2.10.4 Conditional True/False Relations)

ElemImps - 2.13 State Transitions Implementation

Possible execution paths between policies is shown using a transition relation as shown in Figure 5.18. (Design ElemDefs - 2.11 State Transitions and Possible Execution Paths)

ElemImps - 2.14 Equivalent States and Variables Implementation

To indicate that two states/variables in two different conversations are considered the same is shown using an equivalence relation as shown in Figure 5.19. (Design ElemDefs - 2.12 Equivalent States and Variables)

ElemImps - 2.15 Stray Comments Implementation

Comment nodes are implemented as the one shown in Figure 5.20.

ElemImps - 2.15.1 Stray Comment Labelling Implementation

Comment nodes are labeled by the implementing class name ‘Uml-CommentNode’. (Design ElemDefs - 2.16.1 Stray Comment Labelling)
ElemImps - 2.15.2 Stray Comment Contents Implementation

The content of this kind of node is the collected comments. (Design ElemDefs - 2.16.2 Stray Comment Contents)

ElemImps - 2.15.3 Stray Comment Colour Implementation

Comment nodes are coloured green. (Design ElemDefs - 2.16.3 Stray Comment Colour)

ElemImps - 2.16 Stray Lisp Code Implementation

Lisp code nodes are implemented as the one shown in Figure 5.21.

ElemImps - 2.16.1 Stray Lisp Code Labelling Implementation

Code nodes are labeled by the implementing class name ‘UmlCodeNode’. (Design ElemDefs - 2.17.1 Stray Lisp Code Labelling)

ElemImps - 2.16.2 Stray Lisp Code Contents Implementation

The content of this kind of node is the captured Lisp code. (Design ElemDefs - 2.17.2 Stay Lisp Code Contents)

ElemImps - 2.16.3 Stray Lisp Code Colours Implementation

Code nodes are coloured blue. (Design ElemDefs - 2.17.3 Stray Lisp Code Colour)

ElemImps - 2.17 Documentation Information Implementation

The documentation associated with a conversation type or policy is shown as mouse over text when the user’s cursor is above the node’s label as shown in Figure 5.22. (Design ElemDefs - 2.13 Documentation Information)

ElemImps - 2.18 Documenting Comments Implementation

The comments that seem to document a conversation type, conversation call, or policy are added to the mouse over text in ElemImps - 2.17.
Documentation Information Implementation and thus would also be shown in Figure 5.22 (Design ElemDefs - 2.14 Documenting Comments).

ElemImps - 2.19 Smart Comments Implementation

Similar to ElemImps - 2.18 Documenting Comments Implementation when the cursor is above a variable or section within a node’s content the comments corresponding to that content line appear as mouse over text as shown in Figure 5.23 (Design ElemDefs - 2.15 Smart Comments).

ElemImps - 2.20 Minimized Complexity

Nodes have the ability to be removed from view when they are not of concern to the user. This is implemented later on in NodeVisImps - 8 Node Visibilities and in each diagram behaves as follows:

ElemImps - 2.20.1 Remove Node from Speech Act Diagram

For the speech act level diagram, if a node is removed from view than it is removed from the diagram’s list of open (and thus editable) nodes. (Design ElemDefs - 2.18.1 Remove Node from Speech Act Diagram)

ElemImps - 2.20.2 Remove Node from Action Diagram

For the action level diagram, if a node is removed from view it is only hidden from view and is thus recoverable through menu options as discussed in implementation ActMenuImps - 4.2 Show Hidden Nodes Implementation (Design ElemDefs - 2.18.2 Remove Node from Action Diagram)

ElemImps - 2.20.3 Remove Node from Instance Diagram

For the instance level diagram, if an executed conversation is removed from view then it is removed from the diagram’s list of past conver-
TempMenuImps - 3 Speech Act Diagram Menus Implementation

For the speech act diagram implemented by DiagsImp - 1 Speech Act Editor right click functionality is supported specific to the various nodes and the diagram’s background through a popup menu like the ones shown in Figure 5.24 populated by options supporting the following:

TempMenuImps - 3.1 Open/Add Top Level Node Implementation

The background popup can open/add types, stray comments, and stray Lisp code blocks. (Design TempDiagMenus - 3.1 Open/Add Top Level Node)

TempMenuImps - 3.2 Save Current File Implementation

The background popup can save the current collection of conversation types to a specific file. (Design TempDiagMenus - 3.2 Save Current File)

TempMenuImps - 3.3 Import File Implementation

The background popup can open other conversation type files as imports. (Design TempDiagMenus - 3.3 Import File)

TempMenuImps - 3.4 Change File Implementation

The current active conversation type file can be changed. (Design TempDiagMenus - 3.4 Change File)

TempMenuImps - 3.5 Reorder Nodes in File Implementation

The conversation type file collection allows the elements it contains to be reordered by dragging them above and below the other elements in
TempMenuImps - 3.5 Reorder Nodes in File Implementation

The palette panel. (Design TempDiagMenus - 3.5 Reorder Nodes in File)

TempMenuImps - 3.6 Delete Nodes from File Implementation

The conversation type file collection allows the elements it contains to be deleted. (Design TempDiagMenus - 3.6 Delete Nodes from File)

TempMenuImps - 3.7 Add/Delete Child Nodes Implementation

Top level conversations and conditionals can have policies, conditionals, sub conversations, stray comments, and stray Lisp code blocks added to them and removed from them. (Design TempDiagMenus - 3.7 Add/Delete Child Nodes)

TempMenuImps - 3.8 Reorder Child Nodes Implementation

All child nodes of a conversation type can change their declaration order within the data structure. (Design TempDiagMenus - 3.8 Reorder Child Nodes)

ActMenuImps - 4 Action Diagram Menus Implementation

For the action diagram implemented by DiagsImp - 1 Action Level Editor we support right click functionality that is specific to the various nodes and the action diagram’s background through a popup menu like the ones shown in Figure 5.25 populated by options supporting the following:

ActMenuImps - 4.1 Open/Save File Implementation

Background should have the ability to open existing conversation template files, create new ones, and save them to a specific file. Creating/saving performs error checking in case of file overwrite or if a file by that name is already open. (Design ActDiagMenus - 4.1 Open/Save File)

ActMenuImps - 4.2 Show Hidden Nodes Implementation
In the case that nodes are hidden an option is available to recover them.

(Desing [ActDiagMenus - 4.2 Show Hidden Nodes])

**ActMenuImps - 4.3 Add Nodes to File Implementation**

File nodes can have new conversation templates, global policies, stray comments, and stray Lisp code blocks added to them. (Design [ActDiagMenus - 4.3 Add Nodes to File])

**ActMenuImps - 4.4 Change Containing File Implementation**

Note that the drag and drop functionality offered by the palette panel does not assign a file parent as one may not necessarily be available and even if there were some there is no way to know which to add to. Thus these templates are added to the background and must instead be assigned to a file. In case the user assigns them to the wrong file nodes can be moved between files.

**ActMenuImps - 4.5 Delete Nodes from File Implementation**

All nodes contained by a file node can be deleted from the file node. (Design [ActDiagMenus - 4.4 Delete Nodes from File])

**ActMenuImps - 4.6 Reorder Nodes in File Implementation**

All nodes contained by a file node can specify their declaration order within the file. (Design [ActDiagMenus - 4.5 Reorder Nodes in File])

**InstMenuImps - 5 Instance Diagram Menus Implementation**

For the instance diagram implemented by [DiagsImp - 1 Action Level Editor] we support right click functionality that is specific to the various nodes and the instance diagram’s background through a popup menu like the ones shown in Figure 5.27 populated by options supporting the following:

**InstMenuImps - 5.1 Playing Animations Implementation**
Top level conversations can make a conversation play through it’s animation (Requirement (Design [InstDiagMenus - 5.1 Playing Animations]

**InstMenuImps - 5.2 Animation Speeds Implementation**

The amount of time between animation steps can be increased and decreased which is capped by a maximum of 4 seconds and a minimum of 0.5 seconds starting at 1 second. (Design [InstDiagMenus - 5.2 Animation Speeds]

**EditNodeImps - 6 Editing Node Contents Implementation**

All editable nodes (either top level or directly contained by a top level element) allow their contents to be modified meaning that the data structure of the node can be updated through the modifications.

**EditNodeImps - 6.1 Speech Act Editor**

Exclude nodes that are contained by a conversation call. Conversation calls themselves can only have the parameters passed to them modified. (Design [EditNodeContents - 6.1 Speech Act Editor]

**EditNodeImps - 6.2 Action Editor**

Exclude nodes that are contained by a conversation call noting that all conversations in the action diagram are conversation calls. Conversation calls themselves can only have the parameters passed to them modified. (Design [EditNodeContents - 6.2 Action Editor]

**EditNodeImps - 6.3 Instance Viewer**

Excludes all nodes in this diagram as the data structure of past instantiated conversations would have no effect anyway since the conversation is over and is therefore unmodifiable. (Design [EditNodeContents - 6.3]
ParseChangesImpl - 7 Parsing Content Changes Implementation

All editable nodes can refresh their data structures whenever content changes are made via a request made through the action and speech act diagrams’ background popup menu. Note: This performs error checking and thus user warnings. (Design ParseChanges - 7 Parsing Content Changes)

NodeVisImps - 8 Node Visibilities

To keep from bombarding the user with too much information nodes/relations have the ability to be removed from, and added to, the view in the following ways:

NodeVisImps - 8.1 Collapsible/Expandable Nodes

 Nodes that have children are collapsible and expandable as shown by the custom element in Figure 5.6 where ‘-’ means minimize and ‘+’ means maximize. (Design NodeVis - 8.1 Collapsible/Expandable Nodes)

NodeVisImps - 8.2 Closable Nodes

Nodes (and all their children) are removable from the diagram as shown by the custom element in Figure 5.6 where ‘X’ means to close. This function can also be performed through popup menu operations on the node itself. In case a node is edited but not saved a confirmation warning message is shown to the user before close is actually performed. (Design NodeVis - 8.2 Closable Nodes)

NodeVisImps - 8.3 Focusable Nodes

Any node can be focused on, minimizing all other nodes that are open. (Design NodeVis - 8.3 Focusable Nodes)

NodeVisImps - 8.4 Filtering Content
In case the user does not want to see certain pieces of information such as particular kinds of relations or nodes the diagram’s background popup menu has options to set visibility flags for the various nodes and relations. These settings are reflected across all three diagrams. (Design NodeVis - 8.4 Filtering Content)

AnimImps - 9 Animations

Animation is implemented as shown in Figure 5.29 which achieves the following points from app3:design:

AnimImps - 9.1 Observer Design Pattern

The class that represents the Agent node implemented by registers itself as an observer of policy execution events and associates them as keyframes to the appropriate conversation. (Design Anim - 9.1 Observer Design Pattern)

AnimImps - 9.2 Keyframes

A list of keyframes is defined as a list of pairings between policies executed and the event that caused the execution. This list is owned by the instance conversation object. (Design Anim - 9.2 Keyframes)

AnimImps - 9.3 Policies Executed

The contains relation that points to the executed policy has the event’s message content and resulting next state placed as a label on it. (Design Anim - 9.3 Policies Executed)

AnimImps - 9.4 Conversation State

Every state that the conversation moves into during animation becomes bolded. (Design Anim - 9.4 Conversation State) Note that if a child conversation is not used then it will not receive bolding on its state as
states are transformed from child to parent.

ParseNCapImpl - 10 Parser And Comment Capturing Heuristic

The parser achieves all points in the design as follows:

ParseNCapImpl - 10.1 Term Identification

Any sequence of characters that do not start with a ‘(‘ or ‘)’ or ‘:’ and do not match any of our special function names (see ParseNCapImpl - 10.2 Bracket Pairing following) are considered a term. (Design ParseNCapture - 10.1 Term Identification)

ParseNCapImpl - 10.2 Bracket Pairing

Every occurrence of ‘(‘ is eventually followed by a ‘)’ otherwise parsing stops and an error is displayed to the user. The file therefore does not get loaded. (Design ParseNCapture - 10.2 Bracket Pairing)

ParseNCapImpl - 10.2.1 Conversation Type Detection

(defun ... (conversation ...) ) is implemented as a DefunNode (Design ParseNCapture - 10.2.2 Conversation Type Call Detection)

ParseNCapImpl - 10.2.2 Global Policy Detection

(agent.put-policy (policy ... ) ...) is implemented as a FunctionNode (Design ParseNCapture - 10.2.3 Global Policy Detection)

ParseNCapImpl - 10.2.3 Local Policy Detection

(policy ...) is implemented as a FunctionCallNode (Design ParseNCapture - 10.2.4 Local Policy Detection)

ParseNCapImpl - 10.2.4 Conditional Detection

(if ... ): is always implemented as a ConditionalNode however it only indicates a runtime structural change to a conversation type when it
occurs within the type’s function definition. This is caught during conversation type construction. (Design ParseNCapture - 10.2.5 Conditional Detection)

ParseNCapImpl - 10.2.5 Stray Lisp Code Detection

Anything else that does not fit any of the above and is not a comment is implemented as a GeneralListNode. (Design ParseNCapture - 10.2.6 Stray Lisp Code Detection)

ParseNCapImpl - 10.3 Colon Pairing

Every occurrence of ‘:’ must be followed by a name and an immediate code section (i.e. :precondition code, :ghost T, etc) otherwise parsing stops and an error is displayed to the user. The file therefore does not get loaded. (Design ParseNCapture - 10.3 Colon Pairing)

ParseNCapImpl - 10.4 Comment Capturing Heuristic

The rules listed in Design ParseNCapture - 10.4 Comment Capturing Heuristic have all been implemented. It may be helpful to refer to Figure 5.30 in order to easily grasp the following:

ParseNCapImpl - 10.4.1 Single Line Comments

A semicolon ‘;’ indicates that everything up to the end of the line is part of a comment. These are implemented as a CommentNode and have a decoration indicating that they are a single line comment. (Design ParseNCapture - 10.4.1 Single Line Comments)

ParseNCapImpl - 10.4.2 Multiline Comments

A “#|” indicates that everything up to the next “|#” is part of a comment. These are also implemented as a CommentNode but have a decoration indicating that they are a multi line comment though they do not necessarily span multiple lines. (Design ParseNCapture -
10.4.2 Multiline Comments

ParseNCapImpl - 10.4.3 Trailing Comment Detection

If the comment appears at the end of a line of code then it is a trailing comment for that code. This is achieved by allowing any code nodes described in ParseNCapImpl - 10.1 Term Identification to own a collection of such comments. (Design ParseNCapture - 10.4.3 Trailing Comment Detection)

ParseNCapImpl - 10.4.4 Stray Comment Detection

If the comment is on its own line and is not tabbed, it is a stray comment that does not refer to anything. These comments are therefore implemented as a UmlCommentNode as described in ElemImps - 2.15 Stray Comments Implementation. (Design ParseNCapture - 10.4.4 Stray Comment Detection)

ParseNCapImpl - 10.4.5 Preceding Comment Detection

If the comment is on its own line and is tabbed then it refers to the next line of code if one exists. These comments therefore either get added to the next code node described in ParseNCapImpl - 10.1 Term Identification or else they are defined as stray comments and implemented as a UmlCommentNode as in ElemImps - 2.15 Stray Comments Implementation. (Design ParseNCapture - 10.4.5 Preceding Comment Detection)

PrettyPrintImps - 11 Pretty Printing Implementation

The pretty printer is a notion used for saving Lisp code nodes to a file in a readable fashion. This is achieved through the use of the toString() method which creates a string representation of the code in order for it to be written to a file. Therefore each Lisp code node defines this method and recursively
assembles all child Lisp code nodes by calling their `toString()` methods while obeying the rules from Design PrettyPrint - 11 Pretty Printing. It may be helpful to refer to Figure 5.31 in order to easily grasp these rules:

PrettyPrintImps - 11.1 Space Separated Fields

All fields within a bracket pairing are separated by a single space. (Design PrettyPrint - 11.1 Space Separated Fields)

PrettyPrintImps - 11.2 Maximum Line Length

When a line exceeds a length of 100 characters a newline is inserted after the current field is finished printing. The next line is then tabbed appropriately. (Design PrettyPrint - 11.2 Maximum Line Length)

PrettyPrintImps - 11.3 Print Using Code Patterns

Code that falls into one of the categories presented in Chapter 2 § 2.4 follows those patterns for its layout (indentation, placement of newlines, etc.). (Design PrettyPrint - 11.3 Print Using Code Patterns) Also note that since the end representation is recursively constructed all child nodes get tabbed as in Figure 5.31 meaning that next time the file is parsed the stand alone comment ‘;comment’ will become associated with the policy definition.

PrettyPrintImps - 11.4 Code Fragment Comment Placement

Comments that are associated with a given piece of code are printed before the code regardless of whether they were trailing or not. They maintain association with the code by being tabbed to the code’s matching indentation. (Design PrettyPrint - 11.4 Owned Comment Placement)

PrettyPrintImps - 11.5 Newline after Single Line Comments

Single line comments are always followed by a newline (Design PrettyPrint - 11.5 Newline after Single Line Comments)
Appendix E

User Acceptance Tests

Table E.1: Palette File Name

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Palette File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>1</td>
</tr>
<tr>
<td>Description</td>
<td>The current loaded template file name and path is visible as mouse over text in the palette window.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running</td>
</tr>
<tr>
<td>Steps</td>
<td>1) User moves mouse into palette area</td>
</tr>
<tr>
<td>Expected Results</td>
<td>The path to and the name of the current loaded template is visible as mouse over text.</td>
</tr>
</tbody>
</table>

Table E.2: Switch Palette File

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Switch Palette File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>2</td>
</tr>
<tr>
<td>Description</td>
<td>The current loaded template file can be changed to another of the user’s choosing.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks diagram background  
2) User selects “Open Working Templates File”  
3) User navigates to the file using file chooser popup and selects “OK” |
| Expected Results | The Palette is reloaded with the comments, code blocks, and templates the new file names. The mouse over text of the palette area is also reset to reflect the file opened. |
## Table E.3: Create New Palette File

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Create New Palette File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>3</td>
</tr>
<tr>
<td>Description</td>
<td>The current loaded template file can be changed to a blank one.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks diagram background  
  2) User selects “Create Working Templates File”  
  3) User chooses name and location for the new file using and selects “OK” |
| Expected Results | The Palette is emptied of all contents (like the file). The mouse over text of the palette area is also reset to reflect the new file. |

## Table E.4: Save Palette File

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Save Palette File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>4</td>
</tr>
<tr>
<td>Description</td>
<td>The current loaded template file can be saved to a file of the user’s choosing.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks diagram background  
  2) User selects “Save Working Templates File”  
  3) User chooses name and location for the new file and selects “OK” |
| Expected Results | The Lisp Representation of all nodes in the Palette area are saved to the file path/name combination given. |

## Table E.5: Import Additional Files

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Import Additional Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>5</td>
</tr>
<tr>
<td>Description</td>
<td>The user can import other files for reference and their file names and paths are all visible as mouse over text in the palette window.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the speech act diagram’s background.  
  2) User selects “Import Template File”  
  3) User chooses file using the file chooser popup and selects “OK” |
| Expected Results | The file is imported into the background and the file name and path combination is added to the mouse over text in the palette window. |
### Table E.6: Reorganize elements in Palette

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Reorganize elements in Palette</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>6</td>
</tr>
<tr>
<td>Description</td>
<td>The nodes declared in the current template file can have their declaration order changed.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A non empty palette file is present with at least two nodes.</td>
</tr>
<tr>
<td>Steps</td>
<td>1) The first node is dragged below the second node in the palette.</td>
</tr>
<tr>
<td>Expected Results</td>
<td>The list of nodes is reorganized so that the first node is now in the second position of the palette.</td>
</tr>
</tbody>
</table>

### Table E.7: Hide Palette Comments

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Hide Palette Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>7</td>
</tr>
<tr>
<td>Description</td>
<td>Comment nodes in the palette can be hidden from view.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running.</td>
</tr>
</tbody>
</table>
| Steps          | 1) User right clicks the palette panel’s background  
                 2) User selects “Show/Hide Comments” |
| Expected Results | Comment nodes are hidden from view and empty spaces are collapsed. |

### Table E.8: Reorganize elements in Palette with hidden comments

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Reorganize elements in Palette with hidden comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>8</td>
</tr>
<tr>
<td>Description</td>
<td>Tests reorganizing current template file while comments are not visible.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A non empty palette file is present with one comment and two non comment nodes. The comment node is first. The comment node is not visible.</td>
</tr>
</tbody>
</table>
| Steps                 | 1) The second visible node is brought above the first  
                          2) User right clicks the palette panel’s background  
                          3) User selects “Show/Hide Comments” to reveal the comment node |
| Expected Results       | The comment node is still the first node. The second and third nodes have traded places. |
Table E.9: Drag Comment from Palette to Speech Act diagram

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Drag Comment from Palette to Speech Act diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>9</td>
</tr>
<tr>
<td>Description</td>
<td>Verify that comments dragged from the palette to the speech diagram are the correct ones.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. The palette contains at least one comment node. The current diagram is the speech act diagram.</td>
</tr>
<tr>
<td>Steps</td>
<td>1) User drags a comment node from the palette to the diagram.</td>
</tr>
<tr>
<td>Expected Results</td>
<td>A UmlCommentNode is added to the diagram containing the comments that the palette node was representing.</td>
</tr>
</tbody>
</table>

Table E.10: Drag Comment from Palette to non speech act diagram

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Drag Comment from Palette to non speech act diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>10</td>
</tr>
<tr>
<td>Description</td>
<td>Verify that comments dragged from the palette to a non speech act diagram do nothing.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. The palette contains at least one comment node. The current diagram is not the speech act diagram.</td>
</tr>
<tr>
<td>Steps</td>
<td>1) User drags a comment node from the palette to the diagram.</td>
</tr>
<tr>
<td>Expected Results</td>
<td>No change to the diagram occurs.</td>
</tr>
</tbody>
</table>
### Table E.11: Remove Comment from diagram

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Remove Comment from diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>11</td>
</tr>
<tr>
<td>Description</td>
<td>Removing a comment from the diagram is possible.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A comment node from the palette is present in the speech act diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the UmlCommentNode.  
2) User selects “Remove From Diagram” |
| Expected Results | The UmlCommentNode is no longer present in the diagram. |

### Table E.12: Modify Comment Contents and Save to Palette

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Modify Comment Contents and Save to Palette</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>12</td>
</tr>
<tr>
<td>Description</td>
<td>Modifying and saving changes to a comment node are retained by the system.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A comment node from the palette is present in the speech act diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User modifies the contents of the comment node.  
2) User right clicks the comment node.  
3) User selects “Save Changes to Template”  
4) User right clicks the comment node.  
5) User selects “Remove From Diagram”  
6) User again adds the corresponding comment from the palette. |
| Expected Results | The UmlCommentNode added to the diagram contains all the changes made to the original. |
Table E.13: Add New Comment to Palette

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Add New Comment to Palette</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>13</td>
</tr>
<tr>
<td>Description</td>
<td>Tests that making a new comment automatically adds it to the palette’s node list.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running.</td>
</tr>
<tr>
<td>Steps</td>
<td>1) User right clicks the speech act diagram’s background. 2) User selects “Create Comment Block”</td>
</tr>
<tr>
<td>Expected Results</td>
<td>A default UmlCommentNode has been added to the diagram and a corresponding node is present in the palette.</td>
</tr>
</tbody>
</table>

Table E.14: Delete New Comment from Palette

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Delete New Comment from Palette</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>14</td>
</tr>
<tr>
<td>Description</td>
<td>Tests that making a new comment automatically adds it to the palette’s node list.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A new comment node has recently been added to the palette and is currently on the diagram.</td>
</tr>
<tr>
<td>Steps</td>
<td>1) User right clicks the comment node. 2) User selects “Delete From Templates”</td>
</tr>
<tr>
<td>Expected Results</td>
<td>The UmlCommentNode has been removed from the diagram and its corresponding node is no longer present in the palette area.</td>
</tr>
</tbody>
</table>

Table E.15: Drag Lisp Conversation Template from Palette

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Drag Lisp Conversation Template from Palette</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>15</td>
</tr>
<tr>
<td>Description</td>
<td>Tests that dragging a Lisp conversation from the palette creates the appropriate UmlConversationNode.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. The palette contains a Lisp conversation template that represents code which should create child UML nodes.</td>
</tr>
<tr>
<td>Steps</td>
<td>1) User drags a conversation node from the palette to the diagram.</td>
</tr>
<tr>
<td>Expected Results</td>
<td>A UmlConversationNode has been added to the diagram with the correct contents and child nodes corresponding to the Lisp code the palette node represents.</td>
</tr>
<tr>
<td>Table E.16: Minimize open Lisp Conversation Template</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Test Case ID</td>
<td>Minimize open Lisp Conversation Template</td>
</tr>
<tr>
<td>Test Case Number</td>
<td>16</td>
</tr>
<tr>
<td>Description</td>
<td>Verifies that a lisp conversation with children can have all its children hidden through the minimization operation.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A UmlConversationNode with children is present in the diagram and all children are currently visible.</td>
</tr>
<tr>
<td>Steps</td>
<td>1) User selects the minimization icon on the UmlConversationNode.</td>
</tr>
<tr>
<td>Expected Results</td>
<td>All the children of the respective UmlConversationNode are hidden.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table E.17: Maximize open Lisp Conversation Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case ID</td>
</tr>
<tr>
<td>Test Case Number</td>
</tr>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Pre Conditions</td>
</tr>
<tr>
<td>Steps</td>
</tr>
<tr>
<td>Expected Results</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table E.18: Close open Lisp Conversation Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case ID</td>
</tr>
<tr>
<td>Test Case Number</td>
</tr>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Pre Conditions</td>
</tr>
<tr>
<td>Steps</td>
</tr>
<tr>
<td>Expected Results</td>
</tr>
</tbody>
</table>
Table E.19: Add New Lisp Conversation Template to Palette

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Add New Lisp Conversation Template to Palette</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>19</td>
</tr>
<tr>
<td>Description</td>
<td>Tests the adding of a new lisp conversation template to the palette.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. Their is no conversation node in the palette with the name 'new-conversation'.</td>
</tr>
</tbody>
</table>
| Steps         | 1) User right clicks the speech act diagram’s background.  
                 2) User selects “Open/Create Conversation…”  
                 3) User enters the name ‘new-conversation’ in the popup.  
                 4) User clicks ‘OK’. |
| Expected Results | A new conversation template with the name ‘new-conversation’ has been added to the speech act diagram and a representative node has been added to the palette. |

Table E.20: Add Global Policy to new Lisp Conversation Template

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Add Global Policy to new Lisp Conversation Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>20</td>
</tr>
<tr>
<td>Description</td>
<td>Tests adding a global policy to a lisp conversation template.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A conversation template with no children is present in the speech act diagram.</td>
</tr>
</tbody>
</table>
| Steps         | 1) User right clicks the UmlConversationNode.  
                 2) User selects “Add Node” followed by submenu item “Policy”  
                 3) User selects “Yes” when prompted for whether or not global. |
| Expected Results | The conversation has a global policy as a child. |
Table E.21: Add Global Policy to Lisp Conversation Template

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Add Global Policy to Lisp Conversation Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>21</td>
</tr>
<tr>
<td>Description</td>
<td>Tests adding a global policy to a lisp conversation template automatically moves to the head of the child list.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A conversation template with a single local policy child is present in the speech act diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the UmlConversationNode.  
| | 2) User selects “Add Node” followed by submenu item “Policy”  
| | 3) User selects “Yes” when prompted for whether or not global. |
| Expected Results | The conversation has a global policy as a child occurring before the local policy. |

Table E.22: Add Local Policy to new Lisp Conversation Template

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Add Local Policy to new Lisp Conversation Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>22</td>
</tr>
<tr>
<td>Description</td>
<td>Tests adding a local policy to a lisp conversation template.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A conversation template with no children is present in the speech act diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the UmlConversationNode.  
| | 2) User selects “Add Node” followed by submenu item “Policy”  
| | 3) User selects “No” when prompted for whether or not global. |
| Expected Results | The conversation has a local policy as a child. |
Table E.23: Add Comment to new Lisp Conversation Template

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Add Comment to new Lisp Conversation Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>23</td>
</tr>
<tr>
<td>Description</td>
<td>Tests adding a comment to a lisp conversation template.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A conversation template with no children is present in the speech act diagram.</td>
</tr>
</tbody>
</table>
| Steps            | 1) User right clicks the UmlConversationNode.  
                   | 2) User selects “Add Node” followed by submenu item “Comment” |
| Expected Results | The conversation has a comment policy as a child. |

Table E.24: Add Code Block to new Lisp Conversation Template

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Add Code Block to new Lisp Conversation Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>24</td>
</tr>
<tr>
<td>Description</td>
<td>Tests adding a code block to a lisp conversation template.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A conversation template with no children is present in the speech act diagram.</td>
</tr>
</tbody>
</table>
| Steps            | 1) User right clicks the UmlConversationNode.  
                   | 2) User selects “Add Node” followed by submenu item “General Lisp Code” |
| Expected Results | The conversation has a code block as a child. |

Table E.25: Set ordinality of global policy node to beginning

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Set ordinality of global policy node to beginning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>25</td>
</tr>
<tr>
<td>Description</td>
<td>Tests setting ordinality of a global policy under a lisp conversation template around other global policies.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A conversation template with two global policy children is present in the speech act diagram.</td>
</tr>
</tbody>
</table>
| Steps            | 1) User right clicks the second UmlPolicyNode.  
                   | 2) User selects “Change Ordinality”  
                   | 3) User chooses index 0 and clicks “OK” |
| Expected Results | The conversation’s global policy has moved to the first position. |
Table E.26: Set position of node to end

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Set position of node to end</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>26</td>
</tr>
<tr>
<td>Description</td>
<td>Tests setting ordinality of a node under a lisp conversation template.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A conversation template with a global policy child and two local policy children (in that order) is present in the speech act diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the second UmlPolicyNode.  
       2) User selects “Change Ordinality”  
       3) User chooses index 2 and clicks “OK” |
| Expected Results | The conversation’s first local policy has moved to the last position. |

Table E.27: Add Conditional with default condition to Lisp Conversation Template

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Add Conditional with default condition to Lisp Conversation Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>27</td>
</tr>
<tr>
<td>Description</td>
<td>Tests the warning popup when adding a default conditional to a lisp conversation template.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A conversation template with no children is present in the speech act diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the UmlConversationNode.  
       2) User selects “Add Node” followed by submenu item “Conditional”  
       3) User selects “OK” |
| Expected Results | A warning popup informs the user that the condition does not occur anywhere in the parent conversation. |
### Table E.28: Add Conditional with proper condition to Lisp Conversation Template

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Test Case Number</th>
<th>Description</th>
<th>Pre Conditions</th>
<th>Steps</th>
<th>Expected Results</th>
</tr>
</thead>
</table>
|              |                  | Add Conditional with proper condition to Lisp Conversation Template | The system is running. A conversation template with no children is present in the speech act diagram. | 1) User right clicks the UmlConversationNode.  
2) User selects “Add Node” followed by submenu item “Conditional”  
3) User enters condition ‘conversation-name’ and selects “OK” | The conversation now has a conditional as a child with two empty code node children. |

### Table E.29: Remove Conditional’s ‘then-case’ child

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Test Case Number</th>
<th>Description</th>
<th>Pre Conditions</th>
<th>Steps</th>
<th>Expected Results</th>
</tr>
</thead>
</table>
|              |                  | Remove Conditional’s ‘then-case’ child | The system is running. A conversation template with a default conditional child is present in the speech act diagram. | 1) User right clicks the ‘then-case’ node.  
2) User selects “Unlink from parent” | The ‘then-case’ child node is no longer a child under the conditional. |

### Table E.30: Add Comment to Conditional

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Test Case Number</th>
<th>Description</th>
<th>Pre Conditions</th>
<th>Steps</th>
<th>Expected Results</th>
</tr>
</thead>
</table>
|              |                  | Add Comment to Conditional | The system is running. A conversation template with a default conditional child is present in the speech act diagram. | 1) User right clicks the UmlConditional node.  
2) User selects “Add Node” followed by submenu item “Comment” | The conditional now has a comment node added to it. |
### Table E.31: Add Local Policy to Conditional

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Add Local Policy to Conditional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>31</td>
</tr>
<tr>
<td>Description</td>
<td>Tests adding a local policy node to a conditional.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A conversation template with a conditional child is present in the speech act diagram and the conditional has only one child.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the UmlConditional node.  
2) User selects “Add Node” followed by submenu item “Policy”  
3) User selects “No” when prompted for whether or not global. |
| Expected Results | The conditional now has a local policy node added to it in whatever position (then or else) that the single child was not in. |

### Table E.32: Swap children of Conditional

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Swap children of Conditional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>32</td>
</tr>
<tr>
<td>Description</td>
<td>Tests swapping children of a conditional.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A conversation template with a conditional child is present in the speech act diagram and the conditional has two children.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the UmlConditionalNode.  
2) User selects “Swap Then-Else” |
<p>| Expected Results | The conditional’s children have swapped condition positions. |</p>
<table>
<thead>
<tr>
<th>Table E.33: Add Global Policy to Conditional</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Test Case ID</strong></td>
</tr>
<tr>
<td><strong>Test Case Number</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Pre Conditions</strong></td>
</tr>
</tbody>
</table>
| **Steps** | 1) User right clicks the UmlConditionalNode.  
2) User selects “Add Node” followed by submenu item “Policy”  
3) User selects “Yes” when prompted for whether or not global. |
| **Expected Results** | The conditional has assumed the index of 1, just after the first global policy. |

<table>
<thead>
<tr>
<th>Table E.34: Add Conversation call to new Lisp Conversation Template</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Test Case ID</strong></td>
</tr>
<tr>
<td><strong>Test Case Number</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Pre Conditions</strong></td>
</tr>
</tbody>
</table>
| **Steps** | 1) User right clicks the UmlConversationNode.  
2) User selects “Add Node” followed by submenu item “Conversation...”  
3) User selects the desired conversation template from the drop down. 4) User selects “OK” |
| **Expected Results** | A conversation call is present as a child to the parent conversation. |
### Table E.35: Bind Variable to Conversation call

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Bind Variable to Conversation call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>35</td>
</tr>
<tr>
<td>Description</td>
<td>Tests binding of variables between parent-child conversation templates results in an equivalence relation being created.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A conversation template with a conversation call child is present in the speech act diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the child `UmlConversationNode`.  
2) User selects “Bind Variables”  
3) User selects the desired variables from the drop downs.  
4) User selects “OK” |
| Expected Results | An equivalence relation is present between the two conversations pointing at the variables. |

### Table E.36: Bind State to Conversation call

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Bind State to Conversation call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>36</td>
</tr>
<tr>
<td>Description</td>
<td>Tests binding of state between parent-child conversation templates results in an equivalence relation being created.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A conversation template with a conversation call child is present in the speech act diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the child `UmlConversationNode`.  
2) User selects “Bind Variables”  
3) User selects the desired states from the drop down.  
4) User selects “OK” |
| Expected Results | An equivalence relation is present between the two conversations pointing at the states. |

### Table E.37: Add Imported Conversation Call

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Add Imported Conversation Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>37</td>
</tr>
<tr>
<td>Description</td>
<td>Tests that imported conversation templates can be added as conversation calls.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A conversation template with a no children is present in the speech act diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the `UmlConversationNode`.  
2) User selects “Add Node” followed by submenu item “Conversation...”  
3) User selects the desired conversation template from the drop down.  
4) User selects “OK” |
| Expected Results | A conversation call is present as a child to the parent conversation. |
Table E.38: Switch Contains Visibilities

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Switch Contains Visibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>38</td>
</tr>
<tr>
<td>Description</td>
<td>Tests that contains relations can be removed from visibility.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A conversation template with at least one child is present in the speech act diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the diagram’s background.  
2) User selects “Visibility Selections” and submenu item “Contains Visibility” |
| Expected Results | All contains relations have been removed from the diagram. |

Table E.39: Switch Equivalence Visibilities

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Switch Equivalence Visibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>39</td>
</tr>
<tr>
<td>Description</td>
<td>Tests that equivalence relations can be removed from visibility.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A equivalence relation is present in the speech act diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the diagram’s background.  
2) User selects “Visibility Selections” and submenu item “Equivalence Visibility” |
| Expected Results | All equivalence relations have been removed from the diagram. |

Table E.40: Switch Transition Visibilities

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Switch Transition Visibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>40</td>
</tr>
<tr>
<td>Description</td>
<td>Tests that transition relations can be removed from visibility.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A transition relation is present in the diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the diagram’s background.  
2) User selects “Visibility Selections” and submenu item “State Visibility” |
| Expected Results | All transition relations have been removed from the diagram. |
### Table E.41: Switch Comment Visibilities

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Switch Comment Visibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>41</td>
</tr>
<tr>
<td>Description</td>
<td>Tests that comment nodes can be removed from visibility.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A comment node is present in the diagram.</td>
</tr>
</tbody>
</table>
| Steps         | 1) User right clicks the diagram’s background. 
2) User selects “Visibility Selections” and submenu item “Comment Node Visibility” |
| Expected Results | All comment nodes have been removed from the diagram. |

### Table E.42: Switch Code block Visibilities

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Switch Code block Visibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>42</td>
</tr>
<tr>
<td>Description</td>
<td>Tests that code block nodes can be removed from visibility.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A code block is present in the diagram.</td>
</tr>
</tbody>
</table>
| Steps         | 1) User right clicks the diagram’s background. 
2) User selects “Visibility Selections” and submenu item “Code Node Visibility” |
<p>| Expected Results | All code block nodes have been removed from the diagram. |</p>
<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Switch Policy Visibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>43</td>
</tr>
<tr>
<td>Description</td>
<td>Tests that policy nodes can be removed from visibility.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A policy node (local or global) is present in the diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the diagram’s background.  
2) User selects “Visibility Selections” and submenu item “Policy Visibility” |
| Expected Results | All policy nodes have been removed from the diagram. |

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Switch Conversation Visibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>44</td>
</tr>
<tr>
<td>Description</td>
<td>Tests that conversation nodes can be removed from visibility.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A conversation node is present in the diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the diagram’s background.  
2) User selects “Visibility Selections” and submenu item “Conversation Visibility” |
| Expected Results | All conversation nodes have been removed from the diagram. |

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Switch Mouse Over Text Visibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>45</td>
</tr>
<tr>
<td>Description</td>
<td>Tests that conversation nodes can be removed from visibility.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A node is present in the speech act diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the diagram’s background.  
2) User selects “Visibility Selections” and submenu item “Mouse Over Text”  
3) User moves mouse over the node’s label area. |
| Expected Results | No mouse over message is generated within 5 seconds (supposed to be instant). |
### Table E.46: Reactivate all Visibilities

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Reactivate all Visibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>46</td>
</tr>
<tr>
<td>Description</td>
<td>Tests that reactivating visibilities also works.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. Every type of node and relation should be present in the diagram. All Visibilities are deselected.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the diagram’s background.  
2) User selects to activate every visibility in the menu. |
| Expected Results | All relations and nodes have been re-added to the diagram. |

### Table E.47: Remove child from Lisp Conversation Template

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Remove child from Lisp Conversation Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>47</td>
</tr>
<tr>
<td>Description</td>
<td>Tests that children can be removed from conversation templates.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A conversation template with at least one child is present in the speech act diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the child  
2) User selects “Unlink from parent” |
| Expected Results | The child is no longer present under the parent conversation. |

### Table E.48: Edit documentation of Lisp Conversation Template

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Edit documentation of Lisp Conversation Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>48</td>
</tr>
<tr>
<td>Description</td>
<td>Tests that nodes with documentation can receive edits.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A conversation template is present in the speech act diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the node  
2) User selects “Edit Documentation”  
3) User modifies dialogue contents and selects “OK”  
4) User saves changes to templates, removes the node, then adds it again. |
| Expected Results | The documentation changes have been retained. |
Table E.49: Edit comments of Lisp Conversation Template’s variable

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Edit comments of Lisp Conversation Template’s variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>49</td>
</tr>
<tr>
<td>Description</td>
<td>Tests that nodes with documentation can receive edits.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. A conversation template is present in the speech act diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks one of the node’s variables.  
2) User selects “Edit Comments”  
3) User modifies dialogue contents and selects “OK”  
4) User saves changes to templates, removes the node, then adds it again. |
| Expected Results | The changes to the corresponding variable’s comments have been retained. |

Table E.50: Open Agent initialization file

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Open Agent initialization file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>50</td>
</tr>
<tr>
<td>Description</td>
<td>Tests that agent init files can be opened.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. The action level diagram is selected.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the action level diagram’s background.  
2) User selects “Open agent init file...”  
3) User navigates the file system for the file in question and selects “OK”. |
| Expected Results | A file node is now present on the diagram parenting all comments, code blocks, global policies, and conversation calls that the file defines. |
### Table E.51: Create Agent initialization file

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Create Agent initialization file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>51</td>
</tr>
<tr>
<td>Description</td>
<td>Tests that agent init files can be created.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. The action level diagram is selected.</td>
</tr>
</tbody>
</table>
| Steps   | 1) User right clicks the action level diagram’s background.  
|         | 2) User selects “Create agent init file...”  
|         | 3) User navigates the file system for the desired location.  
|         | 4) User enters a file name and selects “OK”. |
| Expected Results | A file node has been added to the diagram. |

### Table E.52: Drag Conversation Call from Palette with success

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Drag Conversation Call from Palette with success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>52</td>
</tr>
<tr>
<td>Description</td>
<td>Tests that conversation calls can be created.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. The action level diagram is selected. The palette contains at least one conversation node.</td>
</tr>
</tbody>
</table>
| Steps   | 1) User drags a conversation node from the palette to the diagram.  
<p>|         | 2) User fills in all desired values (and all required) and selects “OK”. |
| Expected Results | A conversation call is now present on the diagram. |</p>
<table>
<thead>
<tr>
<th>Table E.53: Drag Conversation Call from Palette with fail</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Test Case ID</strong></td>
</tr>
<tr>
<td><strong>Test Case Number</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Pre Conditions</strong></td>
</tr>
</tbody>
</table>
| **Steps** | 1) User drags a conversation node from the palette to the diagram.  
2) User fills in none of the required values and selects “OK”. |
| **Expected Results** | An error message is displayed to the user with the option to try again or cancel. |

<table>
<thead>
<tr>
<th>Table E.54: Drag Conversation Call from Palette with cancel</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Test Case ID</strong></td>
</tr>
<tr>
<td><strong>Test Case Number</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Pre Conditions</strong></td>
</tr>
</tbody>
</table>
| **Steps** | 1) User drags a conversation node from the palette to the diagram.  
2) User selects “Cancel”. |
| **Expected Results** | The diagram remains unchanged. |

<table>
<thead>
<tr>
<th>Table E.55: Add Comment to File Node</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Test Case ID</strong></td>
</tr>
<tr>
<td><strong>Test Case Number</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Pre Conditions</strong></td>
</tr>
</tbody>
</table>
| **Steps** | 1) User right clicks the UmlFileNode.  
2) User selects “Add Node” followed by the submenu item “Comment Block” |
| **Expected Results** | The file node now has an extra comment node added. |
Table E.56: Add Code Block to File Node

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Add Code Block to File Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>56</td>
</tr>
<tr>
<td>Description</td>
<td>Tests adding a code block node to a file node.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. The action level diagram is selected. There is a file node present on the action level diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the UmlFileNode.  
2) User selects “Add Node” followed by the submenu item “Code Block” |
| Expected Results | The file node now has an extra code block node added. |

Table E.57: Add Global Policy to File Node with success

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Add Global Policy to File Node with success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>57</td>
</tr>
<tr>
<td>Description</td>
<td>Tests adding a global policy node to a file node.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. The action level diagram is selected. There is a file node present on the action level diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the UmlFileNode.  
2) User selects “Add Node” followed by the submenu item “Global Policy”  
3) User fills in all the fields and selects “OK” |
| Expected Results | The file node now has an extra global policy node added. |

Table E.58: Add Global Policy to File Node with cancel

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Add Global Policy to File Node with cancel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>58</td>
</tr>
<tr>
<td>Description</td>
<td>Tests cancelling the addition of a comment node to a file node.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. The action level diagram is selected. There is a file node present on the action level diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the UmlFileNode.  
2) User selects “Add Node” followed by the submenu item “Comment Block”  
3) User fills in all the fields and selects “Cancel” |
| Expected Results | The diagram is unchanged. |
Table E.59: Migrate Conversation Call to File Node

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Migrate Conversation Call to File Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>59</td>
</tr>
<tr>
<td>Description</td>
<td>Tests moving a conversation call node to a file node.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. The action level diagram is selected. There is a file node present on the action level diagram. There is a conversation call present in the action level diagram.</td>
</tr>
</tbody>
</table>
| Steps         | 1) User right clicks the UmlConversationNode.  
                 2) User selects “Add To File...”  
                 3) User chooses the name of the file node to add the call to from the dropdown. |
| Expected Results | The conversation node has left its previous position to be the last child of the file node. |

Table E.60: Save Agent Initialization File

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Save Agent Initialization File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>60</td>
</tr>
<tr>
<td>Description</td>
<td>Tests saving an agent initialization file.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. The action level diagram is selected. There is a file node present on the action level diagram.</td>
</tr>
</tbody>
</table>
| Steps         | 1) User right clicks the UmlFileNode.  
                 2) User selects “Save To Init File...”  
                 3) User navigates to a new file path. 4) User enters a new file name and selects “OK” |
| Expected Results | A file is present with the given path/name combination containing all the comments, code blocks, global policies, and conversation calls that were defined in the diagram. |
Table E.61: Modify agent initialization files and save

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Modify agent initialization files and save</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>61</td>
</tr>
<tr>
<td>Description</td>
<td>Tests modifying referenced agent init files and overwrites.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>Start in CASA Agent Context The action level diagram is selected. There is a UmlAgentNode present on the action level diagram. There is at least one file node present under the agent node and it is visible.</td>
</tr>
<tr>
<td>Steps</td>
<td>1) User right clicks the UmlFileNode 2) User selects “Save To Init File...” 3) User selects “OK”</td>
</tr>
<tr>
<td>Expected Results</td>
<td>The original file has been overwritten with the given path/name combination containing all the comments, code blocks, global policies, and conversation calls that were defined in the diagram.</td>
</tr>
</tbody>
</table>

Table E.62: Play Conversation Instance Animation

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Play Conversation Instance Animation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>62</td>
</tr>
<tr>
<td>Description</td>
<td>Test conversation instance animation playback.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. The instance level diagram is selected. There is an observed conversation instance on the instance diagram.</td>
</tr>
<tr>
<td>Steps</td>
<td>1) User right clicks the top most UmlConversationNode 2) User selects “Play Animation”</td>
</tr>
<tr>
<td>Expected Results</td>
<td>A time delayed animation takes place of the conversation instance’s progression.</td>
</tr>
</tbody>
</table>
### Table E.63: Clear Conversation Instance Animation

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Clear Conversation Instance Animation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>63</td>
</tr>
<tr>
<td>Description</td>
<td>Test conversation instance animation reset.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. The instance level diagram is selected. There is an observed conversation instance on the instance diagram. The conversation has recently been animated.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the top most UmlConversationNode  
2) User selects “Reset Animation” |
| Expected Results | The conversation takes on its default form with no bolded state variables, all contains relations left blank, and the mouse over text contains no event sequence. |

### Table E.64: Close Conversation Instance

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Close Conversation Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Number</td>
<td>64</td>
</tr>
<tr>
<td>Description</td>
<td>Tests closing conversation instances.</td>
</tr>
<tr>
<td>Pre Conditions</td>
<td>The system is running. The instance level diagram is selected. There is an observed conversation instance on the instance diagram.</td>
</tr>
</tbody>
</table>
| Steps | 1) User right clicks the top most UmlConversationNode  
2) User selects the close icon on the UmlConversationNode. |
| Expected Results | The conversation instance is no longer a child of the agent node and cannot be recovered through the use of the maximization icon. |