Implementing Norm-Governed Multi-Agent Systems

Robin Catherine Bennett

November 2013

Supervised by Marek Sergot

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing of Imperial College London and the Diploma of Imperial College London
Copyright Declaration

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.
Abstract

The actions and interactions of independently acting agents in a multi-agent system must be managed if the agents are to function effectively in their shared environment. Norms, which define the obligatory, prohibited and permitted actions for an agent to perform, have been suggested as a possible method for regulating the actions of agents.

Norms are local rules designed to govern the actions of individual agents whilst also allowing the agents to achieve a coherent global behaviour. However, there appear to be very few instances of norm-governed multi-agent systems beyond theoretical examples.

We describe an implementation strategy for allowing autonomous agents to take a set of norms into account when determining their actions. These norms are implemented using directives, which are local rules specifying actions for an agent to perform depending on its current state. Agents using directives are implemented in a simulation test bed, called Sinatra. Using Sinatra, we investigate the ability of directives to manage agent actions.

We begin with directives to manage agent interactions. We find that when agents rely on only local rules they will encounter situations where the local rules are unable to achieve the desired global behaviour.

We show how a centralised control mechanism can be used to manage agent interactions that are not successfully handled by directives. Controllers, with a global view of the interaction, instruct the individual agents how to act. We also investigate the use of an existing planning tool to implement the resolution mechanism of a controller.

We investigate the ability of directives to coordinate the actions of agents in order to achieve a global objective more effectively. Finally, we present a case study of how directives can be used to determine the actions of autonomous mobile robots.
Acknowledgements

I would like to express my sincere thanks and appreciation to the following people. Without their support and contributions, this thesis would not have been possible.

- To my supervisor, Marek Sergot, who has guided and encouraged me, not only during my PhD, but throughout my time at Imperial.

- To David McBride and Gareth Smith, who have generously shared words of wisdom; in particular, their insights on achieving a PhD and on the nature of spoons.

- To Mathew, whose patience, understanding and willingness to accept unusual working hours allowed this thesis to be written.

- To my parents, Carolyn and Jeremy, who have supported and encouraged me through all my endeavours.
Statement of Originality

I declare that the content of this thesis is the product of my own work and that all material that is not my own has been properly acknowledged.

- ROBIN CATHERINE BENNETT
Contents

Abstract 4

Acknowledgements 5

Statement of Originality 6

List of Tables 11

List of Figures 13

List of Listings 15

1 Introduction 17
   1.1 Taking a set of norms into account ............................. 18
   1.2 Using local rules to manage the actions of agents .............. 20
   1.3 Structure of the thesis ........................................... 20
   1.4 Contributions ..................................................... 22

2 Background 24
   2.1 Norms in multi-agent systems ................................. 24
      2.1.1 Related concepts ............................................ 26
      2.1.2 Areas of study ............................................... 28
   2.2 Implementing multi-agent systems ............................... 28
      2.2.1 Implementation tools for simulated multi-agent systems ... 29
      2.2.2 Real world multi-agent systems ............................ 32
   2.3 Implementing norms ............................................... 33
      2.3.1 Norm emergence .............................................. 33
      2.3.2 Norm adherence .............................................. 34
      2.3.3 Representation of norms ................................... 35
      2.3.4 Types of norms .............................................. 37
      2.3.5 Norm-aware BDI agents .................................... 38
      2.3.6 Strategies for norm compliance ............................ 40
      2.3.7 Distributed control of agents .............................. 41
      2.3.8 Norm violation mechanisms ................................. 43
      2.3.9 Norm conflict resolution .................................. 43
2.4 Priorities on rules ................................................. 44
  2.4.1 Non-monotonic reasoning with priorities ................. 45
  2.4.2 Strategies for using priorities ............................. 49
  2.4.3 Compiling priorities in logic programs .................... 58
2.5 Review .......................................................... 62

3 Sinatra: a simulation test bed for implementing norm-governed multi-agent systems 64
  3.1 Sinatra overview ................................................ 65
  3.2 Sinatra agent properties ....................................... 69
    3.2.1 Agent perceptions ......................................... 71
    3.2.2 Agent actions ............................................. 73
    3.2.3 Agent memory .............................................. 74
    3.2.4 Sinatra agents as mobile robots ......................... 74
  3.3 Behaviour method ............................................... 75
    3.3.1 Behaviours ................................................ 76
    3.3.2 Behaviour translation .................................... 84
    3.3.3 Using a behaviour translation ............................ 95
    3.3.4 Generating state-action tables ......................... 97
  3.4 Agents using behaviours ...................................... 103
    3.4.1 Understanding a state-action table entry ............... 103
    3.4.2 Using a state-action table ............................... 104
  3.5 Agents using multiple behaviours ............................ 106
    3.5.1 Implementing a behaviour-switch ....................... 107
    3.5.2 Using a behaviour-switch ............................... 108
    3.5.3 Incorporating a new behaviour ........................... 110
    3.5.4 Behaviours as actions .................................. 111
  3.6 Summary ....................................................... 113

4 Behaviours to manage agent interactions ................................ 114
  4.1 Managing undesirable agent interactions ..................... 115
  4.2 Example: Using behaviours to prevent stationary interactions ... 116
    4.2.1 Implementing the highway behaviour .................... 117
    4.2.2 Demonstration ............................................ 120
    4.2.3 Implementing the strict traffic lanes behaviour ....... 122
    4.2.4 Demonstration ............................................ 127
    4.2.5 Summary .................................................. 129
  4.3 Example: Using behaviours to resolve stationary interactions ... 130
    4.3.1 Implementing the obstacle avoidance behaviour ......... 131
    4.3.2 Demonstration ............................................ 134
    4.3.3 Implementing the traffic law behaviour .................. 139
7.2.2 Implementing the formation behaviour ................. 225
7.2.3 Observations and results ................................... 229
7.2.4 Summary ..................................................... 230
7.3 Example: Using behaviours to coordinate hide-and-seek agents ... 231
  7.3.1 Implementing the seeking agents ......................... 232
  7.3.2 Implementing the coordinated search behaviour .......... 237
  7.3.3 Observations and results ................................... 241
  7.3.4 Summary ..................................................... 243
7.4 Conclusion ..................................................... 244
  7.4.1 Coordination through expected patterns of behaviour .... 244
  7.4.2 Coordination through agent communication ............... 245
  7.4.3 Local coordination using behaviours ..................... 246
8 Case study: Autonomous robot assistants 248
  8.1 Case study outline ............................................. 248
  8.2 Moving around the department ................................. 250
    8.2.1 Safe robot movement behaviour ......................... 250
    8.2.2 Robot movement behaviour-switch ...................... 255
  8.3 Robot assistant behaviours ................................... 256
    8.3.1 Visitor guides ............................................. 257
    8.3.2 Helping staff .............................................. 260
    8.3.3 Lost child ................................................ 265
    8.3.4 Robot assistant behaviour-switch ....................... 269
  8.4 Discussion ..................................................... 271
    8.4.1 Behaviours ................................................ 271
    8.4.2 Sinatra simulation ........................................ 273
  8.5 Case study review ............................................. 274
9 Conclusion ...................................................... 277
  9.1 Taking a set of norms into account using behaviours ........... 277
  9.2 Comparison of Sinatra and Packet-World ...................... 278
    9.2.1 Simulation and visualisation ............................. 280
    9.2.2 Environment .............................................. 280
    9.2.3 Agents ................................................... 281
  9.3 Using behaviours to manage the actions of agents ............ 282
    9.3.1 Using behaviours to manage agent interactions .......... 283
    9.3.2 Using controllers when behaviours are unsuccessful .... 283
    9.3.3 Using behaviours to coordinate agent actions .......... 285
  9.4 Developing behaviours for a norm-governed application ....... 286

Bibliography 289
List of Tables

2.1 Classification of Hory’s method. .......................... 54
2.2 Classification of Hansen’s method. ......................... 57
2.3 Classification of the method of Delgrande et al. ........ 61

3.1 The perception methods of the Sinatra agents. .......... 72
3.2 The action methods of the Sinatra agents. ............... 74
3.3 The memory capabilities of the Sinatra agents. ........ 74
3.4 Classification of the behaviour method. .................. 84
3.5 The dictionary of tokens for the simple goal-directed navigation. . 104
3.6 The dictionary of tokens for the example master behaviour-switch. . 109
3.7 The methods used by Sinatra agents to consult a state-action table. 111

4.1 The perception methods for the highway behaviour. ........... 119
4.2 The dictionary of tokens for the highway behaviour. ....... 119
4.3 The perception methods for the strict traffic lanes behaviour. .. 124
4.4 The dictionary of tokens for the strict traffic lanes behaviour. .... 124
4.5 The perception methods for the traffic law behaviour. ........ 140
4.6 The dictionary of tokens for the traffic law behaviour. .......... 141

6.1 Plan-generation time and size of the generated output file for the row
obstruction interaction domain containing different numbers of agents. 206

7.1 The perception methods for the robot rugby domain. ....... 221
7.2 (part 1) The action methods for the robot rugby domain. .... 223
7.1 (part 2) The action methods for the robot rugby domain. .... 224
7.2 The dictionary of tokens for the wedge formation behaviour. .. 227
7.3 Overall results of the robot rugby simulations, showing the actual and
percentage distribution of game results. ....................... 230
7.4 The communication methods for the coordinated search behaviour. 239
7.5 Overall results of the hide-and-seek simulations, showing the average
number of time steps taken for the seeking agents to find the hiding
agent. ................................................................. 242

8.1 The dictionary of tokens for the safe robot movement behaviour. ... 253
8.2 The dictionary of tokens for the robot movement behaviour-switch. . 257
8.3 The dictionary of tokens for the guide behaviour. . . . . . . . . . . . . . . . . . . . . . . . . . . 259
8.4 The dictionary of tokens for the deliver behaviour. . . . . . . . . . . . . . . . . . . . . . . . . . 264
8.5 The dictionary of tokens for the search behaviour. . . . . . . . . . . . . . . . . . . . . . . . . . . 268
8.6 The dictionary of tokens for the robot assistant behaviour-switch . . 271
List of Figures

3.1 Sinatra GUI showing three agents in a grid environment. . . . . . . . 65
3.2 The Sinatra UML class diagram. . . . . . . . . . . . . . . . . . . . . 67
3.3 The Sinatra agent architecture. . . . . . . . . . . . . . . . . . . . . 71
3.4 The STAG process. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
3.5 A single agent in Sinatra using the simple goal-directed navigation. . 106
4.1 An undesirable agent interaction. . . . . . . . . . . . . . . . . . . . . 114
4.2 Two agents in Sinatra using the highway behaviour. . . . . . . . . . . 120
4.3 A stationary interaction caused by the highway behaviour. . . . . . . . 122
4.4 Two agents in Sinatra using the strict traffic lanes behaviour. . . . . . 127
4.5 An agent trapped in the south-east corner of the grid. . . . . . . . . . 128
4.6 The exception case to the strict traffic lanes behaviour allows station-
ary interactions to occur. . . . . . . . . . . . . . . . . . . . . . . . . . . . 129
4.7 Demonstration of the obstacle avoidance behaviour (part 1). . . . . . 135
4.8 Demonstration of the obstacle avoidance behaviour (part 2). . . . . . 135
4.9 Demonstration of the obstacle avoidance behaviour (part 3). . . . . . 135
4.10 Demonstration of the obstacle avoidance behaviour (part 4). . . . . . 136
4.11 Demonstration of a repeated state interaction (part 1). . . . . . . . . 137
4.12 Demonstration of a repeated state interaction (part 2). . . . . . . . . 137
4.13 Demonstration of a repeated state interaction (part 3). . . . . . . . . 137
4.14 Demonstration of a repeated state interaction (part 4). . . . . . . . . 138
4.15 Demonstration of a repeated state interaction (part 5). . . . . . . . . 138
4.16 Demonstration of a repeated state interaction (part 6). . . . . . . . . 138
4.17 Demonstration of a repeated state interaction (part 7). . . . . . . . . 139
4.18 Demonstration of the traffic law behaviour (part 1). . . . . . . . . . . 144
4.19 Demonstration of the traffic law behaviour (part 2). . . . . . . . . . . 144
4.20 Demonstration of the traffic law behaviour (part 3). . . . . . . . . . . 144
4.21 Demonstration of the traffic law behaviour (part 4). . . . . . . . . . . 145
4.22 A stationary interaction that cannot be resolved by the traffic law
behaviour. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 145
5.1 Demonstration of the repeated state controller (part 1). . . . . . . . . 167
5.2 Demonstration of the repeated state controller (part 2). . . . . . . . . 167
5.3 Demonstration of the repeated state controller (part 3) ........ 167
5.4 Demonstration of the repeated state controller (part 4) ........ 168
5.5 Demonstration of the repeated state controller (part 5) ........ 168
5.6 Demonstration of the repeated state controller (part 6) ........ 168
5.7 Demonstration of the repeated state controller (part 7) ........ 169
5.8 Demonstration of the repeated state controller (part 8) ........ 169
5.9 Demonstration of the repeated state controller (part 9) ........ 169
5.10 Demonstration of the stationary controller (part 1) ........... 176
5.11 Demonstration of the stationary controller (part 2) ........... 177
5.12 Demonstration of the stationary controller (part 3) ........... 177
5.13 Demonstration of the stationary controller (part 4) ........... 177

6.1 A row obstruction interaction ...................................... 196
6.2 Demonstration of the UMOP controller (part 1) ............... 203
6.3 Demonstration of the UMOP controller (part 2) ............... 203
6.4 Demonstration of the UMOP controller (part 3) ............... 204
6.5 Demonstration of the UMOP controller (part 4) ............... 204
6.6 Demonstration of the UMOP controller (part 5) ............... 204
6.7 Demonstration of the UMOP controller (part 6) ............... 205

7.1 Two teams in the robot rugby domain ............................ 224
7.2 The blue team in a wedge formation ............................. 226
7.3 The blue team in a line formation ............................... 228
7.4 Three agents in the hide-and-seek domain ...................... 237

9.1 The Sinatra implementation of the Packet-World domain ....... 279
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The simple goal-directed navigation input file.</td>
<td>101</td>
</tr>
<tr>
<td>3.2</td>
<td>Excerpt from the state-action table for the simple goal-directed navigation.</td>
<td>102</td>
</tr>
<tr>
<td>3.3</td>
<td>The state-action table for the example master behaviour-switch.</td>
<td>108</td>
</tr>
<tr>
<td>4.1</td>
<td>The highway behaviour input file.</td>
<td>121</td>
</tr>
<tr>
<td>4.2</td>
<td>(part 1) The strict traffic lanes behaviour input file.</td>
<td>125</td>
</tr>
<tr>
<td>4.1</td>
<td>(part 2) The strict traffic lanes behaviour input file.</td>
<td>126</td>
</tr>
<tr>
<td>4.2</td>
<td>(part 1) The obstacle avoidance behaviour input file.</td>
<td>132</td>
</tr>
<tr>
<td>4.1</td>
<td>(part 2) The obstacle avoidance behaviour input file.</td>
<td>133</td>
</tr>
<tr>
<td>4.2</td>
<td>The behaviour-switch input file for the traffic law behaviour.</td>
<td>141</td>
</tr>
<tr>
<td>4.3</td>
<td>The state-action table generated from the behaviour-switch for the traffic law behaviour.</td>
<td>142</td>
</tr>
<tr>
<td>4.4</td>
<td>The traffic law behaviour input file.</td>
<td>143</td>
</tr>
<tr>
<td>5.1</td>
<td>The behaviour-switch input file to invoke the repeated state controller.</td>
<td>162</td>
</tr>
<tr>
<td>5.2</td>
<td>The behaviour-switch input file to follow the instructions of the repeated state controller.</td>
<td>166</td>
</tr>
<tr>
<td>5.3</td>
<td>The behaviour-switch input file to invoke the stationary controller.</td>
<td>172</td>
</tr>
<tr>
<td>5.4</td>
<td>The behaviour-switch input file to follow the instructions of the stationary controller.</td>
<td>175</td>
</tr>
<tr>
<td>6.1</td>
<td>A toy NADL domain description.</td>
<td>192</td>
</tr>
<tr>
<td>6.2</td>
<td>Plan-simulation of the toy NADL domain.</td>
<td>195</td>
</tr>
<tr>
<td>6.3</td>
<td>(part 1) The row obstruction interaction domain description.</td>
<td>199</td>
</tr>
<tr>
<td>6.2</td>
<td>(part 2) The row obstruction interaction domain description.</td>
<td>200</td>
</tr>
<tr>
<td>6.3</td>
<td>(part 1) Plan-simulation of the row obstruction interaction.</td>
<td>201</td>
</tr>
<tr>
<td>6.2</td>
<td>(part 2) Plan-simulation of the row obstruction interaction.</td>
<td>202</td>
</tr>
<tr>
<td>6.3</td>
<td>(part 1) Plan-simulation of the three-agent row obstruction interaction.</td>
<td>210</td>
</tr>
<tr>
<td>6.2</td>
<td>(part 2) Plan-simulation of the three-agent row obstruction interaction.</td>
<td>211</td>
</tr>
<tr>
<td>6.3</td>
<td>(part 1) Plan-simulation of the three-agent adversarial row obstruction interaction.</td>
<td>212</td>
</tr>
<tr>
<td>6.2</td>
<td>(part 2) Plan-simulation of the three-agent adversarial row obstruction interaction.</td>
<td>213</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>7.1</td>
<td>The rugby behaviour input file.</td>
<td>225</td>
</tr>
<tr>
<td>7.2</td>
<td>The wedge formation behaviour input file</td>
<td>227</td>
</tr>
<tr>
<td>7.3</td>
<td>The formation behaviour input file.</td>
<td>229</td>
</tr>
<tr>
<td>7.4</td>
<td>The A* search algorithm</td>
<td>235</td>
</tr>
<tr>
<td>7.5</td>
<td>The independent search behaviour input file.</td>
<td>236</td>
</tr>
<tr>
<td>7.6</td>
<td>The coordinated search behaviour input file.</td>
<td>241</td>
</tr>
<tr>
<td>8.1</td>
<td>The safe robot movement behaviour input file.</td>
<td>254</td>
</tr>
<tr>
<td>8.2</td>
<td>The robot movement behaviour-switch input file.</td>
<td>256</td>
</tr>
<tr>
<td>8.3</td>
<td>The guide behaviour input file.</td>
<td>259</td>
</tr>
<tr>
<td>8.4</td>
<td>(part 1) The deliver behaviour input file.</td>
<td>262</td>
</tr>
<tr>
<td>8.3</td>
<td>(part 2) The deliver behaviour input file.</td>
<td>263</td>
</tr>
<tr>
<td>8.4</td>
<td>The search behaviour input file.</td>
<td>267</td>
</tr>
<tr>
<td>8.5</td>
<td>The robot assistant behaviour-switch input file.</td>
<td>270</td>
</tr>
</tbody>
</table>
1 Introduction

A single autonomous agent situated in an environment only has to consider the possible non-deterministic outcomes of its actions when determining how it should act. In a system containing multiple independently acting agents, however, there will be situations where the agents will interact with each other. When these interactions occur, agents may be prevented from moving as they had intended due to the actions of the other agents. Alternatively, agent interactions may be necessary in order for a team of agents to achieve a global objective. Therefore, a method for managing the actions of agents is necessary if the agents are to coexist and to function together effectively.

A norm defines what is obligatory, what is permitted and what is prohibited. In the field of multi-agent systems, norms (and the related concept of social laws) have been suggested as a possible mechanism for allowing autonomous agents to coordinate their actions dynamically and so to coexist while achieving their individual goals [ST95]. Norms are local rules applicable to individual agents. By adopting the norms of the system each agent is constrained in the actions that it can perform in certain situations, which jointly allows the agents to be able to achieve a desired global behaviour.

Despite their wide acceptance within the multi-agent systems community and the apparent received opinion on the benefit of using norms [AGNT12], there appears to have been very little consideration of how to implement norm-governed agents. Existing examples may contain only a single agent [GGT10] or discuss general implementation issues rather than demonstrate an actual implementation [VAD11]. In particular, the question of whether norms really do offer an effective means of coordinating the actions of agents seems not to have received much attention. Instead, existing work may assume an affirmative answer in order to investigate higher level considerations of sanctions, violations and agents that can reason about norms [ADL12].

This thesis aims to address these issues, by developing a method for allowing an agent to comply with a set of norms and investigating what can be learnt about using local rules to manage the actions of autonomous agents. We do this by building a simulation test bed, which we use to carry out experiments on implementing norm-governed multi-agent systems.
1.1 Taking a set of norms into account

Craven and Sergot distinguish between two different types of norms that can be found in multi-agent systems: *system norms* and *agent-specific norms* [CS08, Ser08]. A system norm describes the system designer’s view of the permitted or desirable system behaviours. An agent-specific norm constrains or guides an individual agent’s behaviour, specifying how the agent should behave in order to conform to the system norm. Therefore, agent-specific norms must be described in terms of what an agent can perceive in its environment and the actions the agent can carry out.

Agent-specific norms are intended to be taken into account, in some manner, in the agent’s implementation or reasoning process. To this end, we develop a system using local rules that we call *directives*, which implement an agent-specific norm in a manner that the agents are able to ‘understand’. The directives, therefore, allow an agent to act in a manner that complies with the agent-specific norm.

A directive maps conditions that can hold in the agent’s current state to actions for the agent to perform. A set of directives can be used to implement a *behaviour*, which may allow an agent to comply with one or more agent-specific norms. However, in order to implement a norm, a method for using directives to determine the actions of an agent must be developed.

Originally, we considered the use of teleo-reactive (TR) programs [Nil94] to implement directives, as TR programs can be used to implement even very simple agents. However, the perceptions and actions typically used by TR programs are too low-level.

As an example, to operate within a grid environment, it is helpful for an agent to be able to perceive its location and orientation. At the low-level typical of a TR program, the agent does not perceive its position within the grid directly. Instead, the agent is able to perceive only its immediate surroundings and is (often) unable to perceive its orientation. Rules to control such TR agents must therefore be formulated without reference to specific grid locations and directions. Alternatively, additional TR perception rules must be included to allow the agent to determine its current location and orientation based on a record of successful movements from a known initial position.

Formulating directives for agents using TR programs is therefore very low-level and tedious. For simplicity, we moved to using a more high-level approach. We assume that the agents are equipped with sensors that enable them to directly perceive their location and orientation. For example, a GPS sensor and a compass can provide this information to the agent. These sensors also allow an agent to determine additional information about its environment, such as the direction to its goal location.

We also turned to existing work on reasoning using priorities between rules, an
area which has seen renewed interest in recent years. We consider recent work by Hansen on prioritised conditional imperatives [Han08] and Horty’s framework for reasoning using prioritised default logic [Hor07, Hor12]. The methods discussed in these works describe different ways that priorities can be applied to rules, and in particular, consider rules that describe actions. In addition, conditional imperatives can be seen as a natural way of expressing norms. Therefore, our investigation also provided an opportunity to explore whether these methods for reasoning about a prioritised set of rules can be applied to the implementation of norms for agents.

Much of the discussion in these works concerns different strategies for handling priorities, as well as the application of priorities to rules that concern beliefs and facts. However, for implementing directives, we are interested in rules that concern actions. We find that when rules that concern actions are considered, the treatment of the actions in these rules is not precise, leading to inconsistencies in how the actions are interpreted.

When we consider directives, defined under the strict condition of rules that relate states to actions, many of the issues discussed concerning priorities no longer apply. In particular, maintaining a consistent interpretation of actions in directives eliminates many of the difficulties encountered by Hansen and Horty. Therefore, we are able to develop a simple formalism for reasoning using priorities over the directives that are used to define a behaviour.

Merely developing a system for implementing norms is not sufficient however. A method must be used to investigate whether norms function as expected and are effective in producing the intended agent actions. For this purpose, we develop a multi-agent simulation test-bed, called Sinatra, which allows the actions and interactions of agents following directives to be observed. Formal verification methods, such as model checking, are of limited use since these methods presuppose that we know what properties we want to examine. The problem rather is that local rules tend to produce unexpected emergent behaviours.

The Sinatra simulation test bed allows us to simulate, visualise and analyse the actions of agents that are implemented using directives. Sinatra is similar to the Packet-World test bed developed by Weyns et al. [WHH05]. Packet-World is used to investigate different aspects of the implementation of autonomous agents in multi-agent systems, where the agents cooperate to achieve complex tasks. However, while Packet-World has been used by Weyns et al. to explore a variety of coordination strategies for multi-agent systems, the use of norms has not been considered. Sinatra differs from Packet-World in that Sinatra is specifically designed for the investigation of norms in a variety of multi-agent systems and for the analysis of the interactions of agents using these norms.

Late on in our investigations, a further motivation and possible use for Sinatra became apparent. There has been an emergence of autonomous vehicles and other
robotic devices that are used in real world applications. Examples include warehouse robots [KIV, Gra12] and autonomous robot assistants [YDr11]. In general, these systems appear to use a centralised control system. It is difficult to be sure however, as these approaches may provide a competitive advantage to their owners and details of their implementations are very limited. In particular, transparent implementation frameworks, which can be followed by other developers to build the required functionality, are not provided. To address this lack of transparency, norms can be considered as a possible implementation strategy. Therefore, Sinatra is also intended to be used as a simulation test bed for investigating how (physical) autonomous robots might be managed and controlled.

1.2 Using local rules to manage the actions of agents

Using norms, implemented using directives, we are able to investigate the use of local rules to manage the actions of independently acting agents. We are particularly interested in exploring to what extent norms can be used to manage the interactions of agents in multi-agent systems. Sinatra has been designed to ensure that the simulated agents are as general as possible, allowing for a wide range of potential (physical) implementations to be considered.

By simulating agents in Sinatra and observing their actions using a graphical representation of the simulation, we investigate the effectiveness of norms to allow agents to coexist and to function together effectively. We develop directives that are able to manage the interactions of agents and explore a possible strategy for handling situations where they are not. We also develop directives that allow agents to work together to achieve a global objective. From this investigation, we are able to gain some insights into how local rules can be used to manage the actions of agents effectively and when it may be more suitable to consider a different implementation method.

1.3 Structure of the thesis

The thesis is divided into three parts.

In the first part, Chapter 2 and Chapter 3, we present the necessary background concepts and define a method by which a prioritised set of directives can be used to allow an agent to comply with a norm. These directives, implemented as what we call ‘behaviours’, are used to determine the actions of simulated agents within the Sinatra test bed.

In Chapter 2 we review the definitions of norms in multi-agent systems, before describing different implementation strategies for multi-agent systems and for norm-governed agents. We then provide an overview of non-monotonic reasoning tech-
niques and methods for reasoning using priorities.

In Chapter 3 we outline the basic Sinatra and simulated agent properties, before demonstrating the development of directives and the method by which directives can be used by these agents to determine their actions. We then show how this process is automated for use in Sinatra and describe how agents can be implemented to use multiple ‘behaviours’ as appropriate.

In the second part, Chapter 4 to Chapter 7, we explore the extent to which directives can be used to manage the actions of autonomous agents and a possible solution for when directives are unsuccessful.

In Chapter 4 we investigate the use of directives to manage undesirable agent interactions and demonstrate the use of Sinatra to analyse these directives. We find that directives are able to manage undesirable agent interactions successfully in many situations. However, using local rules to determine the actions of independently acting agents, who must rely on only their local perceptions of the environment, can lead to situations where the local rules are unable to achieve the desired global behaviour.

These problem situations, where the directives are ineffective, will always occur due to the local nature of the rules and the global system behaviour that they are trying to bring about. Rather than seeking to design a complex set of directives that anticipate and cover every possible eventuality, we suggest that directives are an effective mechanism for managing the actions of agents in general situations. However, problem situations are to be expected and should be handled separately, as appropriate.

In Chapter 5 we investigate how directives can be used in conjunction with a centralised control mechanism, or controller, to manage undesirable agent interactions not successfully handled by the directives alone. A controller uses a global view of the multi-agent system, or relevant fragment thereof, to issue instructions to the agents. Agents give up part of their autonomy in order to respond to the instructions of a controller. We develop controllers to resolve two types of undesirable agent interactions and demonstrate these controllers using Sinatra.

The controllers implemented in Chapter 5 use a relatively simple fixed heuristic method to resolve the undesirable interactions. In Chapter 6 we investigate whether state-of-the-art planning tools might be useful for implementing a more sophisticated controller. We investigate the ability of the Universal Multi-agent OBDD-based Planner (UMOP) [JV00] to implement a controller and whether the multi-agent and adversarial planning capabilities of UMOP can provide a benefit to the controller implementation. However, we find that restrictions that must be imposed on the planning domain in order for UMOP to generate a plan preclude the use of UMOP in a viable controller implementation.

In Chapter 7 we explore the use of directives to coordinate the actions of au-
tonomous agents in order to achieve a global objective more effectively. Opportunities for directives to coordinate the actions of agents in two domains are demonstrated using Sinatra, where repeated simulations of agents following these directives are used to determine the benefit, if any, of this additional coordination.

In the final part, Chapter 8 and Chapter 9, we draw all of these strands of investigation together in a case study application and conclude the work presented in this thesis.

In Chapter 8 we present a case study application based on mobile robot assistants. We discuss the ability of Sinatra to implement and test the sets of directives developed for this case study and how Sinatra can be used to assist the development of actual physical robot assistants.

Chapter 9 concludes this thesis, where we summarise what has been achieved and discuss directions for future work.

1.4 Contributions

In this thesis, we make the following contributions.

- We define an implementation method for allowing an agent to take a set of norms into account when determining its actions, based on prioritised sets of directives that together form a ‘behaviour’.

- We explore existing work on reasoning using priorities and how it is applied to rules that contain actions. However, this work addresses a range of issues that do not arise for prioritised directives, allowing a simpler reasoning formalism to be defined.

- We implement a simulation test bed for simulating, visualising and analysing the actions of agents that are implemented using directives.

- We provide a set of illustrative examples that can be used to test different ‘behaviour’ implementations.

- Experiments show that local rules, in the form of prioritised directives, can be used to manage agent interactions but that there will be some situations where the local rules are unable to achieve the desired global behaviour, due to the limited local perceptions of the agents.

- We demonstrate how a centralised control mechanism, which has a global view of the interaction, can be used in conjunction with directives to manage agent interactions where the directives alone are unsuccessful.
• We find that UMOP is unsuitable for implementing the resolution mechanism of a controller. However, existing tools such as planning frameworks can potentially be used as part of the implementation of a controller.

• We demonstrate that directives can be used to coordinate autonomous agents by regulating the actions of agents so that they become more predictable.

• We describe how Sinatra and directives developed using Sinatra can be used to guide the implementation of norm-governed agents in a physical domain.

It is not our intention to attempt to define any quantitative method for determining the effectiveness of a set of norms. Instead, we address the issue of whether norms really do offer an effective means of coordinating the actions of autonomous agents.

To this end, we find that norms provide a general method for allowing autonomous agents to coordinate their actions but are unable to manage all agent interactions successfully. This result is to be expected when using local rules to attempt to bring about a global behaviour. Therefore, to manage all agent interactions, the use of norms should be considered in conjunction with additional coordination mechanisms that possess a global view of the interaction.
2 Background

During this thesis we first develop an implementation strategy for allowing autonomous agents to take a set of norms into account. We then investigate how norms implemented in this manner can be used to manage the actions of agents in a multi-agent system. In this chapter, we discuss the main areas of existing work related to these implementation challenges.

The existing body of work on norms is very extensive and so we focus our discussion on norms in multi-agent systems. We then review strategies for implementing multi-agent systems and for implementing norms. As described in the introduction, we are also interested in investigating how recent work on conditional imperatives and reasoning with priorities can be applied to implementing norm-governed agents. We finish by outlining some of these approaches. There will be some consideration of planning tools during this thesis, but the background for this will be given in the appropriate chapter.

2.1 Norms in multi-agent systems

The concept of norms has a very long tradition in a social, legal and philosophical setting. Norms specifically for agents in a multi-agent systems were first introduced by Shoham and Tennenholtz as they describe the development of social laws within a multi-robot system [ST95]. These social laws are adopted by each robot in the system and constrain how the robots move around the environment, with the intention of preventing collisions between the robots.

At about the same time Jones and Sergot describe how, at the appropriate level of abstraction, law, computer systems and many other kinds of organisational structure may be viewed as instances of normative systems [JS93]. A normative system refers to any set of interacting agents whose behaviour can be regarded as being governed by norms, where the norms specify the obligations and permissions that apply to the agents’ behaviour. (This, though, does not mean that norms should be used explicitly as an engineering device when constructing such systems.)

Since this time, norms have been adopted in the field of multi-agent systems and have been suggested as a possible mechanism for controlling the actions of autonomous agents in a variety of applications. Example domains include: coordination [SA07], security [ZSM07], electronic commerce [VGG+12] and organising
institutions [AADV10].

This range of applications, however, has resulted in a number of varied concepts and definitions being attributed to norms. Efforts are being made to unify the study of norms in multi-agent systems, both in terms of definitions and scope for future research [AGNT12]. Nevertheless, the requirement remains for the type of norm that will be used in this thesis to be clearly outlined.

For this work, a norm defines what is obligatory, what is prohibited and what is permitted. A norm is a local rule, imposing constraints on the behaviour of the individual agents. In this way, norms are used to influence the actions of autonomous agents in order to regulate the behaviour of the multi-agent system as a whole.

This concept of a norm conforms with the description used by Ågotnes et al., which appears to define the simplest common form of norms in multi-agent systems. They define a norm as


However, the definitions and features that are applied to norms in multi-agent systems are seen to vary as focus shifts to different aspects of their use.

There is growing interest in combining the study of norm-governed multi-agent systems and game theory. Norms can be used in a game theoretic setting to provide a payoff that incentivises agents to act in a certain manner. Grossi et al. suggest that

“[n]orms can also be seen as one of the possible incentives to motivate agents […] The fact that norms can be used as a mechanism to obtain desirable system behaviour, i.e., that norms can be used as incentives for agents, implies that in some circumstances economic incentives are not sufficient to obtain such behaviour” [GGT10] p.201.

Failing to comply with the behaviour stipulated by a norm does not automatically imply that some form of penalty will be exacted. However, methods for detecting violations and for issuing sanctions are often seen as integral to norms and so penalties and rewards may be included as part of the norm definition. Such a system is proposed by López y López et al.

“Norms may include rewards to be given when normative goals become satisfied, or punishments to be applied when they are not. Both rewards and punishments are the means for addressee agents to determine what might happen whatever decision they take regarding norms” [LLd06] p.231.
Norms can also provide a mechanism to facilitate cooperation and coordination between agents, focussing on the global objectives that this can achieve. Grossi et al. show this shift in emphasis by describing how norms can be used to regulate a system of agents, rather than just their individual actions.

“Norms are not usually addressed to individual agents, but rather they are addressed to roles played by agents. In this way, norms from a mechanism to obtain the behaviour of agents, also become a mechanism to create the organizational structure of multi-agent systems. The aim of an organizational structure is to coordinate the behaviour of agents so to perform complex tasks which cannot be done by individual agents” [GGT10] p.202.

A multi-agent system that uses norms to regulate the behaviour of agents is referred to as a norm-governed system or a normative system, although the distinction between these two terms is unclear. Norms for a norm-governed multi-agent system may be developed offline by the system designer or may be allowed to emerge from within the system due to the actions of the agents. For both strategies, the norms that are developed must balance the personal goals and freedom of the individual agents with the global objectives of the society.

2.1.1 Related concepts

A number of concepts related to norms exist in the literature, which describe specific aspects of norms or similar constructs for managing agent behaviour. These include social laws, conventions, policies and normative positions. In all of these cases, however, the definitions used are not always consistent and the distinction between these concepts and norms is often unclear.

Social laws are described by Shoham and Tennenholtz as a global coordination mechanism that restricts the activity of agents in order to allow them to be able to achieve their goals while not interfering with other agents.

“The society will adopt a set of laws; each programmer will obey these laws, and will be able to assume that all others will as well. These laws will on the one hand constrain the plans available to the programmer […], but on the other hand will guarantee certain behaviours on the part of other agents” [ST95] p.234.

For both norms and social laws, however, there is the potential for confusion over terminology. These words are applied in other contexts, where they are given a similar but different meaning to that used in multi-agent systems. For example, norms in human societies are often referred to as conventions that guide human behaviour. Sen and Airiau describe this process.
“Conformity to norms reduces social frictions, relieves cognitive load on humans, and facilitates coordination” [SA07] p.1507.

Conventions are expected patterns of behaviour for those participating in interactions, which are developed by learning from previous experiences. A convention makes it easier for the participants to coordinate their actions, specifically when multiple behavioural choices could have originally been selected.

Norms as conventions are, therefore, similar to norms that are studied in multi-agent systems, but lack any reference to the obligatory, permitted and prohibited acts of the agents. The following definition by Wooldridge, in terms of expected patterns of behaviour, refers to a different sense of the words norm and social law to that typically employed in multi-agent systems.

“A norm is simply an established, expected pattern of behaviour; the term social law carries essentially the same meaning, but it is usually implied that social laws carry with them some authority” [Woo02] p.213.

Policies are used in many of the same contexts as norms. For example Aphale et al. describe policies in terms that can easily be applied to norms.

“Policies guide and regulate behaviour of various entities in a system. They are system-level constraints that are independent from the implementation of specific agents and represent the ideals of behaviour of these agents” [ANS¹²] p.1.

While the two terms sometimes appear to be used interchangeably, policies and norms are also considered to refer to different but related concepts. For example, Craven describes policies as a mechanism for allowing agents to comply with norms.

“Norms are here [Cra11] understood to classify different behaviours and combinations of the properties of a system as good or bad. Policies, by contrast, concern the micro-management of the specific actions performed by the devices, agents and components of a system; the policies are intended to achieve, at a practical level, what the more abstract norms recommend” [Cra11] p.2.

Vasconcelos et al. use the term ‘normative positions’ to describe the active behavioural constraints applying to the agent at the current time. While norms include a condition specifying when an agent is obliged, permitted or prohibited to act in a certain way, Vasconcelos et al. use ‘normative positions’ to consider only the effects of norms that are currently triggered. Therefore, ‘normative positions’ in this context are an...
“[e]xplicit representation of the prohibitions, permissions and obligations associated with software agents [...] Agents’ normative positions change as agents act and interact in pursuit of their goals” [VGG+12] p.5990.

The origins of this usage of the term are unclear. The term *normative positions* was introduced in works by Kanger, Lindahl and others (see e.g. [Lin77]) to attempt to formalise legal notions such as duty, right, privilege and immunity. It is not clear whether Vasconcelos et al. are using the term in this sense.

Jones and Sergot use normative positions to describe the range of possible normative relationships that can exist between two entities [JS92]. Formalised using modal logics, only one normative position will be considered to hold in practice, based on a set of norms or other legislation. The normative positions facilitate the specification and analysis of the rules that are being modelled, allowing the obligations of the agents to be described accurately.

2.1.2 Areas of study

We can divide the study of norm-governed multi-agent systems into three areas: specification, implementation and verification.

The specification of a norm-governed system is the formal statement of the behaviour of the system, including the actions of the agents and the set of norms that apply to them. An example of a language for specifying norm-governed systems is the action language nC+ [SC06].

The implementation of norms involves the production of protocols or programs to allow the agents to determine their actions, whilst taking into account a set of norms. This can also include strategies for allowing the emergence of norms within a system and for allowing their enforcement by the society. The implementation of norm-governed systems is the focus of this thesis.

Verification of a norm-governed system is carried out using techniques such as model checking to determine properties of the system, particularly in terms of norm compliance. Examples of possible verifications tools include the logic NC-CTL [AHW10] and associated model checker NORMC [KPA12]; and the multi-agent system model checker MCMAS [LQR09].

2.2 Implementing multi-agent systems

It is an aim of Sinatra that the simulated multi-agent systems that are developed are able to represent physical multi-agent systems populated by autonomous mobile robots. In this section we present a selection of existing multi-agent system implementations for comparison. We begin by describing tools that are used to investigate the implementation of simulated multi-agent systems. We then discuss strategies for
implementing physical multi-agent systems consisting of multiple robots, in particular those used in industry.

### 2.2.1 Implementation tools for simulated multi-agent systems

Sinatra is intended to allow the implementation of norm-governed multi-agent systems to be investigated. Other tools have been developed to investigate aspects of simulated multi-agent systems. Like Sinatra, these tools are test beds that allow different implementation techniques and their effects to be analysed using a representation of a multi-agent system.

First, we describe the Packet-World test bed, which has the greatest similarity to the Sinatra test bed. Although Packet-World does not consider the implementation of norms, the simulation domain and issues considered using Packet-World have helped to inspire the development of Sinatra. In particular, Packet-World aims to represent a multi-agent system containing autonomous mobile robots. We then describe two test beds for investigating properties of norm-governed multi-agent systems. Unlike Sinatra, however, these test beds focus on the specification of the norms involved and the properties of the system that they will bring about. Therefore, these test beds do not include a visualisation of the system.

#### Packet-World

Weyns et al. have developed a test bed, called Packet-World, for investigating *situated multi-agent systems* [WHH05]. A situated multi-agent system is an environment populated by agents that cooperate to achieve complex problems. The agents perceive the environment around them, maintain their own state and determine their own actions. Packet-World (available from the SourceForge website [Wey09b]) is implemented in Java and has been used by Weyns et al. to investigate and evaluate different implementation strategies and issues related to situated multi-agent systems.

Packet-World consists of a grid environment containing coloured destination locations. Scattered throughout the grid are coloured packets. Agents in the environment collect the packets and deliver them to the correspondingly coloured destination. Packet-World has been used to represent a real-world industrial automated warehouse transportation system, populated by automatic guided vehicles (AGVs) [WHH05].

The agents can move into an unoccupied adjacent cell, pick up packets from adjacent cells and drop off packets into unoccupied adjacent cells or the correct destination location. In addition, each agent has a battery, from which energy is consumed whenever the agent performs an action. The agent’s battery can be recharged if the agent is next to a charger, which occupies a grid cell somewhere
in the environment. The agents must ensure that their batteries are kept charged whilst they work to deliver all of the packets. Packet-World agents have only a limited perception of their environment, being usually the distance of two grid cells from their current location in all directions. Therefore, the agents have to search their environment for packets, destinations and chargers [WHH05].

The Packet-World test bed has been used by Weyns et al. to investigate coordination issues within multi-agent systems. In particular, an agent communication protocol is developed in order to set up cooperation between agents. This allows agents to coordinate their actions; for example, to form chains of agents that can more efficiently transport a cluster of packets to the appropriate destination. Methods for indirect communication and coordination of agents are also investigated, being the use of flags, gradient fields and pheromones [WHH05].

Further work by Weyns et al. shows their continued investigation of implementation issues and strategies for realising decentralised control of an automated warehouse transportation system using autonomous agents in a multi-agent system (e.g. [Wey09a]). However, in order to implement decentralised control for autonomous agents, an architectural design solution is presented rather than consideration of the use of norms. To manage agent interactions Weyns et al. define the environment to be an active entity within the multi-agent system [WOO07]. Laws ‘embedded’ in the environment are used to control the activities of agents, ensuring that agent perception, action and communication corresponds to the requirements of the system [WH07].

The implementation of norm-governed agents in Packet-World does not appear to have received much attention. One notable exception is the work of Urovi et al. who have implemented a software framework for computing at runtime the physically possible and permitted actions of agents, as well as any sanctions that should be applied [UBSA10]. This framework is evaluated using a norm-governed version of Packet-World called Open Packet-World. However, the framework is concerned with informing the decision making of agents rather than considering ways of implementing agents that comply with the norms.

**Socio-economic test bed**

A test bed for experimenting with the implementation of norm-governed multi-agent systems has been developed by Pitt et al. [PSA12]. The systems investigated are open electronic institutions of autonomous agents, where the agents share access to a constrained pool of resources. Within each system, a community of agents is established who have the ability to specify how the resources should be allocated. The aim is to determine the properties required for the system to be *enduring*, meaning that the resources are managed in a sustainable manner rather than attempting to achieve an optimal allocation.
The resources that exist within the system are replenished at varying rates during the lifetime of the system. All agents have the ability to appropriate resources, although only agents who are members of the community have permission to take these resources, up to their specific allocation. Agents may, with certain probabilities, violate the norms of the system by taking more than their allocated quantity of a shared resource [PSA12].

Agents take on roles in the community to allow the agents to be self-governing. Agents in different roles can exercise different powers over the other agents in the system, including the ability to exclude agents from being able to appropriate resources, to allocate resources to other agents and to report agents that take more than their allocation [PSA12].

Pitt et al. use the socio-economic principles of Elinor Ostrom to specify potentially desirable properties of the system. Six of these principles are implemented in the test bed. Repeated simulations are carried out to identify the requirements for an enduring system under different environmental constraints, including varying rates of intentional and unintentional violations on the part of the agents. Pitt et al. demonstrate that implementing these six principles in a self-organised system can improve the endurance of the institution when managing a constrained resource [PSA12].

The experiments of Pitt et al., however, also show that these six principles are not always able to ensure the endurance of the electronic institution. The open nature of the system, the process of self-government and non-compliance of the agents mean that resources in the system may still become depleted despite the socio-economic principles that have been implemented. In addition, the implementation of the principles needs to be tuned to the specific properties of the system, such as the level of non-compliance amongst the agents. Additional mechanisms are required to allow the agents to learn how the system may evolve when different levels of resources are available and also how to respond to these changes appropriately [PSA12].

Executable specification of norm-governed systems

Artikis describes a system for specifying the operation of a norm-governed multi-agent system using the C+ action language or the event calculus (EC) [Art03]. The systems described are open agent systems containing a society of agents, where the actions of the agents are governed by a set of norms. Written as a C+ action description or as an EC action description, the specification is also executable, meaning that it can be used to determine which specific obligations and privileges pertain to any given agent in a particular circumstance. The specification is also used, in some cases, to prove certain general properties of the system as a whole.

The C+ and EC action descriptions contain the social constraints for the system, which restrict the actions of the agents in the society. The social constraints define
the possible valid actions of the agents, as well as the permitted, obligatory and empowered actions of the agents according to their social roles [Art03].

A C+ or EC action description defines a transition system where the states of the system correspond to the results of the agents’ actions. Software tools are used to investigate the execution of these action descriptions by querying the outcome of the system after a specified series of events have taken place. The result of this query is a state of the transition system describing the powers, obligations and sanctions that are associated with the agents at this time [Art03].

The executable specifications using C+ and EC are demonstrated for a variation of the contract-net protocol and an argumentation protocol. In each case the associated software tools are used to animate and to validate the specifications in response to different queries, allowing the user to identify the empowered, permitted and obligated actions of the agents [Art03].

2.2.2 Real world multi-agent systems

Real world systems involving multiple robots operating in a shared environment are becoming more commonplace, particularly in an industrial setting. These systems must resolve many of the same issues discussed in the context of multi-agent systems, such as how to manage the interactions of the robots and how to allow the robots to work together effectively. However, it does not appear that there are any real world industrial examples where norms are used to manage the interactions of robots.

Instead of norms, which attempt to manage the interactions of agents in a decentralised manner, a centralised approach is normally used. Decentralised control of autonomous robots generally allows the system to be more scalable and flexible than a system using centralised control. However, without the ability to control all robots simultaneously, a system using decentralised control can fail to solve tasks that a centralised system would be able to handle. A centralised control system appears to be preferred in many industrial situations because it can be more reliable, despite the risks of having a potential single point of failure and a possible bottleneck in communications.

For example, Sánchez and Latombe show that centralised planning in a multi-robot system can be significantly more reliable than when plans are developed for each robot individually, as would be required for a decentralised control system [SL02]. For manipulator robots in a spot-welding station, a centralised planning system was able to find a solution to take each robot from its initial state to its goal state without colliding with any other robots or obstacles. Using a decentralised approach, however, the system was unable to find a non-colliding set of plans for the robots 30% to 75% of the time.

Warehouse robots are a common example of mobile robots operating under a centralised control system. A central stock system receives orders and directs robots
to collect the required items from the warehouse, before delivering these to a human for checking and processing. An example of such a centralised warehouse system is developed by KIVA Systems, a manufacturer of ‘mobile-robotic fulfilment systems’, which is now a subsidiary of Amazon.com [KIV]. Similar systems are used by Net-A-Porter, Asda and Ocado [Gra12].

Although not using norms, some examples of decentralised control of robots in industry do exist. In these examples swarm robotics is used to coordinate the actions of autonomous robots. Swarm robotics allows a team of relatively simple robots to work together by utilising basic communication methods. The communication between the robots builds a system of constant feedback, allowing a desired collective behaviour to emerge from the interactions of the robots.

The Fraunhofer Institute is investigating the potential for swarm robotics to implement warehouse robots [Fra12]. An example of swarm robotics already in use are SIGA robots – Santander Interactive Guest Assistants – used at the Santander Banking Group’s visitor centre. The SIGA robots are autonomous robots that are able to guide a visitor to a pre-selected destination, whilst avoiding obstacles in their path. The designers, YDreams, say that the SIGA robots are one of the first applications of swarm robotics to a commercial setting [YDr11].

2.3 Implementing norms

A number of different techniques for implementing norm-governed agents or norm-governed multi-agent systems have been proposed. Techniques for norm implementation can be divided into two branches, which we will refer to as norm emergence and norm adherence. Norm emergence is the process by which norms are created or discovered by a group of agents. Norm adherence is the process by which agents interpret, reason about and comply with a set of norms that already exist.

In this section, an overview of existing implementation strategies for both norm emergence and norm adherence are presented. As this thesis is concerned with the implementation of a set of existing norms, however, we focus our discussion on strategies for norm adherence and present only a sample of norm emergence techniques.

2.3.1 Norm emergence

Norm emergence concerns the formation of norms by agents and the spread of norms amongst the population of a multi-agent system. Agents that are capable of developing and spreading norms require learning and reasoning abilities that are not necessarily required by agents that are designed to follow an existing set of norms.

As an example, Sen and Airiau demonstrate how the mutual benefits that exist for agents who coordinate their actions can be the catalyst for the development of
norms [SA07]. Using a stage game, a payoff matrix and agents implemented to make use of a learning algorithm, repeated interactions amongst a group of agents are shown to lead to the development of ‘rules of the road’.

Riveret et al. describe an alternative method for norm emergence, which also leads to the development of rules of the road [RRS12]. Probabilistic argumentation rules are used by a system of agents, in conjunction with a reinforcement learning algorithm, to allow the emergence of a convention to drive on the left or the right side of the road. Sanctions are used by agents to punish each other for actions that lead to undesirable states and for violations of obligations, which allow the norms of the system to develop.

Having developed a set of (possibly contradictory) norms, different strategies can be used to allow a (consistent) set of norms to spread to other agents in the system. In one approach, Lotzmann describes the spread of norms as the “two-way dynamic” between *immergence* and *emergence* [Lot10].

“[Immergence is an] intra-agent process by means of which a normative belief is formed into the agents’ minds. [Emergence is an] inter-agent process by means of which a norm not deliberately issued spreads through a society” [Lot10] p.67.

Immergence allows an agent to identify what the norms of the society might be. In contrast, emergence allows the agents in the society to create and enforce norms by evaluating the actions of others.

These processes are applied in a simulation example of agents writing and modifying a Wikipedia of articles. Agents are initially implemented to follow one of three writing styles and to issue ‘blame’ to other agents for articles that violate this style. Over the course of the simulation, agents are seen to switch styles such that an approved writing style emerges for the group [Lot10].

Savarimuthu et al. describe an alternative method for allowing agents to infer the norms of a multi-agent system, by utilising data-mining techniques [SCPP10]. An algorithm is proposed, which allows an agent observing the actions of other agents to recognise norms that exist in the multi-agent system. An observing agent records the order of actions that agents perform and when sanctions are imposed on these agents. Sanctions occur when an agent fails to perform a required action, allowing an observing agent to infer the actions obligated by the norm. The algorithm is demonstrated using the example of a restaurant simulation. Agents observing the actions and sanctions of the other agents are able to identify the ‘norm of tipping’.

### 2.3.2 Norm adherence

Having developed a set of norms, whether through norm emergence or norms created by the system designer, agents must take these norms into account when determining...
their actions. Norm adherence concerns the representation of norms within an agent model, strategies for allowing agents to understand and to comply with these norms, and examines mechanisms that can be employed when norm violations occur. In the rest of this section, we describe different methods and implementation guidelines that have been proposed for norm adherence.

A range of strategies for implementing agents to take a set of existing norms into account have been suggested. When strategies for norm adherence are suggested, however, they appear to assume that an agent already knows how to act in order to comply with the norm, or at least how to act in order to achieve its goals. In these cases the issue of norm adherence becomes the implementation of constraints, monitoring systems, sanctions or some other mechanism to manipulate, force or persuade an agent to comply with a set of norms.

We begin by reporting an overview of different norm representation techniques and the different types of norms that can be applied to agents, as an introduction to the range of approaches that exist. We then describe the implementation of norms in BDI agents, the agent model most commonly associated with norm-governed agents. We consider strategies for allowing an agent to comply with a set of norms and how the distributed control of agents can be enforced. Finally, we describe techniques for handling norm violation and for norm conflict resolution.

Throughout the work presented below, the issue of implementing norms that can effectively manage the actions of agents does not appear to be considered. We find that the example norms and norm-governed systems are often very abstract or assume a suitably intelligent agent that is already able to ‘understand’ how to comply with its norms. Multi-agent systems that are defined in terms of norms do exist, for example in electronic commerce applications [DKS02]; however, these systems are built around and constrained by a set of norms. The question of how to introduce a set of norms to an existing multi-agent system in order to manage the agents’ actions and to bring about some form of coordination amongst these agents does not appear to be addressed.

2.3.3 Representation of norms

A comparison study of norms is presented by Neumann [Neu12]. In this comparison, a study of norms and how they are represented in agent architectures is carried out, drawing on 14 examples from different approaches to norms. Based on these architectures, key components and possible strategies for norm implementation are suggested, by which agents may be implemented to recognise norms and to incorporate norms in their reasoning processes. An overview of the range of possible approaches highlighted by Neumann is given below.
• **Concepts of norms.** The first component identified for distinguishing different agent architectures is how norms are conceived. Norms can be thought of as *constraints* on the behaviour of individual agents; as *obligations* that are conditional on specific circumstances; or as *abstract concepts* that assume conflicts may occur between obligations and are used by agents to reason about their actions in concrete situations.

• **Norm dynamics.** Norms may change over time, although in most architectures norms are static rules implemented by the system designer. Where norm dynamics are considered, architectures may allow *norm spreading* between agents. Neumann also identifies *norm innovation* as a potential norm dynamic, where new norms are added to the system by the system designer or the agents themselves. None of the sample agent architectures considered in Neumann’s survey include this type of norm dynamic, though the work of Sen and Airiua [SA07] and Riveret et al. [RRS12] demonstrate norm innovation in practice.

• **Social representation.** Norms are a possible technique for managing agent interactions, for which multiple agents are required by necessity. However, norms also regulate individual behaviour and so can equally be investigated in the context of a single agent. Therefore, agent architectures can have an explicit population of multiple agents or be designed for a single agent. For architectures that focus on a single agent, a society of agents can be represented in an indirect manner using a *mental representation* within the architecture of a single agent.

• **Norm conflicts.** In some architectures, contradictions between the rules given to the agents lead to errors. However, the potential for conflicts to arise due to norms is a natural result of rules that influence agent actions when not all actions can be executed concurrently. Different architectures identify the potential for conflicts to occur from different sources. Conflicts can occur *between the agents* during the process of norm innovation; conflicts can occur *between the norms and the individual goals* of an agent; and conflicts can occur *between the norms* themselves, especially when norms derive from different normative authorities.

The comparison survey by Neumann highlights that there is a range of possible approaches to studying norms and that research on norms can fall into a number of different areas. The examples that are presented in the rest of this section go some way to demonstrate this breadth of research. Nevertheless, the categories described above allow similarities to be identified between different approaches in terms of the definition of norms and how these norms are represented.
López y López et al. also tackle this issue when developing what they propose as a canonical model of norms [LLd06]. This model identifies possible components of a norm, allowing a variety of norms to be defined in terms of this model. The norm model is then used as part of the model of a normative agent and a normative multi-agent system, forming a framework for describing norm-governed multi-agent systems. This framework is described in general terms and does not appear to have been applied to an example system.

The model of norms proposed by López y López et al. consists of seven elements. A norm must contain a non-empty set of normative goals, which are directed at a non-empty set of addressee agents. A norm may also include a set of agents that benefit from these goals being achieved. The norm must be defined such that it is applicable within a specific context, although circumstances where the norm is not applicable may be defined as exception states. Finally, a norm may specify rewards and punishments to be given depending on whether a normative goal is achieved [LLd06].

2.3.4 Types of norms

While potentially falling into the canonical model of norms described by López y López et al., the norms that can be applied to an agent may vary significantly in terms of their definition. Different types of norms can be identified, depending on whether the norms are defined in terms of general system behaviours or concrete agent actions.

Craven and Sergot distinguish between two different types of norms that can be found in multi-agent systems: system norms and agent-specific norms [CS08, Ser08]. A system norm describes the system designer’s view of the permitted or desirable system behaviours. An agent-specific norm constrains or guides an individual agent’s behaviour. Agent-specific norms are intended to be taken into account, in some manner, in the agent’s implementation or reasoning process. Therefore, agent-specific norms must be described in terms of what an agent can perceive in its environment and the actions the agent can carry out.

As an example, consider the following traffic law designed to specify in what circumstances agents should give way to each other. This traffic law will be used as part of the implementation of norm-governed agents in Section 4.3.3.

Agents travelling north or west take priority when agents interact. Therefore, agents travelling south or east must give way to agents travelling north or west.

The first sentence is the system norm, describing the behaviour desired by the system-designer. The second sentence is the agent-specific norm, describing the
desired behaviour in terms of the perceptions and actions of the individual agents. The directives that are developed as part of this thesis are a possible method for allowing an agent to take into account an agent-specific norm.

System norms and agent-specific norms are defined by Craven and Sergot using a green and red colouring over a transition system. System norms are represented by colouring states and transitions as green or red, depending on whether or not they are permitted or desirable. Agent-specific norms are represented by colouring the action performed by an agent in each transition, forming a coloured strand of the transition system for each particular agent. The colouring of the agent’s actions as green or red describes whether the agent has complied with its individual norms. System norms are used to govern system behaviour as a whole, whereas agent-specific norms are used to govern an agent’s individual actions [CS08].

A potentially similar distinction between types of norms is highlighted by Aldewereld et al., who differentiate between norms defined in an abstract manner in an organisational specification and the concrete instantiation required for the norms to be applied in practice [AADV10]. Norms are parsed from an abstract general specification into a set of abstract if-then rules. Aldewereld et al. describe the use of counts-as relations to define properties of actions and states within a specific context. Using these counts-as relations, agents are able to reason about norm compliance by grounding the abstract norms within the current context.

The example scenario presented by Aldewereld et al. concerns crisis management. A general norm is used to ensure coordination between superiors and crisis handlers in the event of an emergency.

“Crisis handlers need to inform their superiors through adequate measures.”

Depending on the actual event taking place, such as a car crash or a major flooding incident, the norm is grounded to the appropriate context. Therefore, the agents that are able to act in the roles of crisis handlers and superiors, as well as what constitutes an appropriate channel of communication between these two, depends on the counts-as relations for the current context [AADV10].

### 2.3.5 Norm-aware BDI agents

The agent models most frequently associated with norms are based on the BDI (belief-desire-intention) model. A BDI agent maintains a set of beliefs about the world, a set of desires that describe the motivations and goals of the agent and a set of intentions that represent what the agent has chosen to do. BDI agents fulfil their intentions by executing plans that are stored in a plan library. The BDI architecture allows agents to reason about their choice of plan by balancing requirements based on their beliefs, desires and plans that the agent is currently executing.
The BDI agent model can be extended in different ways in order to allow the agents to take into account norms in their reasoning process.

Dignum et al. distinguish between obligations, norms, desires and goals, leading to an extended BDI architecture that explicitly represents these motivations [DKS02]. Desires describe the agent’s own interests, obligations describe the interests of other agents and norms describe the interests of the society. Dignum et al. argue that candidate goals can arise from any of these motivations. Once selected, a candidate goal becomes an intention, where the agent generates and executes a plan to achieve this goal.

The model is demonstrated in an electronic commerce setting, where an agent finds itself in the situation of having obligations to buy a book from three vendors but without sufficient funds to pay more than one of them. Priorities between the agent’s desires, obligations and norms of the society are used as part of the agent’s reasoning process in order to resolve its conflicting obligations, leading to a candidate goal [DKS02].

Meneguzzi and Luck describe a BDI extension that allows agents to generate high-level plans that adapt the agent’s behaviour in response to norms adopted during runtime [ML09]. Plans are generated to direct the agents to adopt new goals in response to obligations and to suppress intentions in response to prohibitions.

In more recent work, Meneguzzi, Luck et al. propose an alternative BDI extension that allows agents to customise existing plans in order to ensure the agents comply with the norms [MVOL12]. Active norms are used to annotate the agent’s existing plans with additional constraints, based on the conditions that are obligatory or prohibited according to the norms. In this way, specific plan instances that comply with the norms can be executed, without the need for all potentially violating plans to be suppressed. Norm violation can also occur in a controlled manner, when goal achievement outweighs the utility of norm compliance, by making explicit the norms violated by each step of a plan.

BDI agents and extensions to allow the inclusion of norms can be implemented in a number of systems. One example is the agent-orientated programming language AgentSpeak [Rao96], used by Meneguzzi and Luck to implement the high-level plan algorithms described above [ML09]. Another system is the combined language and execution platform 2APL [Das08]. Alechina et al. have developed an extension to 2APL to create the norm-aware programming language N-2APL [ADL12]. N-2APL provides support for introducing normative concepts of obligations, prohibitions, sanctions and deadlines to an existing BDI agent.

An N-2APL agent uses a scheduling algorithm described by Alechina et al. to reason about plans to achieve its current goals and obligations. The scheduling algorithm directs the N-2APL agent to adopt a preference-maximal set of plans, which will contain the greatest number of highest priority plans that can be achieved at
the current time. The preference-maximal set of plans allows the agent to maximise the priority of the goals and obligations that it can achieve, whilst ensuring that the agent does not violate a prohibition of the same or higher priority [ADL12]. While the syntax and semantics of N-2APL are presented by Alechina et al., there does not appear to be an example of its use at this time.

The BDI agent model is often used to define agents that are able to reason about norms. However, the BDI agent model focusses on mechanisms for plan choice and plan maintenance from an existing set of plans. The generation of these plans and, in particular, how to generate an initial plan that complies with a norm is not within the scope of the model.

2.3.6 Strategies for norm compliance

Specific suggestions for implementing norm compliance are discussed by Grossi et al. [GGT10], who outline different ways to implement norms in a multi-agent system and how these can be specified in a game-theoretic framework. Grossi et al. are concerned with finding a possible solution to the norm implementation problem, which they define as

“How to make agents comply with a set of norms in a system?” [GGT10] p.196.

A labelled transition system model is used to represent the different states resulting from the possible actions of an agent. Norms are considered as labelling transitions to legal and illegal states and are expressed using deontic logic. Norm implementation is discussed using regimentation and the concept of retarded preconditions. The use of enforcement via sanctions to ensure that agents comply with the norms is also considered [GGT10].

Regimentation, a term originally coined by Jones and Sergot [JS93], involves modifying the transition system model so that transitions that would lead to an illegal state no longer appear in the model. Therefore, it is impossible for an agent in the system to execute an action that is not permitted by the norms [GGT10].

Retarded preconditions control the execution of non-deterministic actions where some of the possible outcomes are illegal states but not others. Using a retarded precondition, the execution of a given action is only possible if it results in a certain specified effect.

“In short, actions are allowed to be executed under circumstances which can possibly lead to violations, but only if the effects are still acceptable. If they are not, then nothing has happened” [GGT10] p.214.

In the case that an illegal state is reached the action is rolled-back as if the action had never been executed. The implementation of a retarded precondition is described
using a blocks world example, where blocks may only be moved if the resulting towers fulfill specified requirements. How this process would be carried out in practice is unclear.

Enforcement via sanctions uses payoffs to influence the rational choices of the agents so that they do not violate the norms, while still maintaining the full range of actions available to the agents. The implementation of enforcer agents within the system is considered, who monitor the actions of other agents and reduce payoffs when violations occur. However, Grossi et al. observe that the implementation of norms via an enforcer agent results in the need for more norms to be implemented to control the enforcer agent itself. They suggest that only full regimentation or an automatic enforcement mechanism will resolve this issue [GGT10].

More specific suggestions for norm compliance are presented by Vanhée et al., as part of a high-level overview of issues to be considered when implementing norms [VAD11]. These issues are tailored towards intelligent BDI agents that can reason about norm adoption and compliance. In the case of norm compliance, possible techniques for modifying an existing set of agent plans to take into account norms are described.

In the first technique, a duplicate plan is created, containing the additional actions prescribed by the norm. This duplicate plan is used instead of the original plan in cases where the norm condition holds. In the second implementation technique, the original agent plan is augmented by conditional statements to execute the norm actions when the norm condition holds. As noted by Vanhée et al., however, both these techniques for norm compliance can quickly lead to a large increase in the number or the size of the agents’ plans [VAD11].

An indirect consequence of norm compliance amongst intelligent agents is also highlighted by Vanhée et al. [VAD11]. Agents may plan their behaviour based on the expected norm abiding behaviour of the other agents in the system. This can lead to better coordination between norm abiding agents, as well as potential gains for agents that intentionally violate a norm.

2.3.7 Distributed control of agents

Outside of the multi-agent system community, in the software engineering community, Minsky and colleagues describe law-governed interaction (LGI), which considers issues that are related to norm compliance and distributed control. LGI is a message-exchange mechanism that allows coordination and control of multi-agent systems. The interactions of the autonomous agents of the system are governed by a global policy, consisting of a number of laws. These laws are applied locally at each agent but with the capacity to bring about global results within the agent community [MU00, ZSM07]. While not explicitly using the terminology of norms, the properties of LGI systems are comparable to norm-governed systems [Min12].
Assume a law \( \mathcal{L} \) that is applicable to all the members of a community, which is referred to by Minsky as an \( \mathcal{L} \)-community.

“Such a common law is necessary [for] the members of an \( \mathcal{L} \)-community to collaborate harmoniously and to compete safely with each other – depending on the law that governs them all – even with no knowledge of the nature and intention of each other. This is analogous to the manner in which social laws can create harmonious societies” [Min12] p.385.

In other words, laws that are complied with by a group of agents are used to manage the agent interactions.

The laws of an LGI policy follow an event-condition-action structure. The event is a \textit{regulated event}, controlled by an LGI law that occurs locally at the agent. The condition refers to the \textit{control-state} of the agent, being the history of its interactions with other LGI agents. The action is a possibly empty set of \textit{primitive operations} that are carried out at the agent as directed by an LGI law.

Law enforcement, or compliance, is carried out by a controller associated with each agent, which maintains the control-state of the agent. Based on the laws of the policy, the controller decides how to react to messages sent and received by its agent. An agent sending or receiving a message that does not comply with the policy will be prevented from doing so by its controller. All controllers must be verified for their correct implementation, but once in place, the controllers will ensure the compliance of all agents to the laws of the policy. A Java implementation of the LGI mechanism, called Moses, is available from the Moses website [SM05].

Another decentralised strategy for norm compliance is presented by Vasconcelos et al. In this instance, norms are considered in terms of ‘normative positions’, which they describe as the specific permissions, prohibitions and obligations related to illocution (speech) acts between agents [VGG+12]. Vasconcelos et al. introduce \textit{normative structures}, being a set of rules that can be used to manage the triggering and expiration of norms whilst agents interact. An algorithm is presented, allowing conflicts between norms to be resolved at runtime by modifying new normative positions that are triggered based on the normative positions that already hold.

Similar to the LGI approach described by Minsky et al., a \textit{governor agent} is associated with each agent. The governor agent ensures that the messages sent by its agent comply with the norms of the system, maintains the agent’s social state and informs the agent about the normative positions that apply to it. However, Vasconcelos et al. distinguish their approach from that of LGI.

“[In LGI] each agent has a local message interface that forwards legal messages according to a set of norms. Since these interfaces are local to each agent, norms can only be expressed in terms of actions of that
agent. This is a serious disadvantage, e.g. when one needs to activate an obligation to one agent due to a certain message of another one\textsuperscript{5} [VGG\textsuperscript{*12}] p.5997.

The governor agent of Vasconcelos et al. is part of a distributed social network through which agents may communicate. Therefore, norms can be triggered based on the actions of another agent in the system.

Potentially, the system of Vasconcelos et al. would allow an agent to receive an obligation to act due to an interaction between agents observed by the governor agent but not involving the agent itself. While the reason for such an obligation would be difficult to justify, it is not possible to express this obligation using LGI. However, it does not appear that LGI prevents norms being defined in terms of messages received by an agent, which would also be in terms of the actions of the sending agent. Therefore, the distinction between the expressiveness of the approaches of Minsky et al. and Vasconcelos et al. remains unclear.

2.3.8 Norm violation mechanisms

As highlighted by Neumann [Neu12], conflicts between norms are a natural result of rules that attempt to influence agent actions. When conflicts occur, whether between different norms or norms and an agent’s individual goals, the result will be that some norms may be violated.

A common solution to norm violation is the introduction of sanctions via an enforcer. As described by Grossi et al. [GGT10], enforcement can be used as a method for norm compliance, by using payoffs to influence the rational behaviour of an agent. Enforcement can also be used to issue sanctions to punish agents that violate norms.

In their implementation overview, Vanhée et al. discuss the use of sanctions as feedback for violating agents [VAD11]. They suggest that sufficiently intelligent agents can learn new norms via such sanctions, as well as cases where norm violation is ‘acceptable’ because no sanction is received. Vanhée et al. also briefly outline the use of a repair action in response to norm violation, which attempts to return the system to a desirable state.

2.3.9 Norm conflict resolution

The issue of what happens when conflicting norms apply in an agent’s current situation is also considered as part of the implementation of norms. Different techniques for conflict resolution are discussed by Neumann [Neu12]. These techniques include: maximisation of expected utility; a priority ordering between norms or other components; levels of abstraction, such as default rules and exceptions; and consideration of the relative importance of different normative authorities. The suitability of each
A technique for resolving a particular conflict may depend on the type of conflicts that are present in the agent architecture.

Aphale et al. describe a strategy for conflict detection and resolution between the active policies or norms that constrain the agent’s actions and the agent’s library of plans for achieving its goal [ANŠ12]. Using intelligent agents that are able to reason about the effects of their plans given the currently applicable policies, a strategy is described for identifying the most relevant conflicts based on the goals of the system. Alphale et al. distinguish between two different types of conflicts: logical conflicts between different policies and functional conflicts between policies and the agents’ goals. Algorithms are proposed for the detection of functional conflicts.

Having identified a conflict using this strategy, different conflict resolution mechanisms are also discussed by Aphale et al. [ANŠ12]. These resolution mechanisms include (for policies as norms): adding a new policy; modifying the policy in some way, in terms of the actions it controls or the activation and expiration conditions of the policy; and prioritising policies.

In the development of their strategies for conflict detection and resolution, however, Aphale et al. identify that processes for the detection and resolution of conflicts are computationally expensive. They suggest that this issue should be taken into account when implementing conflict detection and resolution systems by identifying the most important conflicts to handle.

“[T]he reasoning mechanism must focus on the most relevant conflicts, given the goals of the organisation/agent. Conflicts occurring in a system are of varying significance. Some conflicts have higher chances of occurring and they need to be resolved statically, while others have very rare chances of occurring and they can be resolved at runtime” [ANŠ12] p.7.

2.4 Priorities on rules

When rules are used to direct an agent how to act, the potential for conflicts to occur between these rules is always present. A conflict occurs in this context when an agent is directed to perform two or more incompatible actions. As described above, such conflicts can occur between different norms that apply to an agent, as well as between norms and the rules used to direct an agent towards its goals. Conflicts can also occur between the rules used to allow an agent to take a norm into account, specifically between the directives that are developed as part of this thesis.

Priorities, often also termed ‘preferences’ in the literature, have been considered for some time as a means of resolving situations where conflicting rules can be
applied. In general, priorities provide a ranking on a set of rules and any conflicts that occur are resolved in favour of higher priority rules. However, a priority order by itself is not sufficient for resolving conflicts between rules – a strategy for using the priority order is required as well. We are interested to see how recent work on reasoning using priorities can be applied to the implementation of norms for agents.

In this section, we first describe the main examples of non-monotonic systems used for reasoning with priorities. We then report an overview of different strategies that can be applied to these priorities in order to obtain the most preferred result of the prioritised rules. We outline two such approaches in more detail. These approaches are those that appear to have the most relevance to the implementation of norms for agents, as they consider priorities between rules that contain actions. Finally, we describe a framework for allowing priority information to be compiled into an extended logic program.

2.4.1 Non-monotonic reasoning with priorities

Before we are able to discuss different strategies for reasoning using priorities between rules, we need to outline the non-monotonic systems used for defining these rules. We now report the basic concepts of default logic, extended logic programs and conditional imperatives.

Default logic

A default rule δ is of the form

\[
\frac{\alpha : \beta_1, ..., \beta_n}{\gamma} \quad (n \geq 0)
\]

where \(\alpha, \beta_1, ..., \beta_n\) and \(\gamma\) are formulae [Rei80]. Informally, the default rule can be read as ‘if \(\alpha\) is derivable and \(\beta_1, ..., \beta_n\) are consistent, then derive \(\gamma\)’.

A normal default rule is a rule of the form

\[
\frac{\alpha}{\gamma}
\]

In what follows, we will use the shorthand notation \(\alpha \rightarrow \gamma\) for a normal default rule.

Following others (e.g. Hory [Hor07]), for a normal default rule \(\alpha \rightarrow \gamma\), let \(\text{Premise}(\delta)\) and \(\text{Conclusion}(\delta)\) denote the premise \(\alpha\) and the conclusion \(\gamma\) of the rule. For a set of normal default rules \(D\), \(\text{Conclusion}(D)\) denotes the set of conclusions for these rules.

\[
\text{Conclusion}(D) \overset{\text{def}}{=} \{\text{Conclusion}(\delta) \mid \delta \in D\}
\]
A default theory is a pair \((W, D)\), where \(W\) is a set of formulae and \(D\) is a set of default rules. An extension \(E\) of \((W, D)\) is a set of formulae that represent an acceptable set of ‘beliefs’, given facts \(W\) and default rules \(D\) \cite{rei80}.

An extension \(E\) of \((W, D)\) is the smallest set of formulae that contain \(W\), are closed under classical consequence \(Th\), and are closed under the default rules \(D\) that are applicable given \(E\) \cite{rei80}, defined as follows.

For a set of formulae \(E\), \(Th(E)\) stands for the set of all classical truth-functional consequences of \(E\), being the set of all formulae derivable from \(E\).

\[
Th(E) \overset{\text{def}}{=} \{ \alpha \mid E \vdash \alpha \}
\]

Let \(R\) be a set of (monotonic, non-default) rules of the form \(\frac{\alpha}{\gamma}\). A set of formulae \(E\) is closed under a set of rules \(R\) when \(T_R(E) \subseteq E\), where \(T_R(E)\) is the set of formulae obtained from \(E\) after one application of the rules in \(R\).

\[
T_R(E) \overset{\text{def}}{=} \{ \gamma \mid \alpha \gamma \in R \text{ and } \alpha \in E \}
\]

The closure of formulae \(W\) under set of rules \(R\) is denoted by \(Cn_R(W)\). \(Cn_R(W)\) is the smallest set of formulae \(E\) that contain \(W\), \(W \subseteq E\); is closed under classical consequence \(Th\), \(Th(E) \subseteq E\); and is closed under the rules \(R\), \(T_R(E) \subseteq E\).

For a default theory \((W, D)\) and a set of formulae \(E\), \(D^E\) is the reduct of \(D\) given \(E\), being the set of default rules in \(D\) that are applicable given \(E\).

\[
D^E \overset{\text{def}}{=} \{ \frac{\alpha}{\gamma} \mid \alpha : \beta_1, \ldots, \beta_n \in D, \gamma, \neg \beta_i \notin E, \text{ for every } i \in 1..n \}
\]

Then \(E\) is an extension of \((W, D)\) when \(E = Cn_{D^E}(W)\).

A prioritised default theory is a tuple \((W, D, <)\) consisting of a set of defaults \(D\) ordered by the priority relation \(<\), and a set of formulae \(W\). The priority relation \(<\) is defined as a strict partial order on \(D\). For any \(\delta_1, \delta_2 \in D\), we write \(\delta_1 < \delta_2\) to mean that default rule \(\delta_1\) has a higher priority than \(\delta_2\).

The priority relation \(<\) can also be applied to sets of defaults, for example \(S < \{\delta\}\). Following Horty \cite{hor07}, for a set of defaults \(S\) where \(S \subseteq D\) and \(\delta \in D\), \(S < \{\delta\}\) specifies that there is at least one default \(\delta' \in S\) such that \(\delta' < \delta\).

**Extended logic programs**

For some time, the system of choice for expressing priorities over rules was default logic. However, more recent work has seen a shift to logic programs and extended logic programs in particular \cite{dstw04}.

In an extended logic program, the symbol \(\neg\) is used for classical negation and \textit{not} is used for negation by failure \cite{gl91}. A literal \(L\) is an expression of the form \(A\) or \(\neg A\), where \(A\) is an atom. A literal of the form \textit{not} \(L\), using negation by failure \textit{not}, is said to be a weakly negated literal.
An extended logic program \( \Pi \) is a set of clauses, or rules. A rule \( r \) in an extended logic program is an expression of the form
\[
L_0 \leftarrow L_1, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n \quad (n \geq m \geq 0)
\]
where \( L_0, \ldots, L_n \) are literals\(^1\) [GL91].

Following others (e.g. Delgrande et al. [DST03]), the literal \( L_0 \) is the head of the rule and the set of literals \( \{L_1, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n\} \) form the body, denoted by \( \text{head}(r) \) and \( \text{body}(r) \) respectively. For a set of rules \( \Pi \), \( \text{head}(\Pi) \) denotes the set of heads of these rules.

\[
\text{head}(\Pi) \overset{\text{def}}{=} \{ \text{head}(r) \mid r \in \Pi \}
\]

The body of \( r \) can be separated into two parts: \( \text{body}^+(r) = \{L_1, \ldots, L_m\} \), known as the prerequisites of \( r \); and \( \text{body}^-(r) = \{L_{m+1}, \ldots, L_n\} \). A rule \( r \) is said to be defeated by a set of literals \( X \) iff \( \text{body}^-(r) \cap X \neq \emptyset \).

If \( n = m \) then \( \text{body}^-(r) = \emptyset \) and \( r \) is referred to as a basic rule. If \( m = 0 \) then \( \text{body}^+(r) = \emptyset \) and \( r \) is referred to as a prerequisite-free rule. If \( n = 0 \) then \( r \) is referred to as a fact, which can be written \( L_0 \leftarrow \), or simply \( L_0 \).

For an extended logic program rule \( r \), \( r^+ \) is the basic rule obtained from \( r \) by deleting all weakly negated literals in the body of \( r \). Conversely \( r^- \) is the prerequisite-free rule obtained from \( r \) [DST03].

\[
\begin{align*}
r^+ & \quad \text{head}(r) \leftarrow \text{body}^+(r) \\
r^- & \quad \text{head}(r) \leftarrow \text{body}^-(r)
\end{align*}
\]

For any set of literals \( X \), the reduct \( \Pi^X \) of the logic program \( \Pi \) is defined as [GL91]

\[
\Pi^X \overset{\text{def}}{=} \{ r^+ \mid r \in \Pi, \text{body}^-(r) \cap X = \emptyset \}
\]

The reduct \( \Pi^X \) contains only basic rules.

For a logic program \( \Pi \) containing only basic rules, the immediate consequence operator \( \mathcal{T}_\Pi(X) \) is defined as

\[
\mathcal{T}_\Pi(X) \overset{\text{def}}{=} \{ \text{head}(r) \mid r \in \Pi, \text{body}(r) \subseteq X \}
\]

The least fixed point of \( \mathcal{T}_\Pi \) is

\[
\mathcal{T}_\Pi \uparrow^\omega \overset{\text{def}}{=} \bigcup_{n \geq 0} \mathcal{T}_\Pi \uparrow^n
\]

\(^1\)Informally, an extended logic program rule can be written equivalently as a default rule of the form
\[
L_1 \land \ldots \land L_m : \neg L_{m+1}, \ldots, \neg L_n \quad (n \geq m \geq 0)
\]
where
\[
T^n_\Pi \overset{n}{\triangleright} 0 \overset{\text{def}}{=} \emptyset, \\
T^n_\Pi \overset{n+1}{\triangleright} \overset{\text{def}}{=} T^n_\Pi (T^n_\Pi \overset{n}{\triangleright})
\]

For a logic program \( \Pi \), the set of literals in the language of \( \Pi \) is denoted \( \text{Lit}(\Pi) \). A set of literals \( X \) is consistent iff it does not contain a complementary pair of literals \( A \) and \( \neg A \), for any atom \( A \).

Let the logic program \( \Pi \) contain only basic rules. Then \( Cn(\Pi) \) denotes the smallest set of literals that is both logically closed and closed under \( T^n_\Pi \), i.e. \( Cn(\Pi) \) is the smallest set of literals \( X \) such that \( X \) is consistent or \( X = \text{Lit}(\Pi) \), and \( T^n_\Pi (X) \subseteq X \). When \( \Pi \) contains only basic rules, \( Cn(\Pi) = T^n_\Pi \overset{\omega}{\triangleright} = \bigcup_{n \geq 0} T^n_\Pi \overset{n}{\triangleright} \).

A set of literals \( X \) is an answer set of an extended logic program \( \Pi \) iff \( Cn(\Pi^X) = X \). Therefore, \( X \) is an answer set of program \( \Pi \) iff \( X = \bigcup_{n \geq 0} T^n_\Pi X \overset{n}{\triangleright} \). If an answer set \( X \) of extended logic program \( \Pi \) contains a pair of complementary literals then \( X \overset{\text{def}}{=} \text{Lit}(\Pi) \) [GL91].

The generating rules \( \Gamma^X_\Pi \) of an answer set \( X \) from an extended logic program \( \Pi \) are defined as [DST03]

\[
\Gamma^X_\Pi \overset{\text{def}}{=} \{ r \in \Pi \mid \text{body}^+(r) \subseteq X \text{ and } \text{body}^-(r) \cap X = \emptyset \}
\]

A prioritised extended logic program is a tuple \((\Pi, <)\) consisting of an ordered set of rules. The priority relation \(<\) is defined as a strict partial order on the rules of \( \Pi \). For any \( r_1, r_2 \in \Pi \), we write \( r_1 < r_2 \) to mean that rule \( r_1 \) has a higher priority than \( r_2 \).

**Conditional imperatives**

As well as default logic and extended logic programs, recent work on priorities has also explored priorities between conditional imperatives [Han08]. In addition, imperatives can be used as a natural way of expressing norms.

A conditional imperative \( i \) is of the form

\[ \alpha \Rightarrow! \beta \]

where \( \alpha \) and \( \beta \) are formulae. The imperative \( i \) demands that \( \beta \) be made true in a situation where \( \alpha \) is true. An unconditional imperative can be written \( \top \Rightarrow! \alpha \), or simply \( !\alpha \) [HM39].

Following Hansen [Han08], for an imperative \( i \) of the form \( \alpha \Rightarrow! \beta \), let \( g(i) \) and \( f(i) \) denote the antecedent \( \alpha \) and consequent \( \beta \) of \( i \) respectively. The materialisation \( m(i) \) of an imperative \( i \) represents the material implication \( g(i) \rightarrow f(i) \). For a set of imperatives \( I \), \( I^m \) denotes the set of material implications for these imperatives.

48
\[ I^m \overset{\text{def}}{=} \{ m(i) \mid i \in I \} \]

Similarly for \( I^g \) and \( I^f \).

A prioritised conditional imperative structure is a tuple \((I, f, g, <)\) consisting of a set of imperatives \(I\) ordered by a priority relation \(<\), and the functions \(g\) and \(f\). The priority relation \(<\) is defined as a strict partial order on \(I\). For any \(i_1, i_2 \in I\), we write \(i_1 < i_2\) to mean that imperative \(i_1\) has a higher priority than \(i_2\) [Han08].

### 2.4.2 Strategies for using priorities

A comparison study of approaches for handling priorities in non-monotonic reasoning is presented by Delgrande et al. [DSTW04]. Based on this survey, the authors go on to suggest a classification system containing a set of criteria that can be used to describe and to compare different approaches to reasoning with priorities.

- **Host system.** The first classification criterion describes the underlying system used to define the prioritised rules; for example default logic or extended logic programs. In this thesis, the directives defined for a behaviour will be implemented as an extended logic program.

- **Meta-level vs object-level.** In a meta-level approach the priority ordering is applied ‘externally’ on the rules of the system, whereas in an object-level approach the priority ordering is applied to constants representing names that are associated with the rules. For example in a meta-level approach, a prioritised default theory \((W, D, <)\) may specify that \(\delta_1 < \delta_2\). In an object-level approach, the priority relation is not between the default rules \(\delta_1, \delta_2\) but between the corresponding names \(n_{\delta_1}, n_{\delta_2}\) and so the equivalent priority relation would be expressed as \(n_{\delta_1} <_{N} n_{\delta_2}\), where \(<_N\) is the priority relation between names.

- **Static vs dynamic priorities.** Related to the distinction between meta-level and object-level priorities is the distinction between static and dynamic priorities. Static priorities are facts specified in advance. Therefore, an approach using meta-level priorities will also use static priorities. Dynamic priorities are contained in rules and so their applicability is determined ‘on the fly’. An approach using object-level priorities has the potential to use both static and dynamic priorities.

- **Properties of the priority order.** The ordering relation applied to the rules may be of different kinds, for example a strict partial order \(<\). In this thesis, a strict total order \(<\) will be used to express priorities between directives.

- **What the priority relation is between.** For example the priorities may be between the rules of a theory expressed using default logic or extended logic.
programs. Alternatively, the priorities may be between literals in the resulting extensions or answer sets.

- **Prescriptive vs descriptive priorities.** In a prescriptive strategy the priority relation $<$ specifies the order in which rules are to be applied, if possible. In a descriptive strategy $<$ specifies the desirability that a rule be applied, or equivalently a ranking on the desired outcomes. A given method for using priorities may contain elements of both of these strategies, or variations on them.

The distinction between a prescriptive and descriptive strategy can be illustrated by an example [DSTW04]. Let the set of rules $R$ be the logic program

$$
\begin{align*}
  r_1 & \quad b \leftarrow a \\
  r_2 & \quad \neg b \leftarrow \\
  r_3 & \quad a \leftarrow
\end{align*}
$$

where $r_1 < r_2 < r_3$.

In a prescriptive strategy, the rule $r_1$ cannot be applied because $a$ does not hold at this time. However, the rules $r_2$ and $r_3$ can be applied, giving a preferred answer set $\{a, \neg b\}$. In a descriptive strategy, it might be observed that by applying the rule $r_3$, the most preferred rule $r_1$ can be applied, giving the preferred answer set $\{a, b\}$.

- **From priorities to the preferred results.** If the priority relation $<$ is a partial order, then this may be extended to give a set of total orders $\preceq$ that are each used to generate a preferred result. Alternatively, the preferred results may be found directly using the partial order.

To find the preferred results, for example of a set of default rules, one option is to find the extensions of the priority-free rules and then determine which of these comply with the priority relation $<$. Alternatively, the preferred result may be found directly using $<$ and the prioritised default rules, without reference to the priority-free rules. Using this latter approach, it is possible that the preferred result is not an extension of the priority-free rules.

We now describe two approaches for reasoning using prioritised rules. These methods represent recent work on reasoning with priorities that also deal with the application of priorities to rules that include actions. We consider these approaches in terms of helping to design rules to implement norm-governed agents.

**Horty’s framework for prioritised default logic**

A descriptive approach to using priorities is described by Horty, where a method is proposed for finding the *proper scenarios* of a given default theory [Hor07, Hor12].
A scenario based on a prioritised default theory \((W,D,<)\) is a subset \(S\) of the set of default rules \(D\). As defined by Horty,

“From an intuitive standpoint, a scenario is supposed to represent the set of defaults that have been accepted by an agent, at some stage of its reasoning process, as providing sufficient support for their conclusions. Our central task in this paper is to characterize, as we will say, the proper scenarios – those scenarios that might ultimately be accepted by an ideal reasoning agent on the basis of the information contained in an ordered default theory” [Hor07] p.5.

The proper scenario consists of what Horty terms binding defaults, which are defined in terms of three concepts: triggered, conflicted and defeated default rules. A default is triggered in a particular scenario if its premise is entailed by the scenario.

**Definition 2.1.** (Triggered defaults) Let \(S\) be a scenario based on the prioritised default theory \((W,D,<)\). The defaults from \(D\) that are triggered in \(S\) are those belonging to the set

\[
\text{Triggered}_{(W,D)}(S) = \{ \delta \in D \mid W \cup \text{Conclusion}(S) \vdash \text{Premise}(\delta) \}
\]

where \(\vdash\) denotes ordinary classical consequence.

A default that is triggered is not binding if its conclusion is conflicted, meaning that the negation of its conclusion is entailed by the scenario.

**Definition 2.2.** (Conflicted defaults) Let \(S\) be a scenario based on the prioritised default theory \((W,D,<)\). The defaults from \(D\) that are conflicted in \(S\) are those belonging to the set

\[
\text{Conflicted}_{(W,D)}(S) = \{ \delta \in D \mid W \cup \text{Conclusion}(S) \vdash \neg \text{Conclusion}(\delta) \}
\]

For determining if a default is defeated, the concept of retracting and adding sets of defaults to the current scenario is used. The notation

\[
S^{D'/S'} = (S - S') \cup D'
\]

represents the result of retracting the defaults in \(S'\) from scenario \(S\) and then replacing these by the defaults in \(D'\). A default is defeated if it is of a lower priority than a set of defaults \(D'\) and if the resulting scenario is consistent.

**Definition 2.3.** (Defeated defaults) Let \(S\) be a scenario based on the prioritised default theory \((W,D,<)\). The defaults from \(D\) that are defeated in \(S\) are those belonging to the set

51
Defeated\(_{(W,D,<)}(S)\) = \{\delta \in D \mid \exists D' \subseteq Triggered\(_{(W,D)}(S)\) s.t.

(1) \(D' < \{\delta\}\),

(2) \(\exists S' \subseteq S\) with \(D' < S'\) s.t.

(a) \(W \cup Conclusion(S^{D'/S'})\) is consistent,

(b) \(W \cup Conclusion(S^{D'/S'}) \vdash \lnot Conclusion(\delta)\}\}

Using these definitions, the defaults in a prioritised default theory \((W,D,<)\) that are binding in a scenario \(S\) are those that are triggered but neither conflicted nor defeated.

**Definition 2.4. (Binding defaults)** Let \(S\) be a scenario based on the prioritised default theory \((W,D,<)\). The defaults from \(D\) that are binding in \(S\) are those belonging to the set

\[
Binding\(_{(W,D,<)}(S)\) = \{\delta \in D \mid \delta \in Triggered\(_{(W,D)}(S)\), \\
\delta \notin Conflicted\(_{(W,D)}(S)\), \\
\delta \notin Defeated\(_{(W,D,<)}(S)\}\}
\]

A scenario where all the defaults in it are binding is a stable scenario.

**Definition 2.5. (Stable scenario)** Let \(S\) be a scenario based on the prioritised default theory \((W,D,<)\). Then \(S\) is a stable scenario iff \(S = Binding\(_{(W,D,<)}(S)\)\).

A scenario \(S\) is a proper scenario iff it is both stable and grounded.

**Definition 2.6. (Grounded scenario)** Let \(S\) be a scenario based on the prioritised default theory \((W,D,<)\). Then \(S\) is a grounded scenario iff \(Th(W \cup Conclusion(S)) \subseteq Cn\(_S(W)\)\).

A proper scenario can also be defined using a quasi-inductive definition, here presented in the style used by Parent when he outlines Horty’s framework [Par10].

**Definition 2.7. (Proper scenario)** Let \(S\) be a scenario based on the prioritised default theory \((W,D,<)\). Then \(S\) is a proper scenario based on \((W,D,<)\) iff

\[
S = \bigcup_{i \geq 0} S_i
\]

where

\[
S_0 = \emptyset \\
S_{i+1} = \{\delta \in D \mid \delta \in Triggered\(_{(W,D)}(S_i)\), \\
\delta \notin Conflicted\(_{(W,D)}(S)\), \\
\delta \notin Defeated\(_{(W,D,<)}(S)\}\}
\]

Using the notion of a proper scenario, an extension of a default theory can be defined as the set of beliefs generated by a proper scenario.

52
Definition 2.8. (Extensions) Let \((W, D, <)\) be a prioritised default theory and \(E\) a set of formulae. Then \(E\) is an extension of \((W, D, <)\) iff \(E = Th(W \cup \text{Conclusion}(S))\), where \(S\) is a proper scenario based on this default theory.

A prioritised default theory \((W, D, <)\) may have multiple proper scenarios if the relation \(< \) is not a total order. Otherwise, the proper scenario is unique.

We demonstrate Horty’s framework using an example [Hor12]. Let \(p\), \(b\) and \(f\) stand respectively for the propositions that Tweety is a penguin, that Tweety is a bird and that Tweety can fly. Let the defaults \(D\) be

\[
\begin{align*}
\delta_1 & \quad p \rightsquigarrow \neg f \\
\delta_2 & \quad b \rightsquigarrow f
\end{align*}
\]

being instances of the general rules that birds fly and that penguins do not. Let \(W = \{p, b\}\) and \(\delta_1 < \delta_2\), specifying that the default about penguins has a higher priority than the default about birds.

Let \(S = \{\delta_1\}\). In order to determine if \(S\) is a proper scenario the method is carried out as follows.

\[
\begin{align*}
S_0 &= \emptyset \\
S_1 &\quad \text{Triggered}_{(W,D)}(S_0) = \{\delta_1, \delta_2\} \\
&\quad \text{Conflicted}_{(W,D)}(S) = \{\delta_2\} \\
&\quad \text{Defeated}_{(W,D, <)}(S) = \{\delta_2\} \\
&\quad = \{\delta_1\}
\end{align*}
\]

\[
\begin{align*}
S_2 &\quad \text{Triggered}_{(W,D)}(S_1) = \{\delta_1, \delta_2\} \\
&\quad \text{Conflicted}_{(W,D)}(S) = \{\delta_2\} \\
&\quad \text{Defeated}_{(W,D, <)}(S) = \{\delta_2\} \\
&\quad = \{\delta_1\}
\end{align*}
\]

\[
S_3 = S
\]

Therefore, \(S = \{\delta_1\}\) is a proper scenario based on the prioritised default theory \((W, D, <)\). As the relation \(<\) is a total order, \(S\) is the unique proper scenario of this default theory. We have \(\text{Conclusion}(S) = \{\neg f\}\) and \(W = \{p, b\}\), and so the extension of the theory is \(E = Th(\{p, b, \neg f\})\) and we can conclude that Tweety does not fly.

Table 2.1 shows the classification of Horty’s method using the classification system of Delgrande et al. Horty’s method is a descriptive method as the proper scenarios are those where the highest priority default rules are applied.

In order to motivate his framework, Horty generally presents examples in an abstract form or examples which appear to refer to beliefs. In some examples,
however, Horty explicitly refers to the default rules as representing reasons to act, rather than reasons to believe.

One such example used by Horty is referred to as the ‘order puzzle’ [Hor07]. Let $w$, $h$ and $o$ stand respectively for the propositions that it is winter, the heating is turned on and the window is open. Let the defaults $D$ be

$$
\begin{align*}
\delta_1 & : h \Rightarrow o \\
\delta_2 & : w \Rightarrow \neg o \\
\delta_3 & : w \Rightarrow h 
\end{align*}
$$

Let $W = \{w\}$ and $\delta_1 < \delta_2 < \delta_3$.

The default rules in this theory are considered by Horty as now describing imperatives, commands or orders. Therefore, the defaults $\delta_1$, $\delta_2$ and $\delta_3$ are to be read as: “whenever the heating is on, the window should be opened”; “during the winter, the window should not be opened”; and “during the winter, the heating should be turned on” respectively.

Using Horty’s framework and demonstrating that this is a descriptive approach, the scenario $S = \{\delta_1, \delta_3\}$ is found to be the proper scenario based on the prioritised default theory $(W,D,\prec)$. We have $\text{Conclusion}(S) = \{o, h\}$ and $W = \{w\}$, and so the extension of the theory is $E = Th(\{w, o, h\})$. Therefore, it is concluded that the heating must be turned on and the window opened.

By including an action on the left hand side of a default rule, however, the interpretation of actions in this example becomes unclear. This is seen during Horty’s discussion of the order puzzle. At some points, the action is considered as having already been carried out, as if it is a fact; while at other times, all actions are considered to be carried out simultaneously, once the reasoning process has been completed.

In fact, Horty states early on when presenting his framework that he will use a relaxed distinction between beliefs and actions in his examples.

“Throughout this book, we will be slipping back and forth, rather casually, between what might be called practical and epistemic reasons –

<table>
<thead>
<tr>
<th>Host system</th>
<th>Default logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meta-level vs object-level</td>
<td>Meta-level priorities</td>
</tr>
<tr>
<td>Static vs dynamic</td>
<td>Static priorities</td>
</tr>
<tr>
<td>Properties of $&lt;$</td>
<td>Strict partial order</td>
</tr>
<tr>
<td>Priority relation</td>
<td>Priorities between rules</td>
</tr>
<tr>
<td>Prescriptive vs descriptive</td>
<td>Descriptive</td>
</tr>
<tr>
<td>Preferred results</td>
<td>Preferred result found using the partial order $\prec$. Preferred result found directly from $\prec$ without reference to the priority-free rules.</td>
</tr>
</tbody>
</table>

Table 2.1: Classification of Horty’s method.
reasons for actions, versus reasons for conclusions” [Hor12] p.18.

Therefore, the treatment of actions in the examples used by Hory is not precise.

**Hansen’s prioritised conditional imperatives**

A prescriptive approach to using priorities is described by Hansen, where a method is proposed for finding the preferred subsets of a prioritised conditional imperative structure [Han08]. A preferred subset of \((I, f, g, <)\) is a subset of the imperatives \(I\) that do not conflict given a specific situation described by a set of formulae \(W\).

Let \(I\) be a prioritised conditional imperative structure \((I, f, g, <)\) and let \(\Delta\) be a subset of \(I\). Then \(P_I(W, \Delta)\) contains just the subsets of \(\Delta\) that are preferred given \(W\). An action or behaviour described by the imperatives is obligatory iff it is so with respect to all the preferred subsets \(P_I(W, \Delta)\) of the imperatives \(\Delta\).

Hansen’s method is inspired by that of Brewka for prioritised default logic, where the default rules are restricted to unconditional defaults of the form \(\top \rightsquigarrow \gamma\) [Bre89]. Brewka’s method maximises the number of higher priority default rules in the preferred subsets. To do this, Brewka’s construction considers the rules in priority order and selects a rule if its conclusion is consistent with \(W\) and the rules selected to that point. Brewka’s approach cannot be applied directly to conditional rules, however, because it only considers the \(Conclusion(\delta)\) of a default rule or the consequent \(f(i)\) of an imperative, with no reference to the premise or antecedent of the rule.

Hansen considers the ‘obeyability’ of rules in order to adapt Brewka’s approach to conditional rules. To do this, instead of checking that the consequents \(I_f\) of the imperatives are consistent, Hansen’s method checks that the materialisations \(I_m\) of the imperatives are consistent. Hansen observes that

“For any unconditional imperative \(i\) we have \(\vdash f(i) \leftrightarrow m(i)\)” [Han08] p.29

and so for unconditional imperatives the methods of Brewka and Hansen will coincide.

**Definition 2.9.** (*Preferred maximally obeyable subsets*) Let \(I = (I, f, g, <)\) be a prioritised conditional imperative structure and \(\Delta\) be a subset of \(I\). Then \(\Gamma \in P_I(W, \Delta)\) iff (i) \(W \not\models \bot\) and (ii) there exists a full prioritisation \(\prec\) of \(<\) such that

\[
\Gamma = \bigcup_{i \in \Delta} \Gamma_i
\]

where

\[
\Gamma_i = \begin{cases} 
\bigcup_{j < i} \Gamma_j \cup \{i\} & \text{if } W \cup \left[ \bigcup_{j < i} \Gamma_j \cup \{i\} \right]_m \not\models \bot \\
\bigcup_{j < i} \Gamma_j & \text{otherwise}
\end{cases}
\]
We demonstrate Hansen’s method using an example [Han08]. This example has the same form as the order puzzle presented by Horty [Hor07]. However, Hansen suggests that the preferred result found by Horty’s framework is unintuitive. Instead, Hansen proposes a different method and interpretation of the obligatory actions given the rules.

Let the imperatives $I$ be

\begin{align*}
  i_1 & \quad \text{drink} \Rightarrow \neg \text{drive} \\
  i_2 & \quad \text{party} \Rightarrow \neg \text{drive} \\
  i_3 & \quad \text{party} \Rightarrow \neg \text{drink}
\end{align*}

where $i_1$ represents your mother’s imperative: “if you drink anything, then don’t drive”, $i_2$ represents your best friend’s imperative: “if you go to the party, then you do the driving” and $i_3$ represents an acquaintance’s imperative: “if you go to the party, then have a drink with me”.

Let $I = (I, f, g, <)$ with $I$ defined as above and $i_1 < i_2 < i_3$, specifying that your mother’s imperative has the highest priority and the imperative of your acquaintance has the lowest priority. Let $W = \{\text{party}\}$, meaning you decide to go to the party.

As $<$ is a total order in this example, there will only be one preferred subset $\Gamma \in P_I(W, I)$. To find the preferred maximally obeyable subset $\Gamma \in P_I(W, I)$ the imperatives are considered in priority order. For an imperative $i$, if the materialisation $m(i)$ is consistent with $W$ and the imperatives accepted before, then $i$ is accepted as part of the preferred subset $\Gamma \in P_I(W, I)$.

First, $i_1$ is accepted as it is consistent with $W$, then $i_2$ is accepted as it is consistent with $W$ and $i_1$, however $i_3$ is not accepted as it would lead to an inconsistent result. Therefore, the preferred maximally obeyable subset $P_I(W, I) = \{\{i_1, i_2\}\}$, so we can conclude that given that you go to the party you must do the driving.

Table 2.2 shows the classification of Hansen’s method. Hansen’s method is an alternative prescriptive approach to that described by Delgrande et al. Instead of using the priority order to specify the order in which the rules should be applied, Hansen’s method uses the priority order to specify the order in which the rules should be obeyed. The preferred subsets of the prioritised conditional imperative structure are found by considering the imperatives in priority order and accepting them into the preferred subset if their materialisation is consistent with $W$ and the materialisations of the imperatives accepted before.

Hansen explicitly attempts to represent actions to be performed in his examples and explores different truth definitions that can be applied to a preferred subset of a set of imperatives in order to determine whether an action is obligatory. Different methods for reasoning using priorities are considered, before the method shown above is presented. This method is chosen by Hansen because it returns what Hansen identifies as the ‘intuitively correct result’ to the order puzzle.
After reading the document, the plain text representation is:

<table>
<thead>
<tr>
<th>Host system</th>
<th>Conditional imperatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meta-level vs object-level</td>
<td>Meta-level priorities</td>
</tr>
<tr>
<td>Static vs dynamic</td>
<td>Static priorities</td>
</tr>
<tr>
<td>Properties of &lt;</td>
<td>Strict partial order</td>
</tr>
<tr>
<td>Priority relation</td>
<td>Priorities between rules</td>
</tr>
<tr>
<td>Prescriptive vs descriptive</td>
<td>Prescriptive-based</td>
</tr>
<tr>
<td></td>
<td>Instead of specifying the order rules are to be applied the priority order specifies the order rules should be obeyed.</td>
</tr>
<tr>
<td>Preferred results</td>
<td>Preferred results found by extending the partial order &lt; to give a set of total orders ≺. Preferred result found directly from &lt; without reference to the priority-free rules.</td>
</tr>
</tbody>
</table>

Table 2.2: Classification of Hansen’s method.

Again, however, the interpretation of actions on the left hand side of imperatives is not always consistent. A different result may be considered intuitively correct or incorrect, depending on whether the actions in a set of imperatives are considered to be enacted all at once, after the reasoning process is complete, or whether actions in the left hand side of imperatives have somehow already been carried out.

**Reasoning with prioritised directives**

As part of this thesis, we develop a system for implementing norms in a manner that can be understood and reasoned about by agents. We use directives to implement these norms, where the directives map conditions that can hold in the agent’s current state to actions for the agent to perform. Therefore, we restrict the valid format of a directive such that actions cannot appear in the antecedent. By limiting the form of directives in this way, we are able to ensure that we maintain a consistent treatment of actions in our rules.

The agent is only capable of performing an established set of actions, such as the actions to turn left, turn right and to move forwards. We do not allow directives that explicitly direct an agent not to perform a specified action. For example, we do not allow a directive that specifies that the agent must not turn left in a particular situation. This does not lead to any loss of expressibility as, in practice, norms that prohibit an action can be equivalently written as directives that specify other actions to be performed. For example, we can instead use directives that specify that an agent in this particular situation must turn right or that the agent must move forwards.

Under these restrictions, it is not possible for the acceptance of one directive to influence whether another directive is triggered or activated. Therefore, we do not allow ‘chaining’ of rules to occur. This significantly simplifies the treatment
of priorities between directives and means that many of the issues encountered by Hory and Hansen when reasoning about rules that relate to actions do not arise.

2.4.3 Compiling priorities in logic programs

While existing methods for reasoning with priorities may explore issues that do not occur when using directives, there are other areas of current research on reasoning with priorities that can assist with the implementation of norms for agents. We now describe a translation framework for compiling priorities in logic programs.

This framework is particularly important to our implementation as, given a method for reasoning with priorities over directives, we require a method by which an agent can be implemented to carry out this reasoning process itself and to identify the preferred actions to perform given a prioritised set of directives. The framework outlined below describes a general method that we are able to adapt for carrying out this process.

A framework for expressing priority information in an extended logic program is described by Delgrande et al. [DST03]. Using this framework, a method for handling priorities can be encoded in an extended logic program by means of a translation. A prioritised extended logic program \((\Pi, <)\) is translated into an extended logic program without priorities \(T(\Pi, <)\), as follows. The answer sets of the translated extended logic program \(T(\Pi, <)\) correspond to the preferred results of the prioritised extended logic program \((\Pi, <)\).

To create the translated extended logic program, the prioritised extended logic program \((\Pi, <)\) must have an associated set \(N\) of terms acting as names for the rules in \(\Pi\). For each rule \(r \in \Pi\) there is a corresponding name \(n_r \in N\). Similarly, for every \(r, r' \in \Pi\) where \(r < r'\), there is a priority order \(<_N\) such that \(n_r <_N n_{r'}\).

In order to compile the priority information of \((\Pi, <)\) into an extended logic program without priorities, the translation creates a tagged logic program \(\Pi'\), where \(T(\Pi, <) = \Pi'\). A tagged logic program contains control elements that are used to ensure that the application of the rules in the original program \(\Pi\) are in accordance with the intended application order given \(<\). The control elements, or tags, allow application of the rules in \(\Pi\) to be detected and controlled.

To create the tagged logic program \(\Pi'\) a set of tags are introduced for each rule \(r \in \Pi\).

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{ap}(n_r))</td>
<td>To detect and control rule application.</td>
</tr>
<tr>
<td>(\text{bl}(n_r))</td>
<td>To detect when a rule is blocked.</td>
</tr>
<tr>
<td>(\text{ok}(n_r))</td>
<td>To control the order that rules are applied.</td>
</tr>
</tbody>
</table>

Additional tags can be introduced according to the method of using priorities that is being encoded.
To detect when a rule \( r \) is applicable the atom \( \text{ap}(n_r) \) is used, separating applicability of a rule from its actual application.

\[
\text{head}(r) \leftarrow \text{ap}(n_r) \\
\text{ap}(n_r) \leftarrow \text{body}(r)
\]

A rule \( r \) is not applicable when one of the elements of \( \text{body}(r) \) cannot be satisfied. In this case the rule is said to be blocked. To detect when a rule \( r \) is blocked the atom \( \text{bl}(n_r) \) is used, which checks each element of \( \text{body}(r) \).

\[
\text{for each } l^+ \in \text{body}^+(r): \quad \text{bl}(n_r) \leftarrow \lnot l^+ \\
\text{for each } l^- \in \text{body}^-(r): \quad \text{bl}(n_r) \leftarrow l^-
\]

Therefore, the applicability of each rule \( r \in \Pi \) is described within the answer set \( X \) of the tagged program \( \Pi' \).

\[
\text{ap}(n_r) \in X \text{ iff } \text{bl}(n_r) \notin X
\]

A rule \( r \) should only be considered if every rule \( r' \in \Pi \) where \( r' < r \) has already been considered first. The atom \( \text{ok}(n_r) \) is used to control the order that rules are considered in the logic program. Therefore, \( \text{ok}(n_r) \) must be taken into account in the use of \( \text{ap}(n_r) \) and \( \text{bl}(n_r) \) shown above.

\[
\text{ap}(n_r) \leftarrow \text{ok}(n_r), \text{body}(r) \\
\text{for each } l^+ \in \text{body}^+(r): \quad \text{bl}(n_r) \leftarrow \text{ok}(n_r), \lnot l^+ \\
\text{for each } l^- \in \text{body}^-(r): \quad \text{bl}(n_r) \leftarrow \text{ok}(n_r), l^-
\]

The tag \( \text{ok}(n_r) \) is defined so that it will hold when the next highest priority rule has been considered.

\[
\text{ok}(n_{r_i}) \leftarrow \\
r_{i-1} < r_i: \quad \text{ok}(n_{r_i}) \leftarrow \text{ap}(n_{r_{i-1}}) \\
r_{i-1} < r_i: \quad \text{ok}(n_{r_i}) \leftarrow \text{bl}(n_{r_{i-1}})
\]

The general framework of Delgrande et al. is demonstrated by an example \([\text{DST03}]\). Let the rule \( r_2 \) be

\[
r_2 \quad p \leftarrow q, \lnot w
\]

Let there be some rule \( r_1 \) such that \( r_1 < r_2 \) and let \( n_i \) denote the name of rule \( r_i \). The translation \( T(r_2) \) is

\[
p \leftarrow \text{ap}(n_2) \\
\text{ap}(n_2) \leftarrow \text{ok}(n_2), q, \lnot w \\
\text{bl}(n_2) \leftarrow \text{ok}(n_2), \lnot q \\
\text{bl}(n_2) \leftarrow \text{ok}(n_2), w \\
\text{ok}(n_2) \leftarrow \text{ap}(n_1) \\
\text{ok}(n_2) \leftarrow \text{bl}(n_1)
\]
The translation method described so far is general. Delgrande et al. demonstrate their translation on a prescriptive approach to using priorities. In particular, this strategy is able to reason about logic programs containing dynamic priorities. In order to encode this feature, an additional tag atom $\text{rdy}(n_r, n_{r'})$ for $r, r' \in \Pi$ is added, where $\text{rdy}(n_r, n_{r'})$ is used to define $\text{ok}(n_r)$ based on the priority order between rule names $<_N$.

| $\text{rdy}(n_r, n_{r'})$ | To control the order that rules are applied when the original logic program rules contain dynamic priorities. |

**Definition 2.10.** *(Translation $\mathcal{T}_D$)* Let $\Pi$ be an extended logic program over language $\mathcal{L}$ where $\Pi = \{r_1, ..., r_k\}$ and $\Pi$ is ordered by a static priority relation $<_k$ or dynamic priorities in the rules of $\Pi$. Let $\mathcal{L}^+$ be the language obtained from $\mathcal{L}$ by adding for each $r, r' \in \Pi$ the atoms $\text{ap}(n_r)$, $\text{bl}(n_r)$, $\text{ok}(n_r)$ and $\text{rdy}(n_r, n_{r'})$. The logic program $\mathcal{T}_D(\Pi)$ over $\mathcal{L}^+$ is

$$\mathcal{T}_D(\Pi) = \bigcup_{r \in \Pi} \tau(r)$$

where the set $\tau(r)$ consists of the following rules, for each $l^+ \in \text{body}^+(r)$, $l^- \in \text{body}^-(r)$ and $r', r'' \in \Pi$.

- $a_1(r)$: $\text{head}(r) \leftarrow \text{ap}(n_r)$
- $a_2(r)$: $\text{ap}(n_r) \leftarrow \text{ok}(n_r), \text{body}(r)$
- $b_1(r, l^+)$: $\text{bl}(n_r) \leftarrow \text{ok}(n_r), \text{not } l^+$
- $b_2(r, l^-)$: $\text{bl}(n_r) \leftarrow \text{ok}(n_r), l^-$
- $c_1(r)$: $\text{ok}(n_r) \leftarrow \text{rdy}(n_r, n_{r_1}), ..., \text{rdy}(n_r, n_{r_k})$
- $c_2(r, r')$: $\text{rdy}(n_r, n_{r'}) \leftarrow \text{not } (n_r <_N n_{r'})$
- $c_3(r, r')$: $\text{rdy}(n_r, n_{r'}) \leftarrow (n_{r'} <_N n_r, \text{ap}(n_{r'}))$
- $c_4(r, r')$: $\text{rdy}(n_r, n_{r'}) \leftarrow (n_{r'} <_N n_r, \text{bl}(n_{r'}))$
- $t(r, r', r'')$: $(n_r <_N n_{r''}) \leftarrow (n_r <_N n_{r'')); (n_{r'} <_N n_{r''})$
- $as(r, r')$: $\neg(n_{r'} <_N n_r) \leftarrow (n_r <_N n_{r'})$

The rules $t(r, r', r'')$ and $as(r, r')$ are used to handle dynamic priorities.

We demonstrate the translation of the method for using priorities described by Delgrande et al. using an example [DST03]. Let the logic program $\Pi$ be

- $r_1$: $\neg a \leftarrow$
- $r_2$: $b \leftarrow \neg a, \text{not } c$
- $r_3$: $c \leftarrow \text{not } b$
- $r_4$: $(n_2 <_N n_3) \leftarrow \text{not } d$

where $n_i$ denotes the name of rule $r_i$.

The translation $\mathcal{T}_D(\Pi)$ contains the following rules
Using the translation $T_D(\Pi)$ the preferred answer set of $\Pi$ is found to be $\{\neg a, b, (n_2 <_N n_3)\}$. This answer set agrees with the dynamic priority order $n_2 <_N n_3$, which specifies that rule $r_2$ has a higher priority than rule $r_3$ in the case that $d$ cannot be derived. Therefore, $b$ is in the preferred answer set rather than $c$.

This translation method for using priorities is classified in Table 2.3, using the classification provided by Delgrande et al. in their survey [DSTW04]. The method of Delgrande et al. is prescriptive, where the rules are considered for application in priority order. This is achieved by use of the tags, which allow rule application to be controlled and so the method ensures that rules are only considered in priority order.

Delgrande et al. also explain how the translation method $T_D$ can be altered for the specific case of a prioritised extended logic program that contains only static priorities. This is done by removing the rules $c_2(r, r'), t(r, r', r'')$ and $as(r, r')$ from the translation, and modifying the rules $c_3(r, r')$ and $c_4(r, r')$ to contain no reference to $(n_{r'} <_N n_r)$ literals.

<table>
<thead>
<tr>
<th>Host system</th>
<th>Extended logic programs under answer sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meta-level vs object-level</td>
<td>Object-level priorities</td>
</tr>
<tr>
<td>Static vs dynamic</td>
<td>Dynamic priorities</td>
</tr>
<tr>
<td>Properties of $&lt;$</td>
<td>Strict partial order</td>
</tr>
<tr>
<td>Priority relation</td>
<td>Priorities between rules</td>
</tr>
<tr>
<td>Prescriptive vs descriptive</td>
<td>Prescriptive</td>
</tr>
</tbody>
</table>
| Preferred results | Preferred results found using the partial order $<$.
Preferred result found directly from $<$ without reference to the priority-free rules. |

Table 2.3: Classification of the method of Delgrande et al.

The framework of Delgrande et al. is demonstrated to be able to encode other existing methods for using priorities to resolve conflicts between rules. However, the framework is only suited to prescriptive approaches [DST03].
2.5 Review

In this chapter we have discussed the main areas of related work pertaining to the implementation of norm-governed multi-agent systems that is the focus of this thesis.

The existing body of work on norms is very extensive and so we have focussed our discussion on norms in multi-agent systems. We have not described existing work from other fields where norms have been adopted, such as in a social or a legal setting. Even with this restriction, we still found that there are a number of varied concepts and definitions attributed to norms in multi-agent systems (section 2.1).

We have highlighted an apparent disparity between regulation strategies employed in simulated multi-agent systems and in real world multi-agent system applications (section 2.2). Real world multi-agent system applications appear to manage the interactions of agents in a centralised manner, although no transparent implementation frameworks are provided. In the few cases where a decentralised control system is used, methods from swarm robotics have been tried. There do not appear to be any real world industrial examples where norms are used to manage the interactions of agents.

When we turn to implementation methods for norm-governed agents or norm-governed multi-agent systems, again, the existing body of work is extensive. We have presented only a small sample of work on norm emergence, focussing instead on the many issues that are encountered as part of norm adherence (section 2.3). Within that topic, we have also found it necessary to limit the approaches presented. For example, we have provided only brief examples of systems for extending BDI agents to allow the inclusion of norms.

We concluded the related work presented in this chapter by outlining different general strategies for reasoning using prioritised rules (section 2.4.2). In particular, we focussed on two specific approaches, those of Horty and Hansen, who have considered the application of priorities to rules that contain actions. The treatment of actions within these approaches was not always consistent, however, particularly in terms of how actions in the antecedent of rules should be interpreted.

Throughout this chapter our intention was to demonstrate that existing work explores a range of possible multi-agent system and norm implementation issues. Nevertheless, the question of whether norms are an effective method for managing the actions of agents in a multi-agent system has not been addressed. Instead, existing work either appears to ignore the potential for using norms, in favour of other (typically centralised) control mechanisms; or to assume that norms provide a guaranteed and beneficial mechanism for managing agent actions, in order to investigate higher level considerations such as sanctions.

In the remainder of this thesis we directly address the issue of whether norms offer an effective means of coordinating the actions of autonomous agents.
In the next chapter, Chapter 3, we describe an implementation method for allowing an agent to take norms into account using prioritised sets of *directives*. We define a directive such that an action cannot appear in the antecedent. This ensures that we maintain a consistent interpretation of actions within directives and eliminates many of the problematic details encountered by Horty and Hansen. We are therefore able to define a simpler reasoning formalism than those described in this chapter (section 2.4.2).

As part of the implementation method described in Chapter 3 we also introduce the simulation test bed Sinatra. Sinatra is specifically designed to assist the simulation, visualisation and analysis of the actions of agents that are implemented using directives. In particular, Sinatra includes features that allow the user to interact with the simulation, making it easier for the user to understand how and when agents follow different directives. This makes Sinatra distinct from the existing simulation tools described in this chapter (section 2.2.1).

In Chapter 4 we use Sinatra and agents implemented using directives to explore the assumption that norms are an effective method for regulating a multi-agent system. We find that using locally applied directives, there will be situations where the agents are unable to act as desired by the system designer. This result echoes the suggestions of others who have investigated strategies for norm compliance (section 2.3.6) and self-governed simulated multi-agent systems (section 2.2.1).

In Chapters 5, 6 and 7 we address this problem with norms by exploring methods for implementing exception-detection and recover mechanisms. These mechanisms can be used to help to overcome the problems encountered by agents in norm-governed systems, where local rules can fail to achieve the desired global result. By combining these additional mechanisms with agents that are implemented using directives, it is possible for a better regulated multi-agent system to be implemented. In Chapter 8 we present recommendations for the implementation of such a system, based on the findings of the previous chapters and assuming that it is desirable for the system to be implemented using norms.
3 Sinatra: a simulation test bed for implementing norm-governed multi-agent systems

We want to investigate the implementation of norms in a multi-agent system, with the intention of determining the extent to which norms can be used to manage agent interactions. However, it is beyond the scope of this work to investigate solutions to the additional problems encountered by actual physical robots in a physical environment. Therefore, we limit our investigate to a simulated multi-agent system.

To this end a simulation test bed, called Sinatra, has been developed. Sinatra is designed to allow the implementation and testing of a set of norm-governed agents in a multi-agent system.

The agents in Sinatra determine their actions by referring to directives that are implemented by the system designer. A directive maps a particular combination of agent perceptions to an action for the agent to perform. A set of directives, together with a set of constraint formulae to prevent certain actions being executed concurrently, form a behaviour. Behaviours provide a mechanism for the implementation of norms that have been specified by the system designer. The behaviours allow an agent to take a norm or set of norms into account when determining its actions.

In this chapter, we begin by providing an overview of the Sinatra test bed and features of its use as a tool for investigating the implementation of norms to manage agent interactions. We then describe the agents that are implemented in Sinatra, before turning to the directives that are used by these agents. We detail the different stages that make up the behaviour method, with a continuing example to demonstrate the development of a behaviour from an initial set of directives. Finally, we show how behaviours are used by a Sinatra agent to determine its actions.

Subsequent chapters will use Sinatra to investigate norms, implemented using directives, to manage the actions of agents. We will implement directives to manage agent interactions, to allow an agent to interact with a centralised control mechanism and to coordinate the actions of agents.
3.1 Sinatra overview

Sinatra is a Java-based simulation and visualisation tool. Using the behaviour method, which we will describe in due course, agents can be implemented to act in different ways depending on different situations they encounter. These agents can then be simulated in Sinatra, where a graphical visualisation of the multi-agent system is used to show the agents’ actions.

Figure 3.1 shows the Sinatra GUI. The environment consists of a fixed-size grid network of locations. Each grid location may contain at most one agent. Agents cannot move outside of the grid and the grid locations do not wrap around at the edges. An agent is represented in Sinatra by a coloured body and a black head, making it possible to determine the direction that the agent is facing.

![Figure 3.1: Sinatra GUI showing three agents in a grid environment.](image)

In general, the number of agents in a multi-agent system may range from just two agents to many hundreds or thousands of agents. Small agent populations allow agent interactions to be investigated at a local (agent) level, focussing on the individual actions of the agents involved. Large agent populations allow agent interactions to be investigated at a global (simulation) level, focussing on global trends rather than individual actions, e.g. the swarm approach to coordination in multi-robot systems.

In this thesis we are interested in the actions of individual agents and so Sinatra is designed to accommodate small agent populations. In our experiments, we will consider multi-agent systems populated by between one and six agents. A system containing a single agent is used to allow the individual actions of an agent to be investigated and verified before other agents are introduced.

The Sinatra simulation domain is made up of a grid of locations for the agents to move between. Each grid cell represents a possible location of the agent within the environment. An agent occupies a single grid cell location and only one agent can occupy a location at a time. We assume that the agents are equipped with some
form of collision prevention mechanism, which enforces the requirement of only one agent per location. Therefore, the size of the grid environment acts as the limiting factor for the number of agents that can be simulated in Sinatra.

The visualisation facilities in Sinatra allow the user to observe the whole simulation domain and the actions of the agents. Using the Sinatra GUI, the user can observe the location and orientation of each agent. The orientation of an agent is updated on the GUI whenever the agent turns. Similarly, the agent is animated moving between two adjacent grid cells whenever the agent performs a move action.

The agents in Sinatra are autonomous agents, who act independently and use directives, implemented as behaviours, to determine their actions and to manage their interactions with other agents. By observing the agents’ actions using the visualisation of the multi-agent system, Sinatra can be used to verify that agents are acting as intended and to experiment with the implementation of new behaviours.

The Sinatra simulation executes each agent on a separate thread. This allows the agents to act concurrently, while also being able to perceive the environment and to determine their actions in an autonomous manner. The actions of multiple agents are synchronised using a simulation clock, where each agent is limited to performing a single action per time step. The current system time is displayed in the Sinatra GUI, where the time shown in Figure 3.1 indicates that the simulation has not yet been started by the user.

Due to the often complicated nature of agent interactions, Sinatra is implemented to allow the user to control the progress of the simulation. These controls are shown in Figure 3.1 as the row of buttons below the grid environment.

The simulation and corresponding visualisation of the multi-agent system can be run in a continuous manner (Run), or stepped (Step) to allow the user to observe each time step in isolation. The simulation can be paused (Pause) at any point while it is running or reset (Reset) to begin a completely new simulation.

Throughout the simulation, a record is kept of the actions performed by the agents. While the simulation is paused, this record can be written to a temporary file (Log) and used to replay the simulation so far. This feature provides the user with the opportunity to observe an aspect of the simulation that has just occurred, such as a particular interaction between the agents. The user can replay the simulation by stepping backwards (<) and forwards (>) between the start of the simulation and the time step reached so far. Once returned to the current time point, the simulation can be resumed (Run or Step) as if it had merely been paused.

It is important when investigating the implementation of norms to be able to view agents following these norms and to be able to observe their interactions. A set of norms developed by the system designer may appear reasonable ‘on paper’ but can result in surprising emergent behaviour when used in practice. For example, a set of directives that works correctly for an individual agent may cause the agent
to become stuck repeating a particular set of states when the agent interacts with another agent. Therefore, Sinatra has been developed to allow different norms, implemented using directives, to be quickly and easily incorporated into the Sinatra agents and then to visualise multiple agents using these directives and interacting in a shared environment.

Figure 3.2 shows the UML class diagram for the Sinatra test bed, showing the main classes in the Sinatra implementation and the relationships between them.

![UML Class Diagram](Image)

**Figure 3.2: The Sinatra UML class diagram.**

The main challenge when implementing Sinatra was to ensure that the agents move correctly, both in terms of maintaining the restriction that only one agent may occupy a grid location at a time and in terms of animating the movement of the agents. As each agent operates on a separate thread, each agent independently perceives its environment and determines how to act based on this local information. This implementation allows the agents to be autonomous; however, it also
means that two agents may independently perceive that the same adjacent location
is unoccupied and both attempt to move into this location during the same time
step.

A lock is associated with each grid cell in the environment. When an agent
attempts to move into an adjacent location the agent must first obtain the lock on
this location. Two agents may attempt to move into the same location during the
same time step but only one agent will hold the lock and so be permitted to move
into this location. The other agent will remain stationary in its current location.
Synchronised Java methods are used for threads to access these lock objects, to
ensure that locking of locations is implemented correctly.

An agent is animated moving between two adjacent grid cells by repeatedly clear-
ing any graphics drawn in these grid cells and redrawing the agent graphic one pixel
further along. During the point when the agent is crossing the boundary between
locations, the agent graphic must be drawn in the correct position in both grid cells.
This allows the animation to show the agent moving seamlessly between adjacent
locations.

As this animation can take about one second to complete, a system clock is used
to ensure that agents that turn or remain stationary during a time step do not
begin to act again until the animation of any moving agents has been completed.
To implement this feature, each agent thread is synchronised on the global Simula-
tion object, which implements the observer design pattern. When an agent thread
has finished acting for a time step, the thread notifies the Simulation object and
then becomes blocked. When the Simulation object has received an update from
each agent thread, the Simulation object increments the global simulation time and
notifies all of its observers, allowing the threads to resume.

Another challenge when implementing Sinatra was the implementation of the
replay method. The location and orientation of each agent is recorded in an XML
file for each time step of the simulation. An XML file format was chosen because the
information required by the replay method is simple and repetitive, making it easy
to encode in an XML schema. Java Architecture for XML Binding (JAXB) is used
to create an ObjectFactory object that records the state of the simulation at the
end of each time step. When the user presses the Log button, a Marshaller object
is used to write these stored simulation states to a temporary XML file. As the
user moves through the replay using the < and > buttons, an Unmarshaller object
reads the appropriate time step from the XML file, which is used to reconstruct the
corresponding simulation state.

The visualisation component in Sinatra is relatively basic and was not the main
focus of the Sinatra implementation. We experimented with ways of displaying
more information about the simulation; for example, to signify when agents use
particular behaviours or perceive certain events. However, this proved difficult to
implement, as just showing agents moving around the environment can be distracting and difficult to follow. Therefore, it was difficult to visualise more information about the simulation in a coherent manner. Instead, the visualisation of the simulation can be used in conjunction with the replay method to allow the user to understand the simulation that is taking place.

### 3.2 Sinatra agent properties

Sinatra is intended to be a general and versatile test bed, allowing for a broad range of norm implementation experiments to be carried out. A general agent model is required for the Sinatra agents that can be easily extended to be applicable in different experiments. The capabilities of the Sinatra agents have been inspired by existing autonomous mobile robots, for example the LEGO Mindstorms® NXT [LEG] and the Roomba® vacuum cleaning robot by iRobot [iRo].

In order to maintain the general nature of the Sinatra test bed, however, the Sinatra agents are intentionally designed to be relatively basic, unintelligent agents. This is to ensure that we do not make unnecessary assumptions about the intelligence or purpose of the agents that will be simulated in Sinatra. More intelligent or sophisticated agents, with additional perception and action capabilities, can be implemented in Sinatra by extending this original agent model; examples are shown in Chapter 7.

For these reasons, the Sinatra agents are modelled on the following design points.

- The Sinatra agents are autonomous mobile agents that inhabit a grid environment.
- The agents are able to move around the environment by turning on the spot and moving forwards.
- The agents possess only the perception capabilities that are required for them to be able to operate within the grid environment.

In addition, we assume that each agent has a goal location somewhere around the periphery of the grid environment that it is attempting to reach. This is not a requirement of Sinatra, but was chosen so that the Sinatra simulation may loosely represent a warehouse environment. The Sinatra agents travel around the ‘warehouse’ in the form of the grid environment, ostensibly collecting and delivering items to human or other users that are outside the environment. Therefore, the goal locations around the periphery of the environment can be thought of as pick-up or drop-off locations for the agents to reach.
In early versions of Sinatra the agents were assumed to be more primitive and were implemented using teleo-reactive (TR) programs to reach their goals. Introduced by Nilsson, a TR program is

“an agent control program that directs the agent toward a goal (hence teleo) in a manner that takes into account changing environmental circumstances (hence reactive)” [Nil94] p.140.

The low-level representation of agent perceptions and actions typically used by TR programs, however, significantly complicated the implementation of behaviours for the agents. For example, as Sinatra agents operate within a grid environment, we often want to have behaviours for these agents that are influenced by the agent’s location and orientation within the grid. However, at the low-level typical of a TR program, the agent does not perceive its position within the grid directly. Instead, the agent is able to perceive only its immediate surroundings and is (often) unable to perceive its orientation.

In order to develop behaviours to control a TR agent, the TR programs must be formulated without reference to specific grid locations and directions. Alternatively, additional TR perception rules must be included to allow the agent to determine its current location and orientation based on a record of successful movements from a known initial position. Therefore, the directives used to control a TR agent must be written at a very low-level and the formulation of these directives quickly becomes tedious.

For simplicity, we instead chose to use a more high-level approach. We assume that the Sinatra agents are equipped with sensors that enable them to directly perceive their location and orientation within the grid environment. For example, a GPS sensor and a compass can provide this information to the agent. These sensors also allow an agent to determine additional information about its environment, such as the direction to its goal location.

This assumption does not preclude TR agents from being simulated in Sinatra, as the higher level directives used by Sinatra agents can still be translated into low-level perceptions and TR rules. However, as the translation would be rather tedious, we have chosen to leave out this low-level representation so that it does not distract from the main purpose of the directives that are being implemented.

Figure 3.3 shows the Sinatra agent architecture based on the design points outlined above. These are the basic features of a Sinatra agent; however, it is intended that additional capabilities can be added to this agent architecture as required. We now describe the Sinatra agent perceptions, actions and memory in more detail. We then turn to the nature of the agent reasoning process, implemented using the behaviour method, which is the focus of the rest of this chapter.
3.2.1 Agent perceptions

Each Sinatra agent occupies a single grid location within the grid environment. We assume that a Sinatra agent is able to perceive its current location within the grid. This information is made available to the agent in the form of the $x$, $y$ coordinates of its position within the environment. Using this perception capability, the agent is able to identify when it has reached its goal location, as long as the goal is also described to the agent in terms of its $x$, $y$ position within the grid.

A cardinal direction system (north, east, south, west) is used to describe the orientation of Sinatra agents and the relative locations of different grid cells. The top of the simulation is considered to be north and the bottom south, the right side of the simulation is east and the left side is west. The Sinatra agent is able to perceive the direction it is facing and is also able to combine these directions with the perception of its current grid location, allowing the agent to perceive in which direction or directions its goal location can be found. For example, a goal that is to the north-east of the agent’s current location will be perceived by the agent as simultaneously a goal to the north and a goal to the east.
As well as turning to face one of the cardinal directions, the agent is also able to move forwards into the adjacent grid location that it is facing. We suppose that the agent is able to perceive the four grid locations that it can move to from its current position. For these four adjacent locations, the agent is able to determine whether the grid location is empty or is obstructed in some manner. The obstruction may be another agent, a wall or an unspecified obstacle within the domain. The agent is unable to perceive what the obstruction is, only that the grid location is obstructed.

A perception capability is implemented in the Sinatra agent code as a perception method. Table 3.1 describes the perceptions methods that are available to every Sinatra agent. Further perception methods can be added when they are required. However, these basic agent perceptions are sufficient to allow the agent to determine where it can move to within the grid environment, to move safely around the environment and to identify when it has reached its goal location.

<table>
<thead>
<tr>
<th>Perception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>Returns the current x, y position of the agent within the grid environment.</td>
</tr>
<tr>
<td>atGoal</td>
<td>True if location is the same as the x, y position recorded as the agent’s goal location.</td>
</tr>
<tr>
<td>direction</td>
<td>Returns the current orientation of the agent. The perception returned by this method will be one of north, east, south or west.</td>
</tr>
<tr>
<td>gnorth, geast, gsouth and gwest</td>
<td>True if the agent’s goal location is found to the north, east, south or west of the agent’s current location. Only one of gnorth and gsouth can be true at any time. If neither are true then the agent is in the same row as its goal location. Similarly for geast and gwest.</td>
</tr>
<tr>
<td>north, east, south and west</td>
<td>Allows the agent to perceive the adjacent grid locations in these directions. Using these methods, the agent can determine if a location is empty or obstructed in some manner.</td>
</tr>
</tbody>
</table>

Table 3.1: The perception methods of the Sinatra agents.

We refer to an agent’s perception method as a high-level perception method if it is implemented in terms of perception methods that have already been defined. For example, atGoal is a high-level perception method that is implemented using the results of other perception methods that are defined in Table 3.1.

In addition to these perceptions methods, a Sinatra agent is also able to perceive whether the action it is attempting to execute has been successful. For example, whether the agent has been able to move into an adjacent grid location. The success of each particular action can be determined by the agent by monitoring whether the appropriate perceptions have changed as expected.
3.2.2 Agent actions

The basic Sinatra agent actions are the ability to turn on the spot and to move forwards. In practice, we assume that a Sinatra agent is only able to turn 90° to the left or to the right. Therefore, in order for the Sinatra agent to turn to face the opposite direction, two turn left or two turn right actions will be required. Turning will change the direction of the agent, whilst its location will remain unchanged.

The Sinatra agent is also able to move forwards into an adjacent grid location. An agent is unable to move into a grid location that is already occupied by another agent, is beyond the perimeter of the grid environment or is obstructed in some other way. Therefore, the agent will first perceive the grid location in front of it to check that the location is empty before it attempts to move. This is done using the north, east, south and west perception methods shown in Table 3.1.

Having determined that it is facing an empty grid location, however, a Sinatra agent may not always be able to move into this new location. This can occur when two agents attempt to move into the same empty grid location during the same time step. We have specified that only one agent may occupy a grid location at any one time. In this situation, one of the agents will be able to move successfully into the empty location, while the other agent will remain in its original location. At the implementation level, the agent that is able to move is the agent that holds the lock on the contested grid location. Therefore, the move forwards action has the possibility of failing, but if executed successfully, the move forwards action will change the location of the agent.

The Sinatra agents are also able to perform a wait action, where no other action is performed by the agent and its position in the grid environment will remain unchanged. This action is also the default or null action performed by an agent when it has no other action to perform.

An action capability is implemented in the Sinatra agent code as an action method. Table 3.2 describes the actions that are available to every Sinatra agent. Further action methods can be added when required in order to extend the agent’s repertoire of actions. However, these basic action methods are sufficient to allow the agent to move freely around the grid environment.

Apart from the moveForwards action method, which has the chance of failure due to the actions of other agents, the action methods shown in Table 3.2 will always succeed. We do not include the potential for actions to fail due to the agents ‘slipping’ or other unexpected mishaps. Actions with a percentage chance of unexpected failure are introduced for the robot rugby domain used in Chapter 7.

We refer to an agent action method as a high-level action method if it is implemented in terms of action methods that have already been defined. For example, an action method moveDiagonally, if implemented in terms of the turnLeft,
<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
<th>Possible outcomes</th>
</tr>
</thead>
<tbody>
<tr>
<td>turnLeft and</td>
<td>Directs the agent to turn 90° to the left and 90° to the right respectively.</td>
<td>Always succeeds.</td>
</tr>
<tr>
<td>turnRight</td>
<td></td>
<td></td>
</tr>
<tr>
<td>moveForwards</td>
<td>Directs the agent to move forwards into the adjacent grid location.</td>
<td>Movement will succeed unless the grid location becomes occupied by another agent.</td>
</tr>
<tr>
<td>wait</td>
<td>The agent intentionally performs no action during this time step.</td>
<td>Always succeeds.</td>
</tr>
</tbody>
</table>

Table 3.2: The action methods of the Sinatra agents.

The `turnRight` and `moveForwards` actions, is a possible example of a high-level action method. We have chosen that the ability to move diagonally is not part of the basic action methods available to the Sinatra agents. Again however, this ability is added to the Sinatra agents for the robot rugby domain used in Chapter 7.

### 3.2.3 Agent memory

In addition to basic perception and action capabilities, the Sinatra agents also require a basic internal memory. This is used to store information required by the agent and that cannot be perceived by observation of the environment. Table 3.3 shows the facts stored by every Sinatra agent as implemented in the experiments that follow. This information requires only a few bytes to store. Other items could also be recorded in the agent’s internal memory, such as a record of actions performed so far.

<table>
<thead>
<tr>
<th>Information</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agentID</td>
<td>The identification number of the agent.</td>
</tr>
<tr>
<td>goalLocation</td>
<td>Records the x, y position of the agent’s goal location within the grid environment.</td>
</tr>
</tbody>
</table>

Table 3.3: The memory capabilities of the Sinatra agents.

### 3.2.4 Sinatra agents as mobile robots

It is intended that the behaviours developed using Sinatra should be able to instruct physical robots to act in the same way as the agents observed in the Sinatra simulation. Therefore, the perception and action capabilities of the Sinatra agents should mirror the potential capabilities of physical mobile robots.

Most of the perception capabilities of the Sinatra agent focus on the position of the agent within the environment. A compass sensor can be used by a physical robot
to determine its orientation. In order to determine the robot’s current location, a form of global positioning system (GPS) or indoor positioning system (IPS) may be suitable. Alternatively, the robot can be implemented to keep track of its current position based on monitoring (and recording) the actions that it has successfully performed and keeping a record of its start location. The Sinatra agents are also able to perceive the adjacent grid locations. This can be achieved for a physical robot by using light sensors or ultrasonic sensors.

The action capabilities of the Sinatra agent can easily be achieved by mobile robots. The Sinatra agents are implemented to turn exactly 90° as this makes minimal assumptions about the physical mobile robot capabilities. Similarly, the Sinatra agents are implemented to move forwards by exactly one grid location. If Sinatra is used to simulate more sophisticated physical devices, then these restrictions can be removed from the Sinatra simulation.

### 3.3 Behaviour method

We now focus on the method by which the Sinatra agents are implemented to determine their actions. We refer to this process as the *behaviour method*.

In order for agents to determine their own actions, each agent will follow one or more *behaviours*. A behaviour consists of a prioritised set of *directives*, specified by the system designer, which instruct an agent to act in a certain manner. A behaviour may be designed to allow an agent to achieve a particular goal or goals, to allow an agent to act in accordance with a norm or set of norms, or to achieve both of these functions simultaneously. Therefore, a behaviour is a versatile implementation mechanism that allows the system designer to configure an agent to act in a certain way.

One example of a behaviour is a set of directives designed to allow an agent to travel from its current location to its goal location. We will refer to this behaviour as the *simple goal-directed navigation* behaviour. In general, this behaviour will direct the agent to turn towards its goal and then to move forwards until it has reached this location. The behaviour will not take into account any potential obstacles that the agent may encounter, specifically the other agents in the domain. Therefore, this behaviour is a *simple goal-directed navigation*.

In this section, we describe the behaviour method and demonstrate its use by means of the *simple goal-directed navigation*. We begin by describing how the system designer creates a *behaviour definition* as a set of prioritised directives. Instantiated with an agent state, the behaviour definition identifies the actions and behaviours for an agent to perform in this state. A translation process is described that converts prioritised directives to an extended logic program without priorities. An answer set solver can then be used to identify the actions and behaviours from a *behaviour*.
Finally, we describe the process for automatically generating the answer sets for all possible behaviour instantiations and using these to construct a state-action table.

3.3.1 Behaviours

An agent uses a behaviour in order to determine how it should act. The agents are autonomous and so they must rely on their own knowledge and observations when making this decision. The information that is available to the agent to base its decisions on is referred to as the agent’s current state.

In order to consult a behaviour, the agent must determine its current state. The current state of an agent is made up of a set of facts that hold for the agent at the current time step. An agent is able to determine its current state by combining the results of its perception methods and the information stored in the agent’s internal memory. The current state for a Sinatra agent will, therefore, contain some or all of the following information, as appropriate for each behaviour.

- The agent’s current goal location.
- The agent’s current orientation and location within the grid environment.
- The relative direction of the agent’s goal from its current location, including whether the agent has reached its goal location.
- The agent’s perceptions of the four adjacent grid locations that it can move to from its current location.

Having identified the agent’s current state, the agent will use a behaviour to determine which action to execute. The rules that make up a behaviour map properties of the agent’s current state to an action for the agent to perform. We refer to these rules as directives.

A behaviour combines directives about how an agent should act in certain circumstances with a priority order between these directives to manage situations where multiple directives can be applied. In addition, the system designer will develop a behaviour with a particular purpose in mind; for example, to achieve a specific agent goal or to allow the agent to interact with other agents in a certain manner.

Behaviour definition

A behaviour definition consists of an ordered set of directives and a set of formulæ that specify constraints on the actions that can be executed.

A directive $d$ is a rule of the form

$$
\text{action/behaviour} \leftarrow \text{condition}
$$
where condition and action/behaviour are made up of tokens that are understood by
the agent.

The condition is a possibly empty conjunction of literals that represent facts that
may hold in the agent’s current state. In the behaviours developed in this thesis, we
have only required a set of atoms to define the condition of a directive. However,
we do not wish to assume that this will always be the case and so we define the
condition as a conjunction of literals.

The action/behaviour token is a single atom that represents a primitive action
method that the agent is capable of executing or a behaviour that the agent has
been implemented to follow. An action/behaviour cannot be part of the condition,
as only facts are permitted in the condition.

A directive, therefore, describes a particular trigger in the agent’s state for when
the specified action should be performed or behaviour should be followed. This
means that directives can be used to allow agents to switch between using different
behaviours and that behaviours can be used in a nested manner in order to determine
how the agent should act. We will discuss this aspect of behaviours in Section 3.5.
However, a token in a directive that refers to an action and a token that refers to a
behaviour are indistinguishable to the behaviour method and so in this section we
describe features that are common to all behaviours.

The conditions of the directives within a behaviour are not necessarily mutually
exclusive. This means that an agent’s current state may contain facts that satisfy
the condition of more than one directive within the behaviour. For example, using
the set of tokens c1, c2 and c3 to represent certain facts that may hold in the agent’s
current state, suppose that D is the following set of directives.

\[
\begin{align*}
  d_1 & \quad \text{action1} \leftarrow c_1, c_2, c_3 \\
  d_2 & \quad \text{action2} \leftarrow c_1, c_2 \\
  d_3 & \quad \text{action3} \leftarrow c_1
\end{align*}
\]

The name \(d_n\) associated with a directive is used to allow us to refer to the individual
directives in \(D\). These names are not a required feature of the behaviour definition.

In some situations we may want the agent to be able to perform multiple actions
concurrently. However, certain combinations of actions may be impossible or should
not be executed together for other reasons. For example, an agent cannot simultane-
ously turn left and turn right. Constraint formulae are used to specify combinations
of actions that cannot be executed simultaneously. These take the general form

\[
\leftarrow \text{action}_1, ..., \text{action}_m
\]

This constraint specifies that the actions \(\text{action}_1\) to \(\text{action}_m\) cannot be executed
simultaneously, but that other subsets of these actions are permitted together. For
the constraint formulae used in behaviours, we will normally restrict actions in pairs, specifying that only one of these actions can be executed at a time.

A priority order between the directives is used to resolve the issue of multiple directives being applicable in the agent’s current state when their associated actions cannot be executed concurrently. We require the priority order to be total. To define the priority order, the system designer specifies the directives in descending priority order. In the case of conflict, we specify that the higher priority directive will determine the action or the behaviour that the agent should execute or follow respectively.

A behaviour definition is a tuple \((W_C, D, \prec)\) consisting of an ordered set of directives \((D, \prec)\) and a set of constraint formulae \(W_C\). The priority relation \(\prec\) is a strict total order over the set of directives \(D\). For any directives \(d, d' \in D\), \(d \prec d'\) means that directive \(d\) has a higher priority than \(d'\).

**Behaviour instantiation**

A behaviour definition is applied by an agent in the context of the facts that hold in the agent’s current state. Therefore, in order to use a behaviour definition, the behaviour must be instantiated with a set of state facts \(W_S\).

A behaviour instantiation is a tuple \((W_S, W_C, D, \prec)\) consisting of a behaviour definition \((W_C, D, \prec)\) and a set of state facts \(W_S\). The state facts are a consistent set of literals that represent facts about the agent’s current state. The range of possible agent states for a behaviour are described in terms of a set of facts. The state facts of a behaviour instantiation are a consistent set of these facts, but not necessarily a complete interpretation.

**Identifying the preferred answer set from a behaviour instantiation**

We now present a method specifying how priorities between directives are handled in a behaviour. The definitions together describe a prescriptive approach for finding the preferred answer sets given a behaviour instantiation, by identifying the consistent set of actions and behaviours associated with the highest priority triggered directives.

We have specified that the action/behaviour of a directive is an atom while the condition contains only literals that represent facts about the current state of the agent. Finally, a directive \(d\) contains no instances of negation by failure \(\text{not}\) and so \(\text{body}^- (d) = \emptyset\). Therefore, a directive \(d\) is a basic logic program rule of the form

\[
\text{head}(d) \leftarrow \text{body}^+(d)
\]

This means that for any set of literals \(X\) and set of directives \(D\), the reduct \(D^X = D\).
We have also specified that directives cannot have an action/behaviour as part of the condition. Therefore, for a set of directives $D$, we have the following property

$$\{\text{head}(d) \mid d \in D\} \cap \{\text{body}^+(d) \mid d \in D\} = \emptyset$$

We could present our definitions in a similar style to that used by Horty when describing his application of priorities to ordered default theories [Hor07]. However, due to the form of directives that we have specified, we are instead able to use a significantly simpler set of definitions. In particular, as we have specified that directives cannot have an action/behaviour as part of the condition, there can be no chaining of directives and so we only need to consider the application of each rule once.

This means that a directive is triggered iff its condition holds in $W_S$.

**Definition 3.1. (Triggered directive)** For a behaviour instantiation $(W_S, W_C, D, \prec)$, a directive $d \in D$ is triggered, $d \in \text{Triggered}(W_S, D)$, iff

$$\text{body}^+(d) \subseteq W_S$$

A directive is defeated if it is inconsistent with a higher priority triggered directive.

**Definition 3.2. (Defeated directive)** For a behaviour instantiation $(W_S, W_C, D, \prec)$, a directive $d \in D$ is defeated, $d \in \text{Defeated}(W_S, W_C, D, \prec)$, iff there exists a set of directives $S$ where $S \subseteq \text{Triggered}(W_S, D)$ and $S \prec \{d\}$ such that

$$\text{Cn}(\{W_C \cup \text{head}(S) \cup \text{head}(d)\}) = \text{Lit}(\{W_C \cup \text{head}(S) \cup \text{head}(d)\})$$

In other words, for a logic program $P = \{W_S \cup W_C \cup S \cup d\}$, the set of literals found by $\bigcup_{n \geq 0} T_P \uparrow_1$ will contain a complementary pair of literals $A$ and $\neg A$.

A directive is binding iff it is triggered but not defeated.

**Definition 3.3. (Binding directive)** For a behaviour instantiation $(W_S, W_C, D, \prec)$, a directive $d \in D$ is binding, $d \in \text{Binding}(W_S, W_C, D, \prec)$, iff

$$d \in \text{Triggered}(W_S, D) \land d \notin \text{Defeated}(W_S, W_C, D, \prec)$$

The binding directives are those used to generate the preferred answer sets of the behaviour instantiation $(W_S, W_C, D, \prec)$. The set of binding directives of a behaviour instantiation are found by considering the directives in priority order.

**Definition 3.4. (Set of binding directives)** Let $(W_S, W_C, D, \prec)$ be a behaviour instantiation. Then $S$ is the set of binding directives in $D$ iff

$$S = \bigcup_{i \geq 0} S_i$$

where
\[ S_0 = \emptyset \]
\[ S_{i+1} = \begin{cases} 
S_i \cup \{d_{i+1}\} & \text{if } d_{i+1} \in \text{Triggered}_{(W_S, D)} \text{ and } \\
Cn(\{W_S \cup W_C \cup S_i \cup d_{i+1}\}) \text{ is consistent} & \text{otherwise} 
\end{cases} \]

For a set of binding directives \( S \), the answer sets of the logic program \( \{W_S \cup W_C \cup S\} \) are the preferred answer sets of the behaviour instantiation \((W_S, W_C, D, \prec)\).

**Definition 3.5. (Preferred answer set)** Let \((W_S, W_C, D, \prec)\) be a behaviour instantiation and \( X \) be a set of literals. Let \( S \) be the set of binding directives in \((W_S, W_C, D, \prec)\). Then \( X \) is the preferred answer set of \((W_S, W_C, D, \prec)\) iff

\[ X = Cn(\{W_S \cup W_C \cup S\}) \]

or equivalently

\[ X = \bigcup_{n \geq 0} T_{\{W_S \cup W_C \cup S\}} \uparrow 1 \]

Using this method, we are able to determine the actions and behaviours that the behaviour instantiation \((W_S, W_C, D, \prec)\) directs an agent in the state \( W_S \) to execute and follow respectively. These actions and behaviours are contained in the preferred answer sets of the behaviour instantiation.

In fact, as we have required that the priority order for a behaviour definition is a total order, we find that a behaviour instantiation will have at most one preferred answer set.

**Lemma 3.1.** Let \((W_S, W_C, D, \prec)\) be a behaviour instantiation and let \( S \) be the binding directives in \((W_S, W_C, D, \prec)\). Then \( \{W_S \cup W_C \cup S\} \) has at most one preferred answer set.

**Proof.** Let \( X_1 \) and \( X_2 \) be two answer sets of \( \{W_S \cup W_C \cup S\} \), \( X_1 \neq X_2 \), and assume both are consistent. Consider the highest priority directive \( d \in D \) such that \( d \) is applied in one answer set, say \( X_1 \), and not in the other. As \( d \) is applied in \( X_1 \) we know that \( d \in \text{Triggered}_{(W_S, D)} \). As \( d \) is not applied in \( X_2 \) then \( d \) must be defeated due to a higher priority triggered directive \( d' \) that is applied in \( X_2 \). But since \( d \) is the highest priority directive that is applied in exactly one of \( X_1 \) and \( X_2 \), then \( d' \) must also be applied in \( X_1 \). Therefore, \( d \) cannot be applied in \( X_1 \), leading to a contradiction.

A behaviour instantiation \((W_S, W_C, D, \prec)\) may direct an agent to perform multiple actions and to follow multiple behaviours. The constraint formulae ensure that when multiple actions and behaviours are found in the preferred answer set, that there are no conflicts between these actions and behaviours. Therefore, if a behaviour instantiation specifies multiple actions and behaviours for the state \( W_S \), then these
actions and behaviours are to be performed concurrently. A behaviour that can
direct an agent to perform multiple actions during the same time is implemented
for the hide-and-seek domain in Chapter 7.

**Simple goal-directed navigation definition**

We describe the development of the behaviour definition for the *simple goal-directed
navigation*. We then demonstrate how to find the actions prescribed by this behaviour
for a particular state of the agent.

The *simple goal-directed navigation* will be used by Sinatra agents to allow them to
move to their goal location. The goal location of a Sinatra agent is a specific location
around the periphery of the grid environment, which is intended to represent a pick-
up or drop-off location within a warehouse. During each time step of the simulation,
the agent has the ability to turn 90° to the left or to the right, or to move forwards
by one grid location in the direction that it is facing.

The *simple goal-directed navigation* definition uses a basic strategy to allow the
agent to reach its goal location. For each time step, if the agent is facing in the
same direction as its goal, for example *gnorth* is true and the *direction*
perception of the agent returns that the agent is facing north, then the agent is directed to
move forwards. If the agent is not facing in the direction of its goal, then the agent
is directed to turn left or right as appropriate until it is facing its goal.

In other words, the *simple goal-directed navigation* instructs the agent to turn
towards its goal and then to move forwards until it is in line with its goal. If at
this point the agent has not reached its goal, meaning that *atGoal* is false, then
one of *gnorth*, *geast*, *gsouth* or *gwest* will still be true. In this case the agent
is again instructed to turn towards its goal and to travel the remaining distance
to this location. The agent reassess which action to perform based on the *simple
goal-directed navigation* during each time step.

This strategy will be written as the set of directives $D$ in a behaviour definition
($W_C, D, \prec$). It is clear from this description of the *simple goal-directed navigation*
that the behaviour definition does not take into account other Sinatra agents that
may exist in the environment. Instead, the behaviour definition focuses only on the
actions of the agent that is following the behaviour, taking the agent on the most
direct route to its goal location.

For this behaviour, we have chosen that the turn and movement actions cannot
be performed concurrently. Therefore, constraint formulae are required as part of
the *simple goal-directed navigation* to constrain the permitted concurrent actions of
the agents.

The behaviour definition for the *simple goal-directed navigation* based on this spec-
ification becomes
The simple goal-directed navigation definition contains the ordered set of directives \((D, \prec)\), followed by the set of constraint formulae \(W_C\). The directives \(D\) are written in descending priority order. Hence, the priority order \(\prec\) is a total order. The names of the directives, \(d_n\), are there to aid the following discussion and are not a required part of the behaviour definition.

To instantiate this behaviour definition, let the agent’s current state include that the agent is facing west, with its goal location somewhere to the north-west of its current location. The other agent perceptions that could form the agent’s current state are not required by the simple goal-directed navigation. Therefore, let the state facts \(W_S\) be

\[
\text{west gnorth gwest}
\]

Using the behaviour instantiation \((W_S, W_C, D, \prec)\), the preferred actions and behaviours for the agent to execute can be determined. These actions and behaviours are found in the preferred answer set of the behaviour instantiation. This process is referred to as the behaviour method.
The preferred answer set is found by first identifying the binding set of directives as described in Definition 3.4. We start with the empty set of directives.

\[ S_0 = \emptyset \]

By considering the directives \( D \) in priority order, we identify those that are triggered and are consistent with the binding directives that have come before. If this is the case then the directive is added to the set \( S \), otherwise the directive is ignored.

The first triggered directive \( d \), where \( \text{body}^+(d) \subseteq W_S \), is \( d_4 \). The current set of binding directives is the empty set and so we have that \( Cn({WC} \cup \emptyset \cup \text{head}(d_4)) \) is consistent. Therefore, the directive \( d_4 \) is added to the binding set of directives.

\[ S_4 = \{d_4\} \]

The next triggered directive is \( d_6 \) but this directive is defeated because \( d_6 \) is inconsistent with the constraint formulae and the directive \( d_4 \).

\[ Cn({WC} \cup \text{head}(S_4) \cup \text{head}(d_6)) = Lit({WC} \cup \text{head}(S_4) \cup \text{head}(d_6)) \]

Therefore, \( d_6 \) is not added to the set of binding directives. Similarly for the directives \( d_7 \) and \( d_{16} \).

Finally, we have considered all of the directives \( D \) in the behaviour instantiation \((W_S, WC, D, \prec)\) and constructed the set of binding directives \( S \).

\[ S = \{d_4\} \]

Having identified the set of binding directives \( S \), we use Definition 3.5 to find the preferred answer set of \((W_S, WC, D, \prec)\) by finding the answer set of \((W_S \cup WC \cup S)\), which represents the following logic program.

\[
\begin{align*}
\text{forward} & \leftarrow \text{gwest, west} \\
& \quad \leftarrow \text{forward, left} \\
& \quad \leftarrow \text{forward, right} \\
& \quad \leftarrow \text{left, right} \\
\text{west} \\
\text{gnorth} \\
\text{gwest} \\
\end{align*}
\]

The single answer set of \((W_S \cup WC \cup S)\) is

\[
\text{west gnorth gwest forward}
\]
This means that, for this current state of the agent, the simple goal-directed navigation will return the token *forward*. This token is understood by the Sinatra agent to refer to the action method `moveForwards`. Therefore, an agent that is facing west with its goal to the north-west while following the simple goal-directed navigation, should attempt to execute the action `moveForwards`.

Table 3.4 shows the classification of the behaviour method using the classification system of Delgrande et al. [DSTW04] (Section 2.4.2). The behaviour method is a fully prescriptive approach, meaning that the directives must be applied in priority order, as long as they are triggered.

<table>
<thead>
<tr>
<th>Host system</th>
<th>Logic programs under answer sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meta-level vs object-level</td>
<td>Meta-level priorities</td>
</tr>
<tr>
<td>Static vs dynamic</td>
<td>Static priorities</td>
</tr>
<tr>
<td>Properties of <code>&lt;</code></td>
<td>Strict total order</td>
</tr>
<tr>
<td>Priority relation</td>
<td>Priorities between rules</td>
</tr>
<tr>
<td>Prescriptive vs descriptive</td>
<td>Prescriptive</td>
</tr>
<tr>
<td>Preferred results</td>
<td>Preferred results found using the total order <code>&lt;</code>. Preferred result found directly from <code>&lt;</code> without reference to the priority-free rules.</td>
</tr>
</tbody>
</table>

Table 3.4: Classification of the behaviour method.

### 3.3.2 Behaviour translation

We are also able to find the preferred result of a behaviour instantiation by using a translation framework based on that of Delgrande et al. [DST03], as outlined in Section 2.4.3. Their general framework allows a prioritised extended logic program \((\Pi, <)\) to be translated into an extended logic program without priorities \(T(\Pi, <)\). The answer sets of the translated logic program \(T(\Pi, <)\) correspond to the preferred results of the prioritised extended logic program \((\Pi, <)\).

The extended logic program without priorities \(T(\Pi, <)\) incorporates the priority order between the logic program rules of \((\Pi, <)\) by using tags to control the execution of the translated logic program rules. Therefore, the extended logic program without priorities \(T(\Pi, <)\) is referred to by Delgrande et al. as a tagged logic program.

In order to develop a translation of a behaviour instantiation \((W_S, W_C, D, \prec)\), the prioritised set of directives \((D, \prec)\) will be translated using our *behaviour translation framework* into an extended logic program without priorities \(T_B(D, \prec)\). The extended logic program \(T_B(D, \prec)\) is a tagged extended logic program.

A translation of the constraint formulae \(W_C\) and the agent’s current state facts \(W_S\) will then be added to the translated logic program \(T_B(D, \prec)\) to form the final translation \(T'_B(W_S, W_C, D, \prec)\). The answer sets of the translated logic program \(T'_B(W_S, W_C, D, \prec)\) will correspond to the preferred answer sets of the original be-
haviour instantiation \((W_S, W_C, D, \prec)\).

Using the behaviour translation framework, an answer set solver can be used to compute the answer sets of the translated logic program \(T_B'(W_S, W_C, D, \prec)\). The actions and behaviours that appear in these answer sets correspond to the actions and behaviours in the original behaviour instantiation. We use the answer set solver \textsc{Clasp}, developed by Gebser et al. [GKNS07], to find the answer sets of \(T_B'(W_S, W_C, D, \prec)\). \textsc{Clasp} is a state-of-the-art answer set solver for extended logic programs.

**Developing the behaviour translation framework**

To explain the behaviour translation framework, we begin by considering an example behaviour definition \((W_C, D, \prec)\).

\[
\begin{align*}
  d_1 \ & \ action1 \leftarrow c_1, c_2, c_3 \\
  d_2 \ & \ action2 \leftarrow c_1, c_2 \\
  d_3 \ & \ action1 \leftarrow c_1 \\
\end{align*}
\]

\[\leftarrow \text{action1, action2}\]

The constraint formula specifies that \text{action1} and \text{action2} cannot be performed together. Recall that the directives are written in descending priority order and so we have \(d_1 \prec d_2 \prec d_3\).

We are able to include the priority order \(\prec\) within the directives of the behaviour definition by looking for directives that contain conflicting heads and defining tag literals to handle the priorities between these directives. We begin by using the tag literal \(ap(d_n)\) defined by Delgrande et al. [DST03] to specify that a directive \(d_n\) has been applied.

\[
\begin{align*}
  d_1 \ & \ action1 \leftarrow ap(d_1) \\
  ap(d_1) \ & \leftarrow c_1, c_2, c_3 \\
  d_2 \ & \ action2 \leftarrow ap(d_2) \\
  ap(d_2) \ & \leftarrow c_1, c_2, \text{not } ap(d_1) \\
  d_3 \ & \ action1 \leftarrow ap(d_3) \\
  ap(d_3) \ & \leftarrow c_1, \text{not } ap(d_2) \\
\end{align*}
\]

\[\leftarrow \text{action1, action2}\]

However, for the directives in a behaviour definition, we know that a directive is applied when it is binding, which occurs when a directive is triggered but not defeated. Therefore, we define \(\text{binding}(d_n), \text{trig}(d_n)\) and \(\text{defeat}(d_n)\) tags and use these to describe when a directive is applied. We demonstrate these tags for the directive \(d_2\).
A directive is binding when it is triggered but not defeated.

\[
\text{binding}(d_2) \leftarrow \text{trig}(d_2), \text{not defeat}(d_2)
\]

When a directive is binding, we know that it is used to generate the preferred answer set of the behaviour instantiation and, therefore, the directive is applied. This replaces the \( ap(d_n) \) tag.

\[
\text{action2} \leftarrow \text{binding}(d_2)
\]

A directive is triggered when the literals in its body hold.

\[
\text{trig}(d_2) \leftarrow c_1, c_2
\]

A directive is defeated if there is a higher priority triggered directive that conflicts with the head of this directive. For the directive \( d_2 \), this occurs when the directive \( d_1 \) is applied.

\[
\text{defeat}(d_2) \leftarrow \text{binding}(d_1)
\]

Using these new tag literals, the directives for the original example behaviour definition can be rewritten to include the priority order \( \prec \).

\[
\begin{align*}
\text{action1} & \leftarrow \text{binding}(d_1) \\
\text{binding}(d_1) & \leftarrow \text{trig}(d_1), \text{not defeat}(d_1) \\
\text{trig}(d_1) & \leftarrow c_1, c_2, c_3 \\
\text{action2} & \leftarrow \text{binding}(d_2) \\
\text{binding}(d_2) & \leftarrow \text{trig}(d_2), \text{not defeat}(d_2) \\
\text{trig}(d_2) & \leftarrow c_1, c_2 \\
\text{defeat}(d_2) & \leftarrow \text{binding}(d_1) \\
\text{action1} & \leftarrow \text{binding}(d_3) \\
\text{binding}(d_3) & \leftarrow \text{trig}(d_3), \text{not defeat}(d_3) \\
\text{trig}(d_3) & \leftarrow c_1 \\
\text{defeat}(d_3) & \leftarrow \text{binding}(d_2) \\
\end{align*}
\]

\[
\leftarrow \text{action1}, \text{action2}
\]

This is a simple strategy for dealing with priorities between the directives in the example behaviour definition. However, we want to be able to generalise this approach to any set of directives in a behaviour definition. In particular, this requires us to be able to define when a directive is defeated in terms of the directive itself, rather than in terms of any other directives in the behaviour.
A directive is defeated when an atom holds in the answer set of the behaviour instantiation that conflicts with the head of this directive. These conflicts between atoms, which represent actions or behaviours, are defined by the constraint formulae in the behaviour definition.

In order to express this conflict in a manner that can be used by the translation, we translate the constraint formulae in the behaviour definition.

\[
\neg \text{action1} \leftarrow \text{action2} \\
\neg \text{action2} \leftarrow \text{action1}
\]

Therefore, we are able to define a directive as being defeated when its head is conflicted.

\[
\text{defeat}(d_2) \leftarrow \neg \text{action2}
\]

By defining a directive to be defeated in this way, however, we must now be careful to ensure that the directives in the translated logic program are only applied in the correct priority order. Therefore, we introduce \(\text{ok}(d_n)\) tag literals, similar to those used in the translation framework of Delgrande et al. [DST03]. The \(\text{ok}(d_n)\) tag literals are used to control the order that directives are applied.

For the two directives \(d_1\) and \(d_2\), it is only safe to start to consider applying the translated logic program rules associated with \(d_2\) when we have finished considering \(d_1\). We will stop considering \(d_1\) once we have applied \(d_1\), have identified that \(d_1\) is defeated or when we have come to consider \(d_1\) and find that it is not triggered.

\[
\text{ok}(d_2) \leftarrow \text{action1} \\
\text{ok}(d_2) \leftarrow \text{defeat}(d_1) \\
\text{ok}(d_2) \leftarrow \text{ok}(d_1), \neg \text{trig}(d_1)
\]

The logic program rule for determining when a directive is defeated must also be modified to take into account the \(\text{ok}(d_n)\) tags. Initially, we suggest the following translation rule.

\[
\text{defeat}(d_2) \leftarrow \neg \text{action2}, \text{ok}(d_2)
\]

To simplify the translation, so that the translated logic program rules are defined similarly for each directive in the original behaviour definition, logic program rules defining \(\text{ok}(d_4)\) tag literals are included in the translation of directive \(d_3\), even though there is no directive \(d_4\). Similarly, an \(\text{ok}(d_1)\) tag is included in the defeated rule for the highest priority directive \(d_1\). Therefore, the fact literal defining \(\text{ok}(d_1)\) must also be included in the translation.

Based on this translation, however, we find that directives are able to be defeated by lower priority directives. This means that including the \(\text{ok}(d_n)\) tag literal in the
defeat\((d_n)\) definition is not sufficient to identify when a directive is defeated. To resolve this issue, we must introduce translation rules that allow us to identify when a directive would be defeated, rather than waiting until we find that the directive is defeated.

We introduce a new type of literal called a c tag. A c tag is a prefix tag attached to each literal \(L\), where \(L\) is the atom \(A\) forming the head of a directive in the behaviour definition or the complementary literal \(\neg A\). The c tag allows the consequences of rules to be considered without leading to complementary literals in the answer set, which would cause an incorrect result. The c tag means that the literals \(cL\) and \(\neg c\neg L\) are not complementary, while the literals \(L\) and \(\neg L\) are complementary.

Using the c tags, we are able to introduce a new logic program rule that is used to identify the consequences of applying a triggered directive, without immediately applying the directive itself. We say that the tag literal \(cL\) is used to identify a ‘committed’ literal \(L\). In this translation rule, we only allow a directive to be applied in this way if the directive is triggered, if it is the right time to consider this directive given the priority order and if the directive is not already inconsistent.

\[
c_\text{action2} \leftarrow \text{trig}(d_2), \text{ok}(d_2), \text{not } c_\text{\neg action2}
\]

The logic program rule for determining when a directive is defeated is now defined in terms of a committed literal.

\[
defeat(d_2) \leftarrow c_\text{\neg action2}, \text{ok}(d_2)
\]

Finally, we must modify the translation of the constraint formula to now refer to these committed literals and we reintroduce the original constraint formula for the non-tagged result of applying the directives.

\[
\leftarrow \text{action1}, \text{action2}
\]

\[
c_\text{\neg action1} \leftarrow c_\text{action2}
\]

\[
c_\text{action2} \leftarrow c_\text{action1}
\]

The translation of the original example behaviour definition becomes

\[
\text{ok}(d_1)
\]

\[
\text{action1} \leftarrow \text{binding}(d_1)
\]

\[
\text{binding}(d_1) \leftarrow \text{trig}(d_1), \text{not } \text{defeat}(d_1)
\]

\[
\text{trig}(d_1) \leftarrow c_1, c_2, c_3
\]

\[
\text{c_\text{action1}} \leftarrow \text{trig}(d_1), \text{ok}(d_1), \text{not } c_\text{\neg action1}
\]

\[
\text{defeat}(d_1) \leftarrow c_\text{\neg action1}, \text{ok}(d_1)
\]

\[
\text{ok}(d_2) \leftarrow \text{action1}
\]

\[
\text{ok}(d_2) \leftarrow \text{defeat}(d_1)
\]

\[
\text{ok}(d_2) \leftarrow \text{ok}(d_1), \text{not } \text{trig}(d_1)
\]
action2 ← binding(d₂)
binding(d₂) ← trig(d₂), not defeat(d₂)
trig(d₂) ← c₁, c₂
c₁ action2 ← trig(d₂), ok(d₂), not c¬action2
defeat(d₂) ← c¬action2, ok(d₂)
ok(d₃) ← action2
ok(d₃) ← defeat(d₂)
ok(d₃) ← ok(d₂), not trig(d₂)

action1 ← binding(d₃)
binding(d₃) ← trig(d₃), not defeat(d₃)
trig(d₃) ← c₁
c₁ action₁ ← trig(d₃), ok(d₃), not c¬action1
defeat(d₃) ← c¬action1, ok(d₃)
ok(d₄) ← action1
ok(d₄) ← defeat(d₃)
ok(d₄) ← ok(d₃), not trig(d₃)

← action1, action2

c¬action1 ← c₁ action2
c¬action2 ← c₁ action1

Using this translation of the original behaviour definition \((W_C, D, \prec)\), we introduce the state facts of a corresponding behaviour instantiation \((W_S, W_C, D, \prec)\). The state facts are a consistent set of facts about the agent’s current state. These facts are not influenced by the priority order over the directives in the behaviour. Therefore, the state facts are added directly to the translated logic program without any translation being applied to them. The answer set solver CLASP is used to determine the action to be carried out based on this behaviour instantiation.

For the behaviour instantiation with a set of state facts \(W_S = \{c₁\}\), the answer set for the behaviour translation contains the action action₁, corresponding to applying the directive \(d₃\). For the set of state facts \(W_S = \{c₁, c₂\}\), the answer set for the behaviour translation contains action₂, corresponding to applying the directive \(d₂\). Finally, for the set of state facts \(W_S = \{c₁, c₂, c₃\}\), the answer set for the behaviour translation contains action₁, this time corresponding to applying the directive \(d₁\).

Therefore, using this translation of the example behaviour definition, we are able to identify correctly when a directive is triggered, defeated and binding and we obtain a single preferred answer set for each of the three possible behaviour instantiations.
Behaviour translation framework

The tagged logic program that will form the translation of a behaviour instantiation will use the following tags.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$cL$</td>
<td>Where $L$ is a literal, such that $L$ is the atom $A$ forming the head of a directive or is the complementary literal $\neg A$. $cL$ identifies a ‘committed’ literal $L$.</td>
</tr>
<tr>
<td>$\text{trig}(d_n)$</td>
<td>To detect when a directive is triggered.</td>
</tr>
<tr>
<td>$\text{defeat}(d_n)$</td>
<td>To detect when a directive would be defeated.</td>
</tr>
<tr>
<td>$\text{binding}(d_n)$</td>
<td>To detect when a directive is binding.</td>
</tr>
<tr>
<td>$\text{ok}(d_n)$</td>
<td>To control the order that rules are applied.</td>
</tr>
</tbody>
</table>

Using these tag literals, we define the behaviour translation framework. For clarity, we break the definition of the translation into two parts, before presenting an overall behaviour translation definition. The first part concerns the translation of the prioritised set of directives within a behaviour instantiation, while the second part concerns the translation of the constraint formulae.

**Definition 3.6.** (Translation $T_B$ (part 1)) Let $(W_S, W_C, D, \prec)$ be a prioritised extended logic program over language $L$ defining a behaviour instantiation. Let $(D, \prec)$ be the prioritised set of directives of this behaviour instantiation. Let $d_n$ denote the name of the directive $d \in D$. Let $L^+$ be the language obtained from $L$ by adding for each $d \in D$ the atoms $\text{trig}(d_n)$, $\text{defeat}(d_n)$, $\text{binding}(d_n)$, $\text{ok}(d_n)$, and for each literal $A = \text{head}(d)$ the literals $cA$ and $c\neg A$.

For the prioritised set of directives $(D, \prec)$, the translated logic program $T_B(D, \prec)$ over $L^+$ is defined as

$$T_B(D, \prec) = \bigcup_{d \in D} \tau(d)$$

where the set $\tau(d)$ consists of the following rules, for each $d, d' \in D$ with $d \prec d'$.

- $\text{binding}_1(d): \text{head}(d) \leftarrow \text{binding}(d_n)$
- $\text{binding}_2(d): \text{binding}(d_n) \leftarrow \text{trig}(d_n), \text{not} \text{defeat}(d_n)$
- $\text{trig}_1(d): \text{trig}(d_n) \leftarrow \text{body}(d)$
- $\text{trig}_2(d): \text{chead}(d) \leftarrow \text{trig}(d_n), \text{ok}(d_n), \text{not} \overline{\text{chead}(d)}$
- $\text{def}(d): \text{defeat}(d_n) \leftarrow \overline{\text{chead}(d)}, \text{ok}(d_n)$
- $\text{ok}_1(d): \text{ok}(d_n) \leftarrow \text{head}(d)$
- $\text{ok}_2(d): \text{ok}(d_n) \leftarrow \text{defeat}(d_n)$
- $\text{ok}_3(d): \text{ok}(d_n) \leftarrow \text{ok}(d_n), \text{not} \text{trig}(d_n)$
The function of the different rules of the translation $\tau(d)$ can be explained as follows.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\textit{binding}_1(d)$</td>
<td>To add the literal $\textit{head}(d)$ to the answer set, which is the effect of $d$ being binding.</td>
</tr>
<tr>
<td>$\textit{binding}_2(d)$</td>
<td>To identify when a directive $d$ is binding.</td>
</tr>
<tr>
<td>$\textit{trig}_1(d)$</td>
<td>To identify when a directive $d$ is triggered.</td>
</tr>
<tr>
<td>$\textit{trig}_2(d)$</td>
<td>To add the literal $\textit{head}(d)$ to the set of committed literals, which is the effect of $d$ being triggered. Therefore the directive $d$ has been ‘committed’, in order to test its consequences.</td>
</tr>
<tr>
<td>$\textit{def}(d)$</td>
<td>To identify when a directive $d$ would be defeated.</td>
</tr>
<tr>
<td>$\textit{ok}_1(d)$, $\textit{ok}_2(d)$ and $\textit{ok}_3(d)$</td>
<td>To identify when the current directive $d$ has been applied, identified as defeated or identified as not being triggered respectively. Used to identify when it is safe to begin considering the next highest priority directive $d'$.</td>
</tr>
</tbody>
</table>

**Definition 3.7.** *(Translation $\mathcal{T}_B$ (part 2))* Let $(W_S, W_C, D, \prec)$ be a prioritised extended logic program defining a behaviour instantiation. Let $W_C$ be the constraint formulae of this behaviour instantiation, where each constraint formula is of the form

\[ \left\arrow A, B \right. \]

where $A$ and $B$ are atoms. Then the translation $\mathcal{T}_B(W_C)$ is defined as

\[ \mathcal{T}_B(W_C) = \bigcup_{f \in W_C} \tau(f) \]

where $\tau(f)$ consists of the following rules for each formula $f \in W_C$.

\[
\begin{align*}
\textit{commit}_1(f) : & \quad \overline{cA} \leftarrow \overline{cB} \\
\textit{commit}_2(f) : & \quad \overline{cB} \leftarrow \overline{cA}
\end{align*}
\]

The function of the different rules of the translation $\tau(f)$ can be explained as follows.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f$</td>
<td>The constraint formula $f$ itself. To constrain the actions and behaviours that can hold concurrently.</td>
</tr>
<tr>
<td>$\textit{commit}_1(f)$ and $\textit{commit}_2(f)$</td>
<td>To explicitly state the actions or behaviours that cannot hold when an answer set has committed to a particular directive. Used to identify when a directive would be defeated.</td>
</tr>
</tbody>
</table>
As we have already described, the c tag is a prefix tag attached to each literal L, where L is the atom A forming the head of a directive in the behaviour definition or is the complementary literal ¬A. The c tag allows the consequences of rules to be considered without leading to complementary literals in the answer set, which would cause an incorrect result. This allows us to identify when a directive is defeated.

Consider two directives d and d', where d ≺ d' and there is a constraint formula such that head(d) and head(d') cannot be true together. The c tag allows a defeated directive d' to be identified using the translation rules trig2(d) and def(d'), without allowing any lower priority directive to defeat the directive d incorrectly.

To complete the translation, the tag ok(d₁) must be included in the translation, along with the state facts WS. The state facts consist only of literals, which are added to the behaviour translation without modification.

By combining the translation of these individual elements, we obtain the overall behaviour translation definition, T_B'(WS, WC, D, ≺), of the behaviour instantiation (WS, WC, D, ≺) using the behaviour translation framework.

**Definition 3.8.** (Behaviour translation T_B') Let (WS, WC, D, ≺) be a behaviour instantiation. The translation of the behaviour instantiation is defined as

$$T_B'(WS, WC, D, ≺) = T_B(D, ≺) \cup T_B(WC) \cup WS \cup \{ok(d₁)\}$$

The answer set of the translated extended logic program T_B'(WS, WC, D, ≺) corresponds to the preferred answer set of the behaviour instantiation (WS, WC, D, ≺). The answer set can be used to identify the preferred actions or behaviours of an agent carrying out the behaviour (WC, D, ≺) in the state WS.

**Simple goal-directed navigation translation**

We demonstrate the behaviour translation framework for the simple goal-directed navigation. For ease of reference, a behaviour instantiation (WS, WC, D, ≺) of the simple goal-directed navigation is shown below, with identifiers assigned to each of the directives.

\[d₁ \quad forward \leftarrow gnorth, north\]
\[d₂ \quad forward \leftarrow geast, east\]
\[d₃ \quad forward \leftarrow gsouth, south\]
\[d₄ \quad forward \leftarrow gwest, west\]
\[d₅ \quad right \leftarrow gnorth, gwest, south\]
\[d₆ \quad right \leftarrow gnorth, west\]
\[d₇ \quad left \leftarrow gnorth\]
For each directive $d_n$, the name of the rule is $n$. Therefore, the name given to directive $d_1$ is ‘1’. In the translation of this behaviour we will omit brackets around the names of directives for clarity.

The translation $T_B(D, \prec)$ contains the following rules. For simplicity, we only show the translation for directives $d_1$, $d_5$, $d_6$, and $d_7$, as the translation of the other directives follow a similar pattern. The translation of directive $d_1$ becomes

$$\begin{align*}
\text{forward} & \leftarrow \text{binding}_1 \\
\text{binding}_1 & \leftarrow \text{trig}_1, \text{not \: defeat}_1 \\
\text{trig}_1 & \leftarrow \text{cgnorth}, \text{cnorth} \\
\text{cforward} & \leftarrow \text{trig}_1, \text{ok}_1, \text{not \: c-forward} \\
\text{defeat}_1 & \leftarrow \text{c-forward}, \text{ok}_1 \\
\text{ok}_2 & \leftarrow \text{cforward} \\
\text{ok}_2 & \leftarrow \text{defeat}_1 \\
\text{ok}_2 & \leftarrow \text{ok}_1, \text{not \: trig}_1
\end{align*}$$

Similarly the translation for directives $d_5$, $d_6$ and $d_7$ becomes

$$\begin{align*}
\text{forward} & \leftarrow \text{binding}_1 \\
\text{binding}_1 & \leftarrow \text{trig}_1, \text{not \: defeat}_1 \\
\text{trig}_1 & \leftarrow \text{cgnorth}, \text{cnorth} \\
\text{cforward} & \leftarrow \text{trig}_1, \text{ok}_1, \text{not \: c-forward} \\
\text{defeat}_1 & \leftarrow \text{c-forward}, \text{ok}_1 \\
\text{ok}_2 & \leftarrow \text{cforward} \\
\text{ok}_2 & \leftarrow \text{defeat}_1 \\
\text{ok}_2 & \leftarrow \text{ok}_1, \text{not \: trig}_1
\end{align*}$$

93
right ← binding5
binding5 ← trig5, not defeat5
trig5 ← cgnorth, cgwest, csouth
cright ← trig5, ok5, not c-right
defeat5 ← c-right, ok5
ok6 ← cright
ok6 ← defeat5
ok6 ← ok5, not trig5

right ← binding6
binding6 ← trig6, not defeat6
trig6 ← cgnorth, cgwest
cright ← trig6, ok6, not c-right
defeat6 ← c-right, ok6
ok7 ← cright
ok7 ← defeat6
ok7 ← ok6, not trig6

left ← binding7
binding7 ← trig7, not defeat7
trig7 ← cgnorth
cleft ← trig7, ok7, not c-left
defeat7 ← c-left, ok7
ok8 ← cleft
ok8 ← defeat7
ok8 ← ok7, not trig7

The remaining directives in $D$ are translated in a similar manner.
The translation $T_B(W_C)$ of the constraint formulae $W_C$ becomes

← forward, left
c-left ← cforward
c-forward ← cleft

← forward, right
c-right ← cforward
c-forward ← cright

← left, right
c-right ← cleft
c-left ← cright
The complete translation $T'_B(W_S, W_C, D, \prec)$ of the behaviour instantiation describing the simple goal-directed navigation applied in the agent state $W_S$ is as follows.

$$T'_B(W_S, W_C, D, \prec) = T_B(D, \prec) \cup T_B(W_C) \cup W_S \cup \{\text{ok}(d_1)\}$$

The answer set solver CLASP is used to find the answer set of $T'_B(W_S, W_C, D, \prec)$. The single resulting answer set corresponds to the preferred answer set of the behaviour instantiation $(W_S, W_C, D, \prec)$. For clarity, the literals that represent the state of the agent and the action for the agent to perform in this state are highlighted in bold in the answer set.

$$\text{ok}1 \ \text{west gnorth gwest trig7 trig6 ok2 trig4 trig16 ok16 c-left}$$
$$\text{ok15 defeat15 c-right ok7 defeat6 ok6 ok14 defeat14 ok13 defeat13}$$
$$\text{ok12 defeat12 ok11 defeat11 ok10 defeat10 ok9 defeat9 ok8 defeat8}$$
$$\text{defeat7 cforward ok5 defeat5 ok4 binding4 defeat16 ok17 forward}$$

The answer set of $T'_B(W_S, W_C, D, \prec)$ contains the action forward. Therefore, the simple goal-directed navigation specifies that, in the situation where the agent is facing west with its goal to the north-west, the agent should move forwards.

### 3.3.3 Using a behaviour translation

A behaviour definition $(W_C, D, \prec)$, as defined by the system designer, is a prioritised set of directives. In this form, it cannot easily be used by a Sinatra agent to determine its actions. However, by using the behaviour translation framework described above, the agent is able to determine the actions to perform in its current state by consulting the answer set of the behaviour instantiation for this state $(W_S, W_C, D, \prec)$. For this, the state facts $W_S$ must specify a complete interpretation of the set of facts that can describe the agent’s state for this behaviour.

Agents can be implemented to reason about their actions either at runtime or at compile time. Runtime decision making means that during each time step an agent will reason about its current state in order to determine how it should act. Compile time decision making means that an agent will have pre-determined how it should act in all possible world states.

Relatively simple agents may be limited to using compile time decision making to determine their actions. More intelligent agents, which have the ability to reason about their actions, have the potential to use runtime decision making. The behaviour translation framework can be used for agents that are capable of runtime decision making and for agents that are limited to compile time decision making.

A sufficiently intelligent agent can maintain an internal representation of the translation of a behaviour definition $(W_C, D, \prec)$. When the agent needs to determine its
actions using this behaviour, the agent can add the literals representing its current state $W_S$ to the behaviour translation and use CLASP to identify the resulting answer set. The actions or behaviours contained within this answer set indicate the actions for the agent to perform or the behaviours for the agent to follow during this time step. The agent will repeat this process for each time step, updating the behaviour translation with the appropriate current state facts.

An agent is able to use compile time decision making by consulting a pre-programmed state-action table to lookup the appropriate action to perform based on the agent’s current state. A separate state-action table is developed for each behaviour. The behaviour translation framework can be used to populate the entries in this state-action table. CLASP is used to find the answer set associated with the behaviour instantiation $(W_S, W_C, D, \prec)$ for each possible agent state $W_S$. The actions or behaviours in this answer set are recorded as the state-action table entry for this state.

When we began to develop Sinatra, the agents were implemented using TR programs. Agents that follow TR programs required compile time decision making methods to determine their actions. Now that the agents are implemented using behaviours, there is also the option of implementing runtime decision making. To maintain consistency with our original agent design, the agents in Sinatra are implemented to use compile time decision making by consulting a state-action table. We experiment with agents that use methods for runtime decision making for the hide-and-seek domain in Chapter 7.

Given a behaviour definition $(W_C, D, \prec)$ containing a set of directives of the form

$$\text{action/behaviour} \leftarrow \text{condition}$$

the aim is to generate a state action table with entries of the form

$$\text{state} \rightarrow \text{action(s) behaviour(s)}$$

The entries in the state-action table will identify the actions and behaviours prescribed by the behaviour definition for each possible current state of the agent.

The state of a state-action table entry is a complete interpretation of the set of literals that describe the possible states of an agent for this behaviour. Therefore, an entry in the state-action table that applies a specific directive will have a state such that

$$\text{condition} \subseteq \text{state}$$

A state-action table entry may have a state where multiple directives are applied. In this situation, the state-action table entry will direct the agent to carry out multiple actions and follow multiple behaviours. The behaviour translation framework uses the constraint formulae to ensure that, when multiple actions and behaviours
are found in the preferred answer set, there are no conflicts between these actions
and behaviours. Therefore, if a state-action table entry specifies multiple actions and
behaviours, then these actions and behaviours are to be performed concurrently.

The behaviour instantiation used to demonstrate the translation of the simple
goal-directed navigation resulted in a preferred answer set that contained the ac-
tion forward. By writing the state $W_S$ as a complete interpretation of the set of
facts describing the agent’s state, the state-action table for the simple goal-directed
navigation will contain the following entry.

$$\neg \text{north} \neg \text{east} \neg \text{south} \text{ west} \neg \text{gnorth} \neg \text{gsouth} \neg \text{geast} \text{ gwest} \rightarrow \text{forward}$$

Although the state is officially defined as a set of literals, we find it convenient to
identify different states in a state-action table using only the positive literals that
hold in the state. The absence of state atoms indicates that the negation of these
atoms hold in this state. Therefore, the above state-action table entry is equivalent
to the following entry

$$\text{west} \text{ gnorth} \text{ gwest} \rightarrow \text{forward}$$

3.3.4 Generating state-action tables

In order to populate the entries of the state-action table for a behaviour, the preferred
answer set of the behaviour definition must be found for every possible state of the
agent. Using the behaviour translation framework, together with the answer set
solver CLASP, we now outline the process of automating the generation of the state-
action table.

The aim is to be able to go from a behaviour definition $(W_C, D, \prec)$, to a complete
state-action table for this behaviour. The state-action table must contain an entry
for each set of state facts $W_S$. The process of translating a behaviour using the
behaviour translation framework and of generating the state-action table is auto-
mated. We will refer to this as the state-action table automatic generation process,
or STAG process.

A Perl implementation is used to carry out the STAG process. The input to the
process is a file containing the behaviour definition created by the system designer
$(W_C, D, \prec)$, together with additional information about the possible agent states.
The output of the process is the state-action table file containing an entry for every
agent state.

Figure 3.4 shows a flowchart of the STAG process, highlighting the three operations
that are carried out as part of the STAG process, as well as the input and output of
each of these steps. We now describe the details of the STAG process.
The input file to the Stag process is a behaviour definition \((W_C, D, \preceq)\). However, the behaviour translation framework requires an agent state \(W_S\) to be included in the behaviour that is translated. This agent state \(W_S\) must be a complete interpretation of the set of literals that can define the agent’s state for this behaviour. Not all combinations of state literals will constitute a valid agent state however. Therefore, the valid agent states must be specified in the input file in some manner. Using the
CLASP input language, it is possible to specify the agent states in a concise manner.

The CLASP input language includes the ability to define aggregate operations over literals. An aggregate provides a compact way for assigning truth values to a range of literals. Using aggregates, we are able to define the range of valid agent states where the behaviour should be applied in just a few lines of the CLASP input language. A single call to CLASP will then automatically generate the preferred answer set of the behaviour for each of these possible agent states.

The input file to the STAG process must also specify the set of atoms that will form the states and the set of atoms that will form the actions and behaviours of the state-action table entries. Using these sets, the STAG process is able to extract the appropriate atoms from each of the answer sets that are found by CLASP. These are used to write the state-action table entries to the output state-action table for the behaviour.

Therefore, the input file to the STAG process contains four elements.

- A behaviour definition \((W_C, D, \prec)\).
- A set of aggregates written using the CLASP input language \(W_{S'}\).
- A list of state atoms for the state-action table entries.
- A list of action and behaviour atoms for the state-action table entries.

The CLASP input language is used to define the behaviour definition \((W_C, D, \prec)\) in the input file. The logic program rules of the behaviour definition are specified using the symbol ‘:-’ to represent \(\leftarrow\). In addition, each logic program rule must be terminated with a ‘.’.

The aggregates \(W_{S'}\) define all the different agent states. These aggregates are written in the input file using the \#count aggregate syntax of CLASP [GKK+10]. Aggregates are operations on a multiset of literals. The \#count aggregate is used to specify a range of literals that can be true together and takes the form

\[
l \#\text{count} \{L_1, \ldots, L_n\} \ u
\]

where \(\{L_1, \ldots, L_n\}\) are a set of literals and \(l, u\) are integers specifying the lower bound and upper bound of the set respectively. The \#count aggregate finds all possible subsets of the literals \(\{L_1, \ldots, L_n\}\) of size \(\geq l\) and size \(\leq u\). In other words, every subset of distinct literals of size between \(l\) and \(u\) will be returned by the aggregate.

In the CLASP input language, the keyword \#count can be omitted from the aggregate. Therefore, the \#count aggregate can be equivalently written as

\[
l \{L_1, \ldots, L_n\} \ u
\]
As an example, in order to specify the range of literals that can be used by the **simple goal-directed navigation** to represent the different directions that the agent can be facing, the following aggregate is used.

\[
1 \{\text{north, east, south, west}\} 1
\]

This aggregate finds all subsets of size 1 of the set \{north, east, south, west\}. Therefore, the aggregate specifies that only one of \{north, east, south, west\} can be true at any one time. This corresponds to the agent facing in one of the four cardinal directions.

For specifying the direction of the agent’s goal from the agent’s current location, the following aggregate is used.

\[
0 \{\text{gnorth, gsouth}\} 1
\]

This aggregate specifies that either gnorth is true, or gsouth is true, or neither of these literals are true. Therefore, this aggregate corresponds to the fact that the agent’s goal location can be to the north of the agent or to the south of the agent, but cannot be both at the same time. In addition, the agent may be in the same row of the grid environment as its goal location, in which case neither gnorth nor gsouth will be true.

A similar aggregate can be used for geast and gwest. Multiple aggregates can be used together in order to define the different sets of facts that will form each of the agent’s states.

Aggregates allow the system designer to specify all valid agent states in a concise manner. However, the number of agent states, being also the number of answer sets found by CLASP and the number of entries in the generated state-action table, will grow exponentially with the introduction of each new set of state atoms and associated aggregate.

Within the input file, the prefixes ‘s:’; ‘in:’ and ‘out:’ are used to identify the aggregates, the list of state atoms and the list of action and behaviour atoms respectively. An example of an input file for the **simple goal-directed navigation** is shown in Listing 3.1.

**State-action table automatic generation process**

The STAG process is implemented in Perl because of Perl’s powerful regular expression language. Using the behaviour definition \((W_C, D, \prec)\) included in the input file, the STAG process creates an intermediate file containing the translation of the behaviour definition using the behaviour translation framework. However, the translation does not contain explicit state facts \(W_S\). Instead, the aggregates \(W_{S^r}\) included
forward :- gnorth, north.
forward :- geast, east.
forward :- gsouth, south.
forward :- gwest, west.

right :- gnorth, gwest, south.
right :- gnorth, west.
left :- gnorth.

right :- geast, gnorth, west.
right :- geast, north.
left :- geast.

right :- gsouth, geast, north.
right :- gsouth, east.
left :- gsouth.

right :- gwest, gsouth, east.
right :- gwest, south.
left :- gwest.

:- forward, left.
:- forward, right.
:- left, right.

s: 1 {north, east, south, west} 1.
s: 0 {gnorth, gsouth} 1.
s: 0 {geast, gwest} 1.

in: north east south west gnorth geast gsouth gwest.

out: left right forward.

Listing 3.1: The simple goal-directed navigation input file.

in the input file are identified by the STAG process and added directly to the inter-
mediate file containing the translated logic program.

The STAG process then calls CLASP, giving as input the intermediate translated
logic program file. CLASP outputs the answer sets, each corresponding to a different
set of state facts $W_S$ described by the aggregates $W_{S'}$.

The answer sets output by CLASP are captured by the STAG process in order to
identify the state-action table entries. Using the set of state atoms and the set of
action and behaviour atoms specified in the input file, the STAG process extracts the
appropriate atoms from the answer sets and writes these to the output state-action
table.
The output state-action table can then be used as a lookup table by the agents in order to determine their actions. Each agent determines its current state $W_S$ and identifies the appropriate entry in the lookup table. The action and behaviour tokens associated with this state allows the agent to determine which action method to execute and behaviour to follow. In this way the generated state-action table allows the agents to follow the directives of the behaviour.

**Output file**

Listing 3.2 shows a selection of the state-action table entries for the simple goal-directed navigation. The state-action table entries for when the agent is facing north and when the agent is facing east are shown, for all the different combinations of goal directions. Again, although the states are defined in terms of a set of literals, we choose to identify the different state-action table entries using only the positive literals that hold in that state.

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>north gnorth geast</td>
<td>forward</td>
</tr>
<tr>
<td>north gnorth gwest</td>
<td>forward</td>
</tr>
<tr>
<td>north gnorth</td>
<td>forward</td>
</tr>
<tr>
<td>north gsouth geast</td>
<td>right</td>
</tr>
<tr>
<td>north gsouth gwest</td>
<td>left</td>
</tr>
<tr>
<td>north gsouth</td>
<td>left</td>
</tr>
<tr>
<td>north geast</td>
<td>right</td>
</tr>
<tr>
<td>north gwest</td>
<td>left</td>
</tr>
<tr>
<td>north</td>
<td>→</td>
</tr>
<tr>
<td>east gnorth geast</td>
<td>forward</td>
</tr>
<tr>
<td>east gnorth gwest</td>
<td>left</td>
</tr>
<tr>
<td>east gnorth</td>
<td>left</td>
</tr>
<tr>
<td>east gsouth geast</td>
<td>forward</td>
</tr>
<tr>
<td>east gsouth gwest</td>
<td>right</td>
</tr>
<tr>
<td>east gsouth</td>
<td>right</td>
</tr>
<tr>
<td>east geast</td>
<td>forward</td>
</tr>
<tr>
<td>east gwest</td>
<td>left</td>
</tr>
<tr>
<td>east</td>
<td>→</td>
</tr>
</tbody>
</table>

Listing 3.2: Excerpt from the state-action table for the simple goal-directed navigation.

The state-action table entries that contain no action correspond to when the agent has reached its goal location. Normally, if an entry in a state-action table contains no action or behaviour token, then the agent will perform the default wait action. However, for the simple goal-directed navigation, when the agent has reached its goal location the behaviour has successfully achieved its objective of guiding the agent to its goal and so there is no need for any action to be associated with this state by
this behaviour.

As mentioned when we described the form of a directive, behaviours can be used in a nested manner to instruct an agent how to act. Therefore, in this situation, a higher level behaviour will specify the action for the agent to perform, or the behaviour that the agent should follow next. In the next two sections we describe how agents are able to use behaviours in this way.

3.4 Agents using behaviours

The behaviour method is designed to allow agents to take norms into account when determining their actions. Starting from a behaviour definition written by the system designer, we have described how a complete state-action table for this behaviour can be generated using the STAG process. We now describe the method by which Sinatra agents are able to use a state-action table to determine how they should act.

3.4.1 Understanding a state-action table entry

The entries in a state action table relate facts about the agent’s current state to actions for the agent to perform or behaviours for the agent to follow in this state.

\[
\text{state} \rightarrow \text{action(s) behaviour(s)}
\]

The elements of the state-action table entries are tokens that can be understood by the agent. Therefore, the agent must be able to relate the tokens contained in a state-action table entry to the perception methods, action methods and behaviours that the agent can execute. This is achieved by storing a dictionary of tokens in the agent’s internal memory.

A direct 1:1 relationship exists between the tokens used to represent the actions and behaviours in a state-action table and the action methods and behaviours available to the agent. However, the relationship between the tokens used to represent the states in a state-action table and the perception methods available to the agent is slightly less straightforward.

The tokens used by a state-action table to describe the state are general facts about the agent’s state, not necessarily the complete information available to the agent itself. For example, the token gnorth is used to specify that the agent’s goal is to the north of the agent, rather than using the x, y position of the agent’s goal location. In addition, the tokens describing a state contain information about the results of the agent’s perception methods, rather than referring to the perception methods themselves.

The current state of the agent can contain information based on the results of all the agent’s perception methods, combined with information stored in the agent’s
internal memory. However, the current state for a behaviour only requires sufficient information for the agent to distinguish when different directives should be followed. Therefore, the states of a state-action table generated from a behaviour definition refer only to these required elements.

Table 3.5 describes the dictionary of tokens used by the Sinatra agents in order to understand the state-action table for the simple goal-directed navigation. The dictionary relates the tokens of the state-action table entries to the results of the agent’s perception methods and the action methods available to the agent, as appropriate. In this instance, there is a 1:1 relationship between the tokens used by the behaviour and the output of the perception methods of the agent, but this is not always the case.

<table>
<thead>
<tr>
<th>Token</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>north, east, south and west</td>
<td>The result of the direction perception method, specifying the direction the agent facing.</td>
</tr>
<tr>
<td>gnorth, geast, gsouth and gwest</td>
<td>The result of the gnorth, geast, gsouth and gwest perception methods, specifying the direction of the agent’s goal location from its current location.</td>
</tr>
<tr>
<td>forward</td>
<td>The moveFowards action.</td>
</tr>
<tr>
<td>left and right</td>
<td>The turnLeft and turnRight actions respectively.</td>
</tr>
</tbody>
</table>

Table 3.5: The dictionary of tokens for the simple goal-directed navigation.

3.4.2 Using a state-action table

An agent will usually have multiple behaviours that it is implemented to follow. Therefore, an agent will follow the directives of its current active behaviour or behaviours. In order to use the state-action table for its active behaviour, an agent must determine its current state using its perception methods, look up the appropriate state-action table entry for this state and then execute the specified action or actions. If the state-action table entry specifies a behaviour for the agent to follow, this behaviour also becomes the agent’s active behaviour and the agent must repeat this process for the new behaviour.

For clarity, however, we first describe how an agent uses a single state-action table. We describe the general methods used by the agent and how these methods are implemented for the simple goal-directed navigation. As will be clear from the development of the simple goal-directed navigation, the entries in the state-action table for this behaviour do not refer to any behaviours.

The currentState method is used by the Sinatra agents to allow them to determine their current state facts for the behaviour. As each behaviour has a separate state-action table, the currentState method must identify the appropriate set of
state facts correctly for the agent’s current active behaviour.

Having identified the set of tokens representing the agent’s current state facts, the `lookUpAction` method is used to identify the corresponding actions for the agent to perform. The `lookUpAction` method consults the appropriate state-action table for the agent’s behaviour, identifies the state-action table entry that corresponds to the state facts found by the `currentState` method and returns the token or tokens specified by this state-action table entry.

In the current implementation of Sinatra, the state-action tables for behaviours are stored as text files. For example, the state-action table for the `simple goal-directed navigation` was shown in Listing 3.2. The `lookUpAction` method uses a Perl regular expression to lookup the agent’s current state in this file and to identify the associated actions or behaviours. Perl has been optimised for fast text processing, meaning that this operation can be carried out efficiently, even when the search space is very large due to an exponential growth in the state space when new variables are introduced.

If, or when, performance during this lookup step becomes an issue, the state-action table can be converted to a Java HashMap object. The state of the state-action table entry will be used as the key to index the hash map and the associated value will return the actions and behaviours for this state. A Java HashMap can be used as the Sinatra agents will only be reading from the state-action table, they will not be modifying the individual entries. Therefore, a ‘synchronized’ hash map implementation is not required. The expected lookup time for a hash map is $O(1)$ and the space requirement is $O(n)$.

The token or tokens returned by the `lookUpAction` method represent the actions and behaviours specified by the behaviour for the agent’s current state. The `actionSwitch` method uses the agent’s dictionary of tokens to understand these tokens. In the case of actions, the `actionSwitch` method identifies the action methods for the agent to execute and calls these methods. We will describe the case for behaviours in due course.

For the `simple goal-directed navigation`, the `currentState` method is implemented to call the `direction`, `gnorth`, `geast`, `gsouth` and `gwest` perception methods. The results of these perceptions are used to construct the set of tokens that represent the state facts $W_S$, based on the dictionary of tokens shown in Table 3.5. The state facts $W_S$ are used by the `lookUpAction` method to identify the correct entry in the state-action table for the `simple goal-directed navigation`. The action token associated with these state facts is then used by the `actionSwitch` method to call either the `moveForwards`, `turnLeft` or `turnRight` actions, as appropriate.

Therefore, we are now able to demonstrate the `simple goal-directed navigation` being used by agents in Sinatra. Figure 3.5 shows a single agent in Sinatra using the `simple goal-directed navigation`. The path that will be followed by the Sinatra
agent is highlighted on the grid, leading the agent to its goal location in the first column of the grid environment. By observing the simulation of the agent using this behaviour in the Sinatra GUI, we are able to see the agent follow this highlighted path and reach its goal location.

![Figure 3.5: A single agent in Sinatra using the simple goal-directed navigation.](image)

3.5 Agents using multiple behaviours

We now describe how Sinatra agents can be implemented to be able to follow multiple behaviours. Directives that instruct an agent to follow a behaviour can be used to implement a nested set of behaviours. We refer to a behaviour that contains these type of directives as a behaviour-switch. Behaviour-switches are used to control the general operations of an agent and to implement more dynamic behaviours.

We have shown how an agent following the simple goal-directed navigation is able to use this behaviour to reach its goal location. However, once the agent has reached this location, the agent must request to be assigned a new goal location by the simulation. A behaviour-switch is used to instruct the agent what to do when it has reached its goal.

In addition, rather than being implemented to follow a single behaviour, agents will usually have multiple behaviours that they are implemented to follow under different circumstances. For example, an agent may follow the simple goal-directed navigation in order to reach a goal location within the grid environment, but switch to following a behaviour that allows the agent to move around an obstacle whenever the agent perceives that its path is obstructed. A behaviour-switch is used to handle this changing of active behaviours.

Alternatively, if the agent is capable of performing multiple actions concurrently, an agent may have multiple behaviours that it is implemented to follow simultaneously. For example, an agent may follow the simple goal-directed navigation, whilst also following a behaviour that directs the agent to send progress messages to another agent in the multi-agent system. A behaviour-switch is used to instruct the
agent which behaviours to follow.

3.5.1 Implementing a behaviour-switch

A behaviour-switch is a high-level form of behaviour. Instead of directing an agent to perform particular actions depending on conditions that hold in its current state, the behaviour-switch also contains directives that direct an agent to follow a different behaviour.

Beyond the nature of the directives in the behaviour, there is no difference between a behaviour that only directs the agent to perform actions and a behaviour-switch, both in terms of how they are implemented and the application of the behaviour translation method and STAG process to these behaviours. A directive always functions the same, whether it refers to an action or a behaviour.

If an agent is implemented to perform one very basic behaviour that can be carried out continuously, such as continually traversing the environment in a set pattern and cleaning any areas of dirt that it encounters, the agent will not require a behaviour-switch. Otherwise, when the agent is able to perform multiple behaviours or is not able to perform one behaviour continuously without receiving further input, then a behaviour-switch must be used to manage the overall behaviour of the agent.

Any behaviour that contains directives that refer to a behaviour is also a behaviour-switch. However, an agent must have one behaviour-switch that operates as the master behaviour-switch. This behaviour-switch is the first behaviour that is consulted by the agent during every time step and so the master behaviour-switch is always one of the agent’s active behaviours.

A behaviour-switch for an agent using the simple goal-directed navigation is used to handle the case where the agent has reached its goal location and needs to select a new goal. An example of the behaviour definition for such a behaviour-switch is shown below.

\[
\begin{align*}
\text{newgoal} & \leftarrow \text{atgoal} \\
\text{goaldirected} & \leftarrow \\
& \leftarrow \text{newgoal, goaldirected}
\end{align*}
\]

By default, the agent will follow the simple goal-directed navigation, indicated by the goaldirected token. However, when the agent has reached its goal location, the agent should perform the newgoal action instead. The details of the action method indicated by the newgoal token are not defined, but the agent uses this method in order to select or to be assigned a new goal location to reach. The \(x, y\) coordinates of the agent’s new goal are stored in the agent’s internal memory.

We have describe how an agent may follow the simple goal-directed navigation in order to reach a goal location within the grid environment, but switch to following a
behaviour that allows the agent to move around an obstacle whenever the agent perceives that its path is obstructed. We call this new behaviour the **obstacle avoidance** behaviour, which will be defined in Section 4.3.1. The **obstacle avoidance** behaviour directs an agent to move around an obstacle in its path.

The behaviour definition of the behaviour-switch for an agent that is able to follow the **simple goal-directed navigation** and the **obstacle avoidance** behaviour is shown below. We call this behaviour-switch the **example master behaviour-switch**.

\[
\begin{align*}
\text{newgoal} & \leftarrow \text{atgoal} \\
\text{obstacle} & \leftarrow \text{obstructed} \\
\text{goaldirected} & \leftarrow \\
& \leftarrow \text{newgoal, obstacle} \\
& \leftarrow \text{goaldirected, obstacle} \\
& \leftarrow \text{newgoal, goaldirected}
\end{align*}
\]

Again, by default, the agent will follow the **simple goal-directed navigation**. When the agent observes that it is **obstructed** the agent will switch to following the **obstacle avoidance** behaviour, indicated by the **obstacle** token.

Based on this behaviour definition, the STAG process can be used to convert this behaviour-switch into a state-action table that can be used by the agents. The state-action table generated by the STAG process for the **example master behaviour-switch** is shown in Listing 3.3.

Listing 3.3: The state-action table for the **example master behaviour-switch**.

\[
\begin{align*}
\text{obstructed atgoal} & \rightarrow \text{newgoal} \\
\text{obstructed} & \rightarrow \text{obstacle} \\
\text{atgoal} & \rightarrow \text{newgoal} \\
& \rightarrow \text{goaldirected}
\end{align*}
\]

### 3.5.2 Using a behaviour-switch

In order to determine the actions to perform during each time step, the agent calls the **act** method. The **act** method is a high-level method that takes as input a token representing an active behaviour of the agent. Therefore, the first time **act** is called during a time step, it will be passed the token for the agent’s master behaviour-switch.

The **act** method then governs the process of determining the agent’s current state for this behaviour, identifying the tokens associated with this state and executing the actions or following the behaviours described by these tokens. To do this, the **act** method uses the same general methods described previously for using a single
state-action table. However, we now describe how these methods are applied to any state-action table, including those that represent a behaviour-switch. As a behaviour-switch is a special kind of behaviour, the methods presented below can be used by an agent to follow the directives of any behaviour.

We describe the process by which the act method is used for a behaviour-switch in terms of the example master behaviour-switch. Therefore, we imagine that the act method has been called with the token master, identifying the example master behaviour-switch. The dictionary of tokens for the state-action table for the example master behaviour-switch is shown in Table 3.6.

<table>
<thead>
<tr>
<th>Token</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>obstructed</td>
<td>The result of the north, east, south or west perception methods, depending on the direction of the agent’s goal. Specifies whether a direction that would lead to the agent’s goal is obstructed.</td>
</tr>
<tr>
<td>atgoal</td>
<td>The result of the atGoal perception method.</td>
</tr>
<tr>
<td>obstacle</td>
<td>The obstacle avoidance behaviour.</td>
</tr>
<tr>
<td>newgoal</td>
<td>The newGoal action method, which selects or assigns a new goal location to the agent. This goal location is recorded in the agent’s internal memory.</td>
</tr>
<tr>
<td>goaldirected</td>
<td>The simple goal-directed navigation behaviour.</td>
</tr>
</tbody>
</table>

Table 3.6: The dictionary of tokens for the example master behaviour-switch.

To use a behaviour-switch, the agent must be able to identify its current state using its perception methods. This step is handled by the currentState method, which is the first method called by the act method. The state facts returned by the currentState method depend on the agent’s current active behaviour.

In the case of the example master behaviour-switch, the agent must use the atgoal perception method, as well as determine whether it is obstructed. To do this, the agent must be implemented with a new method that tests the appropriate perception methods and combines the output of these methods to conclude whether the required property – that the agent is prevented from moving towards its goal – holds in the agent’s current state. If this method returns true, the currentState method adds the token obstructed to the state facts of the agent.

Using the state facts returned by the currentState method, the lookUpAction method consults the appropriate state-action table for the agent’s active behaviour and returns the token or tokens associated with this state. The lookUpAction method is the second method called by the act method.

The act method then calls the actionSwitch method for each token returned by the lookUpAction method. The actionSwitch method uses the agent’s dictionary of tokens to decode each token. In the special case where the lookUpAction method
returns no tokens for a particular agent state, the actionSwitch method calls the default wait action method and the agent performs this action for the time step. Otherwise, if the token refers to an action method of the agent, the actionSwitch method calls this method and the agent executes the appropriate action. In the example master behaviour-switch, this will occur when the newgoal token is returned, allowing a new goal to be assigned to the agent.

If the token refers to a behaviour, however, this behaviour must become an active behaviour of the agent for this time step. Therefore, the actionSwitch method calls the act method again, this time passing in the token for the new active behaviour. For the example master behaviour-switch, this will occur when the obstacle or goaldirected tokens are returned.

Therefore, the act method can be called multiple times during a single time step, each time handling the application of a different active behaviour. Whenever the current state of the agent for this behaviour indicates that a new behaviour should be followed, the act method will allow the agent to follow the directives of this behaviour. When every active behaviour has returned only action tokens, the calls to the act method will terminate and the agent will have executed all of the prescribed actions for this time step.

Table 3.7 describes the methods used by the agent to consult a behaviour. These methods allow the agent to follow the directives of any behaviour, whether it is a behaviour-switch or a behaviour that contains only actions in its directives.

### 3.5.3 Incorporating a new behaviour

For each new behaviour, the following modifications are required to the Sinatra agent implementation.

- The currentState method must be able to construct the set of tokens $W_S$ for the agent to be able to consult the state-action table for the new behaviour.

- The lookUpAction method must be able to identify the correct state-action table to consult for the new behaviour.

- The dictionary of tokens must be updated to include a token for the new behaviour and any state, action or other behaviour tokens that are referred to in the directives of this behaviour.

- New perceptions or actions methods required by the new behaviour will need to be implemented. In some cases, this may involve enabling the agent to combine existing perception methods or action methods into a new high-level perception or action respectively. In other cases, this may involve significantly extending the basic sensory and action capabilities of the Sinatra agents.
Method | Description
--- | ---
act | Instructs an agent how to act based on the active behaviour given as input to this method. The agent determines its current state for this behaviour (currentState) and then consults the appropriate state-action table (lookUpAction). Having identified the token or tokens associated with this state, the agent executes these actions and follows these behaviours (actionSwitch).
currentState | Uses the perception methods of the agent to specify the state facts $W_S$ for the agent’s current state in the manner required by the state-action table for the active behaviour. Returns a set of tokens representing the agent’s state.
lookUpAction | Consults the state-action table for the active behaviour and identifies the appropriate entry based on the state facts returned by currentState. Returns the set of tokens associated with this state by the state-action table, representing the actions and behaviours for the agent to perform and follow during this time step.
actionSwitch | Identifies the action methods and behaviours referred to by the tokens returned by lookUpAction. Executes these action methods and calls act in order to direct the agent to follow the new active behaviours.

Table 3.7: The methods used by Sinatra agents to consult a state-action table.

While the modifications required to the Sinatra agent implementation to introduce new perception or action methods will depend on the complexity of the new capabilities that are being introduced, the modifications required to allow a Sinatra agent to follow a new behaviour are relatively minor. These modifications involve only small changes to the methods defined in Table 3.7.

### 3.5.4 Behaviours as actions

Action methods and behaviours are indistinguishable to the behaviour method when specified in the right hand side of directives. An action method is the implementation of an action available to the agent, where the implementation is written in the agent’s internal code. The actions moveForwards and turnLeft are examples of actions methods available to the Sinatra agents. A behaviour allows the agent to act in a certain manner by directing the agent to execute specific action methods when different conditions hold. Therefore, a behaviour itself can be considered a form of action implementation.

This means that we are able to treat a behaviour that directs an agent to act in a certain way in the same manner as an action method that implements this same physical behaviour of the agent. For example, an agent may be implemented with
an action method `moveToNW`. This action method can be used by the agent over a number of time steps to move to the north-west corner of the grid environment. Each time the `moveToNW` action method is executed, the agent moves forwards or turns in order to be closer to the north-west corner.

The `moveToNW` action method can direct the agent to act in the same way as a *move to north-west* behaviour. The *move to north-west* behaviour will contain directives that instruct the agent when to move forwards and when to turn. The implementation of the `moveToNW` action method and the *move to north-west* behaviour can be such that the actions performed by the agent during each time step would be the same. Therefore, for the behaviour method, the follow directives can have the same meaning.

\[
\text{behaviour} \leftarrow \text{condition} \\
\text{action} \leftarrow \text{condition}
\]

This property allows us to define a behaviour-switch as a prioritised extended logic program and to apply the STAG process to generate the associated state-action table. The difference between an *action* and a *behaviour* in a directive is distinguished by the agent when consulting its dictionary of tokens as part of the `lookUpAction` method.

In this way, a system of nested behaviours can be used to implement an agent, where the actions specified by the directives of an active behaviour may themselves be implemented as another behaviour. This approach means that the individual behaviours can be implemented in a modular manner. Each behaviour definition is written in terms of its own directives, which reduces the size of the state-action table for each behaviour.

Whether an action is implemented as a behaviour or as an action method depends on the choices of the system designer and the capabilities of the agent. As described for the example `moveToNW` action method and *move to north-west* behaviour, both can be used to achieve the same functionality of the agent.

Therefore, a directive in a behaviour to instruct the agent to move to the north-west corner of the grid environment can be written as follows, regardless of whether the token `movetonw` refers to an action method or a behaviour.

\[
movetonw \leftarrow \text{condition}
\]

The token `movetonw` is only specified as an action method or a behaviour when the agent is implemented, via the agent's dictionary of tokens, to understand the state-action table of a behaviour.

In fact, if there is a `moveToNW` action method implemented in the agent's internal code, then a *move to north-west* behaviour can be implemented trivially using this
action method. The behaviour definition will require only a single directive and no constraint formulae.

\[ \text{moveToNW} \leftarrow \]

There is no condition required in this directive as the \text{moveToNW} action method is implemented to execute the correct action for the agent’s current state.

### 3.6 Summary

In this chapter we have described the implementation of the Sinatra test bed. Sinatra is able to simulate the concurrent execution of multiple independently acting agents. These agents are implemented using a general agent model, which can be extended with additional capabilities by the system designer as required.

The Sinatra agents determine their actions based on behaviours that are implemented by the system designer. We have described how behaviours can be unambiguously defined by the system designer in the form of a prioritised extended logic program, containing a set of directives and constraint formulae.

A behaviour definition can be used to generate a state-action table using the \text{STAG} process. The \text{STAG} process makes use of the behaviour translation framework in order to construct the state-action table entries, using \text{CLASP} to generate the preferred result for each agent state.

Sinatra agents are able to understand state-action tables generated from behaviour definitions via a dictionary of tokens stored by the agent. We have described how a Sinatra agent is able to use a behaviour in the form of a state-action table to determine its actions during each time step of the simulation. We have also described how Sinatra agents are able to utilise multiple behaviours by the implementation of behaviour-switches and how behaviours can be nested as a method of implementing different action capabilities.

In the remainder of this thesis we investigate the implementation of norms using Sinatra and the behaviour method. Sinatra allows new behaviours to be implemented efficiently and for the interactions of agents following the directives of these behaviours to be observed and replayed. We are also able to extend the Sinatra agent implementation when desired in order to investigate the use of behaviours in different domains and with agents that possess different physical and reasoning capabilities.

We begin by using Sinatra to investigate norms, implemented using directives, to manage agent interactions.
4 Behaviours to manage agent interactions

In Chapter 3 we showed the development of the simple goal-directed navigation. For a single agent domain, an agent using the simple goal-directed navigation is always able to move around the grid environment and to reach its goal location successfully. However, this is not always the case when there are multiple agents in the grid environment. When agents meet, it is possible for the agents to interact in such a way as to prevent each other from being able to reach their goals.

Figure 4.1 shows an example of such an agent interaction occurring. The two agents are both using the simple goal-directed navigation to determine their actions. However, the agents are blocking each other’s path and so the agents remain stationary.

![Figure 4.1: An undesirable agent interaction, where neither agent can progress towards its goal.](image)

The agent interaction in Figure 4.1 is an example of an undesirable agent interaction, being an interaction that is deemed by the system designer to be undesirable. In particular, an agent interaction that prevents the agents from being able to achieve their goals is (often) an undesirable agent interaction.

A method for managing the interactions of agents is necessary for the agents in the multi-agent system to be able to coexist and operate effectively. We examine the use of norms, implemented using directives, to manage agent interactions.
4.1 Managing undesirable agent interactions

We have identified two possible ways that behaviours can be used to manage agent interactions. Behaviours can attempt to prevent undesirable agent interactions from occurring, or behaviours can attempt to resolve undesirable agent interactions when they do occur.

In order to prevent undesirable agent interactions, the properties of the agents’ actions that lead to a particular interaction occurring must be identified. For example, an undesirable agent interaction may occur more frequently in a certain area of the environment, or when a particular combination of behaviours are used together. The system designer must then design a set of directives that limits the ability of the agents to act in this way. By using the new directives in place of the old behaviour or behaviours, the agents should act in a manner that leads to fewer occurrences of the undesirable agent interaction.

In order to resolve undesirable agent interactions, the system designer must identify general properties that hold whenever a particular interaction is occurring. Using these properties, the system designer can design a set of directives to be used when the agent is participating in this interaction. The new directives should allow the agent to execute a general strategy that will cause the agent to no longer be participating in the undesirable interaction. For example, instructing the agent to move in a certain manner so that the interacting agents will no longer be obstructing each other.

The properties identified by the system designer as holding whenever a particular undesirable agent interaction is occurring can also be used to allow the agents to identify when they are participating in this undesirable interaction. This will become the trigger condition for the agent to know to switch to using the new behaviour in order to resolve the undesirable interaction.

We demonstrate these two strategies for managing agent interactions by developing behaviours that are designed to prevent or to resolve undesirable agent interactions of the type shown in Figure 4.1. We will refer to this type of undesirable interaction as a stationary interaction. A stationary interaction occurs when all the participating agents mutually obstructing each other’s progress, leading to the agents remaining stationary.

When managing agent interactions, however, the system designer must be careful not to constrain the actions of the agents unnecessarily. This issue is highlighted by Shoham and Tennenholtz when describing the development of social laws to prevent collisions between mobile robots.

“[I]n our zeal to prevent collision we must be careful not to preclude the possibility of robots reaching their destinations (so, for example, forbidding all motion would be inappropriate). In fact, we will be interested in
traffic laws that not only allow robots to reach their destinations without collision, but allow them to do so reasonable efficiently” [ST95] p.237.

While this is an important consideration when implementing behaviours, it is not our intention in this chapter to develop behaviours with any specific (nice) properties. Instead, we aim to demonstrate how Sinatra can be used to test and also to visualise what happens when different sets of directives are adopted by the agents in a multi-agent system.

Therefore, in this chapter, we demonstrate the use of Sinatra to implement (some form of) norm-governed multi-agent system, based on two suggested strategies for using behaviours to manage agent interactions. We describe the implementation of four behaviours for use by Sinatra agents. To prevent undesirable agent interactions we develop the highway behaviour and the strict traffic lanes behaviour. These behaviours extend or replace the simple goal-directed navigation. To resolve undesirable agent interactions we develop the obstacle avoidance behaviour and the traffic law behaviour. These behaviours are separate behaviours that are used by the agents in conjunction with the simple goal-directed navigation by making use of a behaviour-switch.

4.2 Example: Using behaviours to prevent stationary interactions

In order to prevent undesirable agent interactions, behaviours must direct the agents to act in a manner that avoids situations where the undesirable interaction can occur. To demonstrate this strategy, we use the concrete scenario of behaviours that are designed to prevent stationary interactions. For this scenario, we consider two example behaviours.

One obvious method is to constrain agent movement to traffic lanes, or ‘highways’ for short. Therefore, we develop the highway behaviour. This behaviour specifies a pair of rows and a pair of columns of the environment that must be used by the Sinatra agents to travel east and west or north and south across the grid respectively.

In the second example, we extend the highway behaviour to develop the strict traffic lanes behaviour. In this behaviour, every row and column of the grid environment has permitted directions of travel that must be obeyed by the agents.

Both the highway behaviour and the strict traffic lanes behaviour are designed to direct an agent to reach its goal using a particular strategy. Therefore, these behaviours are used by Sinatra agents in place of the simple goal-directed navigation.

In these experiments, it is not our intention to develop an optimal set of ‘traffic rules’ for agents in a grid environment. Instead, we demonstrate the implementation of new behaviours using Sinatra, where the behaviours allow the agents to comply
with a norm or set of norms. We also investigate the efficacy of our suggested strategy for using behaviours to manage agent interactions, being to attempt to prevent stationary interactions from occurring.

4.2.1 Implementing the highway behaviour

Stationary interactions between agents using the simple goal-directed navigation occur when agents attempt to move forwards but find themselves in a situation where they mutually obstruct each other. In order to prevent these situations from occurring, the highway behaviour attempts to manage the movement of agents as they try to reach their goals.

We have specified that the goal locations in Sinatra are found around the perimeter of the grid environment, which represent pick-up and drop-off locations for the agents to reach. Therefore, stationary interactions will also often occur around the perimeter of the environment, as there is a higher probability that agents will meet in this area. The highway behaviour attempts to prevent agents from interacting around the perimeter by instructing agents to travel through the middle of the environment as they move towards their goals.

Two rows in the middle of the grid are designated for agents travelling east and west. Similarly, two columns are designated for agents travelling north and south. These pairs of rows and columns act as highways that the agents must use when travelling to their goals.

We now describe the implementation of the highway behaviour. We begin by focussing on the modifications required for a Sinatra agent to be able to use the highway behaviour, before implementing the highway behaviour itself.

Highway behaviour agent perceptions

In order to use the highway behaviour, the Sinatra agents must be aware of the size of the grid environment and the rows and columns that are specified for each direction of travel. This information is stored by the Sinatra agents in their internal memory.

The Sinatra agents must also be extended with additional perception methods. These new perception methods are high-level methods that combine perception methods already available to the agent and information stored in the agent’s internal memory.

To use the highway behaviour, an agent needs to use the correct highway in order to move across the grid environment. In general, an agent using the highway behaviour will travel from the periphery (where it has just reached its previous goal location) towards the centre of the grid. Here, the agent will travel in the appropriate direction along the correct highway until it is in line with its goal location. From this position,
the agent will leave the highway and travel directly towards its goal on the periphery of the grid.

In order to determine the correct highway to travel along, the agent must be able to determine on which edge of the grid environment (north, east, south or west) its goal is located. If the agent’s goal is in a corner of the grid environment, then the agent will observe that the goal is on the north or south edge of the grid environment only. This is to avoid confusion over which highway the agent should use to reach its goal.

For a goal on the north or south edge of the grid environment, the agent must travel along the east or the west highway until it is in line with its goal location. Similarly, for a goal on the east or west edge of the grid environment, the agent must travel along the north or the south highway.

Having identified the correct highway for the agent to travel along, the agent must move to reach this highway. The agent is aware of the positions of the different highways as this information is stored in its internal memory. Therefore, the agent is able to perceive, by comparing its current location to the known position of the highway, whether the agent is on the correct highway, or to the north, east, south or west of this highway.

Once an agent has reached the correct highway, the agent must turn to face the required direction of travel and then move along the highway until it is in line with its goal location. Therefore, the agent must also be able to perceive when it is in line with its goal. Having reached the point where the agent is in line with its goal location, the agent leaves the highway in order to reach its goal.

Table 4.1 shows the additional perception methods that are required by the Sinatra agents in order to use the highway behaviour. These new perception methods combine methods already available to the agent and information stored by the agent. The sensory capabilities of the Sinatra agents do not have to be extended to allow an agent to execute these additional perception methods. Therefore, the new perception methods appear to be appropriate, in that they would be reasonably easy to implement in even a simple robotic device.

New tokens must be added to the agent’s dictionary of tokens in order for the agent to be able to describe the state facts required by the highway behaviour. The state facts of the agent will be based on the results of the additional perception methods that have been introduced. Table 4.2 defines these additional tokens.
<table>
<thead>
<tr>
<th>Perception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rowGoal and columnGoal</td>
<td>True if the agent is in the same row or column as its goal location respectively. Uses the agent’s location and the agent’s goalLocation.</td>
</tr>
<tr>
<td>goalEdge</td>
<td>Returns the edge of the grid environment (north, east, south, west) of the agent’s goal location. In the case of a goal location in a corner of the grid environment, only north or south will be returned. Uses the agent’s goalLocation and the size of the grid environment.</td>
</tr>
<tr>
<td>highway</td>
<td>Returns the highway that an agent needs to use in order to reach its goal location. Uses the relative position of the agent’s location and its goalLocation, as well as the goalEdge of the agent’s goal.</td>
</tr>
<tr>
<td>findHighway</td>
<td>Returns an enumerated type specifying whether the agent is on the highway that it needs to travel along. If the agent is not on the correct highway, then the method returns whether the agent is to the north, east, south or west of this highway. Uses the agent’s location and the positions of the different highways.</td>
</tr>
<tr>
<td>onHighway</td>
<td>Returns true if the agent is on the correct highway and is facing in the correct direction of travel. Uses findHighway, direction and the positions of the different highways.</td>
</tr>
</tbody>
</table>

Table 4.1: The perception methods for the highway behaviour.

<table>
<thead>
<tr>
<th>Token</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>rowgoal and col-goal</td>
<td>The results of the rowGoal and columnGoal perception methods respectively, specifying when the agent is in line with its goal location.</td>
</tr>
<tr>
<td>hdirection</td>
<td>The result of the onHighway perception method, specifying that the agent is on the correct highway and is facing in the correct direction.</td>
</tr>
<tr>
<td>highway</td>
<td>The result of the findHighway perception method when the agent is on the correct highway.</td>
</tr>
<tr>
<td>anorth, aeast, asouth and awest</td>
<td>The results of the findHighway perception method when the agent is not on the correct highway, specifying the relative location of the agent from the correct highway.</td>
</tr>
</tbody>
</table>

Table 4.2: The dictionary of tokens for the highway behaviour.
**Highway behaviour**

Listing 4.1 shows the highway behaviour input file. The highway behaviour contains a set of directives for allowing the agent to reach the correct highway, a set of directives for travelling along the highway and a set of directives for travelling from the highway to the agent’s goal.

The first 12 directives in Listing 4.1 handle the case when the agent has travelled along the appropriate highway and is now in line with its goal, shown by rowgoal and colgoal. These directives allow the agent to leave the highway and to reach its goal location. The next two directives handle the case when the agent is on the correct highway. The agent must turn until it is facing the correct direction of travel and then move along the highway until it is in line with its goal. The final set of directives handle the case when the agent is to the north, east, south or west of the highway that it needs to use. These final directives allow the agent to reach the correct highway from its starting location.

The highway behaviour can be used by the agents as a way of reaching their goal locations. This means that when an agent is implemented to follow the highway behaviour, it will use the highway behaviour instead of the simple goal-directed navigation. Apart from specifying the use of the highway behaviour in place of the simple goal-directed navigation in the agent’s behaviour-switch, the Sinatra agents require no further modifications in order to follow the highway behaviour.

### 4.2.2 Demonstration

Figure 4.2 shows two agents in Sinatra using the highway behaviour. The paths that the agents will follow are highlighted in the appropriate colour. The highlighted paths also demonstrate the locations of the east and the west highways.

![Figure 4.2: Two agents in Sinatra using the highway behaviour.](image)

The red agent, with a goal on the north edge of the grid, will travel from its current position to the west highway. The red agent will then travel west along this highway until it is in line with its goal location. At this point, the red agent will
forward :- gnorth, north, colgoal.
right :- gnorth, west, colgoal.
left :- gnorth, colgoal.

forward :- geast, east, rowgoal.
right :- geast, north, rowgoal.
left :- geast, rowgoal.

forward :- gsouth, south, colgoal.
right :- gsouth, east, colgoal.
left :- gsouth, colgoal.

forward :- gwest, west, rowgoal.
right :- gwest, south, rowgoal.
left :- gwest, rowgoal.

forward :- hdirection.
left :- highway.

forward :- anorth, south.
right :- anorth, east.
left :- anorth.
forward :- asouth, north.
right :- asouth, west.
left :- asouth.

forward :- aeast, west.
right :- aeast, south.
left :- aeast.
forward :- awest, east.
right :- awest, north.
left :- awest.

:- forward, left.
:- forward, right.
:- left, right.

s: 1 {north, east, south, west} 1.
s: 1 {gnorth, gsouth, rowgoal} 1.
s: 1 {geast, gwest, colgoal} 1.
s: 1 {hdirection, highway, anorth, aeast, asouth, awest} 1.

in: north east south west gnorth geast gsouth gwest rowgoal colgoal hdirection highway anorth aeast asouth awest.

out: forward left right.

Listing 4.1: The highway behaviour input file.
leave the west highway in order to reach its goal. Similarly, the blue agent will travel along the east highway to reach its goal location.

Sinatra was used to simulate different sizes of agent population using the highway behaviour. On its own, a single agent using the highway behaviour was always able to reach its goal location. For agent populations of two or more, however, the agents were not always able to reach their goals successfully.

Based purely on observations of the agents in Sinatra, the occurrence of stationary interactions appeared to be less frequent between agents using the highway behaviour than between agents using the simple goal-directed navigation. In particular, there appeared to be fewer stationary interactions around the periphery of the grid environment. However, the occurrence of stationary interactions was still relatively common, especially when agents travel to and from the different highways.

Figure 4.3 shows an example of a stationary interaction caused by agents using the highway behaviour. The red agent is attempting to travel to the north highway in order to reach its goal location on the west edge of the grid. The blue agent is moving along the west highway, where it has met the red agent. From this position, the two agents are mutually obstructing each other and neither agent will be able to reach its goal.

\[\text{Figure 4.3: A stationary interaction caused by the highway behaviour.}\]

This stationary interaction is not unexpected. However, we demonstrate the use of Sinatra in the development of a behaviour and, in particular, how Sinatra is used to observe agents using this behaviour and to identify problems that occur.

4.2.3 Implementing the strict traffic lanes behaviour

The highway behaviour was able to prevent some stationary interactions from occurring. In particular, the use of highways to channel agents needing to travel in particular directions appeared to help organise multiple agents moving towards their goal locations.

For the strict traffic lanes behaviour, the use of highways is extended to apply to
every row and column of the grid. Alternating rows are specified for travelling east and west, while alternating columns are specified for travelling north and south. This specification of the rows and columns are referred to as ‘traffic lanes’. Each grid location will have two permitted directions of travel, one of either east or west and one of either north or south.

Using the strict traffic lanes behaviour means that an agent may not be able to move directly to its goal location. For example, an agent may be in an adjacent grid location to its goal, but due to the traffic lanes, is not permitted to move directly to this location. The agent may be required to move around up to three sides of its goal before it is in a grid location with a direction of travel that allows the agent to reach its goal.

We now describe the implementation of the strict traffic lanes behaviour. We begin by focussing on the modifications required for the Sinatra agents to be able to use the strict traffic lanes behaviour, before implementing the strict traffic lanes behaviour itself.

**Strict traffic lanes agent perceptions**

In order to use the strict traffic lanes behaviour, the Sinatra agents must be aware of the size of the grid environment. This information is stored by the Sinatra agents in their internal memory.

The Sinatra agents must also be aware in some manner of how the directions of travel in the grid environment are specified. As the direction of travel alternates between adjacent lanes, the Sinatra agent only needs to know the general pattern for odd and even numbered rows and columns. Using this information, the agent is able to work out what are the permitted directions of travel based on its current location.

The Sinatra agents are not able to move outside the boundaries of the grid environment. Due to the nature of the traffic lanes, there will be some locations around the periphery of the grid that have a specified direction of travel that would lead the agent outside of the boundaries of the grid. Therefore, a Sinatra agent must also be able to perceive when it is at the edge of the grid environment. This perception was not required for the simple goal-directed navigation or the highway behaviour, as these behaviours would not attempt to direct the agent to move outside of the environment.

Table 4.3 shows the additional perception methods that are required by the Sinatra agents in order to use the strict traffic lanes behaviour. Together these perception methods allow the Sinatra agents to determine what are the permitted directions of travel.

These new perception methods rely on the agent’s existing location perception method and facts about the domain. Therefore, the sensory capabilities of the
<table>
<thead>
<tr>
<th>Perception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>trafficLanes</td>
<td>Returns an enumerated type specifying the permitted directions of travel for the agent based on its current grid location. Uses the agent’s current location and the agent’s knowledge of the direction of travel for odd and even numbered rows and columns.</td>
</tr>
<tr>
<td>atEdge</td>
<td>Returns the edge or edges of the grid environment that the agent is next to, if any. Uses the agent’s current location and the size of the grid environment.</td>
</tr>
</tbody>
</table>

Table 4.3: The perception methods for the strict traffic lanes behaviour.

Sinatra agents do not have to be extended in order to allow the agents to execute these additional perception methods.

New tokens must be added to the Sinatra agent’s dictionary of tokens in order to allow the agent to describe its current state facts for the strict traffic lanes behaviour. Table 4.4 defines these tokens.

<table>
<thead>
<tr>
<th>Token</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>nlane, elane, slane and wlane</td>
<td>The results of the trafficLanes perception method, specifying the permitted directions of travel for the agent based on the traffic lanes.</td>
</tr>
<tr>
<td>walln, walle, walls and wallw</td>
<td>The results of the atEdge perception method, specifying when the agent is next to an edge of the grid environment. We refer to the edges of the grid environment as ‘walls’ in these tokens.</td>
</tr>
</tbody>
</table>

Table 4.4: The dictionary of tokens for the strict traffic lanes behaviour.

**Strict traffic lanes behaviour**

Listing 4.1 shows the strict traffic lanes behaviour input file.

The strict traffic lanes behaviour directs the agent to select a direction of travel from the two permitted directions for the agent’s current grid location. The agent will move in a direction that brings the agent closer to its goal, if the traffic lanes permit travel in this direction. If the traffic lanes do not permit a direction of travel that would bring the agent closer to its goal, then the agent must move in one of the permitted directions.

We have intentionally placed the directives for travelling north or south before the directives for travelling east or west. This means that, in cases where both permitted directions will bring the agent closer to its goal, the agent will select to travel along a column rather than a row. This reduces the number of turning actions required.
Listing 4.2: (part 1) The strict traffic lanes behaviour input file.
forward :- geast, wlane, nlane, walln, west.
left :- geast, wlane, nlane, walln, north.
forward :- geast, wlane, nlane, north.
left :- geast, wlane, nlane, east.
right :- geast, wlane, nlane.

forward :- geast, wlane, slane, walls, west.
left :- geast, wlane, slane, walls, north.
forward :- geast, wlane, slane, south.
left :- geast, wlane, slane, west.
right :- geast, wlane, slane.

forward :- gwest, elane, nlane, walln, east.
left :- gwest, elane, nlane, walln, south.
forward :- gwest, elane, nlane, north.
left :- gwest, elane, nlane, east.
right :- gwest, elane, nlane.

forward :- gwest, elane, slane, walls, east.
left :- gwest, elane, slane, walls, south.
forward :- gwest, elane, slane, south.
left :- gwest, elane, slane, west.
right :- gwest, elane, slane.

:- forward, left.
:- forward, right.
:- left, right.

s: 1 \{north, east, south, west\} 1.
s: 0 \{gnorth, gsouth\} 1.
s: 0 \{geast, gwest\} 1.
s: 1 \{nlane, slane\} 1.
s: 1 \{elane, wlane\} 1.
s: 0 \{walln, walls\} 1.
s: 0 \{walle, wallw\} 1.

in: north east south west gnorth geast gsouth gwest nlane elane slane wlane walln walle walls wallw.

out: forward left right.

Listing 4.1: (part 2) The strict traffic lanes behaviour input file.
by the agent, as it means that an agent will not zigzag across the grid environment when trying to reach its goal.

The strict traffic lanes behaviour can be used by the agents as a way of reaching their goal locations. Therefore, this behaviour can be used instead of the simple goal-directed navigation or the highway behaviour. Apart from specifying the use of the strict traffic lanes behaviour in the agent’s behaviour-switch, the Sinatra agents require no further modifications in order to follow the strict traffic lanes behaviour.

4.2.4 Demonstration

Figure 4.4 shows two agents in Sinatra using the strict traffic lanes behaviour. For each agent, the single highlighted grid location marks the goal of the agent. The three highlighted locations around an agent show the agent’s position at the beginning of the time step and the two permitted directions of travel from this location according to the traffic lanes. Figure 4.4 shows the simulation at the end of a time step, after the agents have already moved in one of these permitted directions.

The traffic lanes mean that there is no opportunity for agents to meet and mutually block each other, as occurs in a stationary interaction. This is because the agents must follow the directions of travel specified by the traffic lanes. The traffic lanes permit travel in opposite directions for alternating rows and columns, but only one direction of travel for each specific row and column. Therefore, the agents cannot mutually obstruct each other as they cannot meet while travelling in opposite directions. This means that the strict traffic lanes behaviour prevents all occurrences of stationary interactions.

There are some limitations to the strict traffic lanes behaviour however. The first limitation is that there is a grid location within the environment that cannot be accessed by agents that obey the traffic lanes. For the Sinatra simulation shown in Figure 4.4, the south-west corner location cannot be accessed. The directions of travel permitted in this row and column of the domain are east and north respec-
tively. Therefore, an agent is unable to enter this grid location while obeying these permitted directions of travel.

The second limitation is that there is a grid location within the environment that an agent can enter but is unable to leave due to the traffic lanes. For the Sinatra simulation shown in Figure 4.4, this location is the south-east corner. The directions of travel permitted in this row and column of the domain are east and south respectively. Therefore, although an agent is able to enter this location, it is unable to leave while obeying these permitted directions of travel. Figure 4.5 shows a situation where the red agent is stuck in the south-east corner.

We use this situation to illustrate how an exception case can be added to the strict traffic lanes behaviour, in order to resolve this undesirable situation. This exception is implemented by adding new directives to the behaviour, which allow an agent to ignore the permitted directions of travel only in the case that the agent is in the south-east corner.

The new directives $D_E$ that implement this exception case are

- left ← walls, walle, east
- right ← walls, walle, south
- forward ← walls, walle

These directives instruct an agent that is in the south-east corner of the grid to turn and to travel either north or west to move out of this location. These exception case directives must be given the highest priority of the directives in the strict traffic lanes behaviour in order for the desired exception behaviour to be implemented correctly.

The modified strict traffic lanes behaviour can be used immediately by the Sinatra agents, once the new state-action table for this behaviour has been generated. This demonstrates the use of Sinatra to develop and test behaviours in an efficient manner.

Unfortunately, allowing agents in the south-east corner to move against the direction of travel specified by the traffic lanes means that the strict traffic lanes behaviour
no longer prevents all occurrences of stationary interactions. Figure 4.6 shows an example of such a stationary interaction.

![Figure 4.6: The exception case to the strict traffic lanes behaviour allows stationary interactions to occur. Note that the blue agent has its goal location immediately to the left of the red agent.]

### 4.2.5 Summary

We have presented two examples of behaviours that can be used to prevent stationary interactions. These behaviours aim to direct the agents to reach their goal locations, whilst also directing the agents to act in a manner that reduces or prevents the occurrence of the undesirable agent interaction.

The **highway** behaviour attempted to reduce the occurrence of stationary interactions by moving agents away from the periphery of the grid while they travel to their goals. The design of this behaviour was based on the observation that stationary interactions often occur around the periphery of the grid environment.

The **strict traffic lanes** behaviour attempted to prevent the occurrence of stationary interactions by restricting the permitted directions of travel for each grid location. The design of this behaviour was based on the observation that stationary interactions involve agents who mutually obstruct each other by attempting to travel in opposite directions. The traffic lanes prevent agents travelling in opposite directions along a row or column.

By observing agents following these behaviours using the Sinatra GUI, we were able to test whether the behaviours worked as intended and to identify situations where the behaviours were unsuccessful. We were able to verify that both behaviours can be used to allow agents to reach their goal locations. In addition, it appeared that the **highway** behaviour was able to reduce the occurrence of stationary interactions, specifically around the periphery of the grid. It also appeared that the **strict traffic lanes** behaviour was able to eliminate stationary interactions altogether. However, this was not the case when the exception case was introduced.

By using Sinatra to observe the agents, we were also able to identify the situations
when stationary interactions would still occur and to attempt to extrapolate the reason for these interactions. For example, the highway behaviour caused more stationary interactions to occur when agents were travelling to and from the different highways.

### 4.3 Example: Using behaviours to resolve stationary interactions

In order to resolve undesirable agent interactions, the agents must be able to identify when the undesirable interaction is occurring and to switch to using a new behaviour. The new behaviour must direct the agent to act in a manner that removes the agent from the undesirable interaction. To demonstrate this strategy, we use the concrete scenario of behaviours that are designed to resolve stationary interactions. For this scenario, we consider two example behaviours.

In the first example, we develop the **obstacle avoidance** behaviour. This behaviour directs an agent to move around an obstacle that it encounters in its path. Therefore, if two agents participating in a stationary interaction switch to using the **obstacle avoidance** behaviour, both agents will move in order to resolve the stationary interaction.

In the second example, we develop the **traffic law** behaviour. In this behaviour, agents travelling in certain directions are given priority over other agents, who must give way. Agents that are travelling south or east must give way to agents that are travelling north or west. Therefore, to resolve a stationary interaction occurring between two agents, only one of the agents will move, while the other agent waits until its path is clear.

Both the **obstacle avoidance** behaviour and the **traffic law** behaviour are designed as behaviours that resolve stationary interactions. Therefore, these behaviours are used by Sinatra agents when the agent has identified that a stationary interaction is occurring. The agent uses a behaviour-switch to switch between different behaviours in this way; for example, from using the **simple goal-directed navigation** to using either of these new behaviours.

Similar to the example behaviours for preventing undesirable agent interactions, it is not our intention to develop an optimal set of ‘traffic rules’ for agents in a grid environment. Instead, we demonstrate the implementation of behaviours using Sinatra, where these behaviours implement an alternative strategy for managing stationary interactions.
4.3.1 Implementing the obstacle avoidance behaviour

We have not assumed any communication capabilities for Sinatra agents, nor are the agents able to perceive more information about their surroundings than whether a location is obstructed, or not. This means that, from an individual agent’s perspective, the other agents in the environment are perceived as movable obstacles that the agent cannot cross.

The obstacle avoidance behaviour is designed to allow an agent to move around an obstacle that the agent perceives as blocking its path. The exact obstacle that the agent is avoiding is not defined. The obstacle may occupy a single grid location or multiple grid locations. Equally, the obstacle may be another agent and so the obstacle may move from the position where it is first observed. Therefore, the obstacle avoidance behaviour, in being designed to allow the agent to move around a general, non-specific obstruction, will allow the agent to move out of a stationary interaction.

We now describe the implementation of the obstacle avoidance behaviour. We begin by focussing on the modifications required for a Sinatra agent to be able to use the obstacle avoidance behaviour, before implementing the obstacle avoidance behaviour itself.

Obstacle avoidance behaviour agent perceptions

In order to use the obstacle avoidance behaviour, the Sinatra agents must be able to observe when they are being obstructed. When describing the basic perception methods of Sinatra agents, we specified that the agents possessed the perception methods north, east, south and west. These perception methods allow the agent to perceive these four grid locations adjacent to the agent’s current position. By observing these grid locations, the agent is able to determine if a location is empty or obstructed in some manner. An obstruction may be a physical obstruction in the grid environment (although we do not demonstrate this type of obstruction), another agent in the environment, or the edge of the grid environment itself. Therefore, the Sinatra agents do not require any additional perception methods in order to be able to use the obstacle avoidance behaviour.

New tokens must be added to the agent’s dictionary of tokens, however, for the agent to be able to describe the state facts required by the obstacle avoidance behaviour. The tokens nobs, eobs, sobs and wobs are added to the agent’s dictionary of tokens. These tokens describe the results of the north, east, south and west perception methods, specifying the directions where the agent is obstructed. The state facts of the agent for this behaviour will use these new tokens to describe the directions that the agent is obstructed in, as well as tokens to describe the relative location of the agent’s goal.
**Obstacle avoidance behaviour-switch method**

In Section 3.5.1 we defined a behaviour-switch for an agent that is able to follow the simple goal-directed navigation and the obstacle avoidance behaviour. This behaviour-switch was called the example master behaviour-switch. Using this behaviour-switch, or a similar behaviour definition for when a different behaviour is used to direct the agent towards its goal, the agent is able to switch to using the obstacle avoidance behaviour when it observes that it is obstructed.

In Table 3.6 we defined the tokens that are used by the Sinatra agents to be able to describe the state facts required by the example master behaviour-switch. In particular, we defined the obstructed token, which specifies whether a direction that would lead to the agent’s goal is obstructed, based on the results of the north, east, south or west perception methods.

**Obstacle avoidance behaviour**

The obstacle avoidance behaviour directs the agent to attempt to move around an obstacle, whilst also attempting to allow the agent to move closer to its goal location. In this way, an agent using the obstacle avoidance behaviour is able to navigate past an obstacle in a manner that is beneficial to achieving the agent’s goal. This behaviour can be used with any of the navigation behaviours that have been developed so far. However, the behaviour is most suited for use with the simple goal-directed navigation, as the obstacle avoidance behaviour does not take into account any highways or traffic lanes that might be defined in the environment.

Listing 4.1 shows the obstacle avoidance behaviour input file. The obstacle avoidance behaviour assumes that the agent is always obstructed in a direction that would lead the agent closer to its goal.

```plaintext
right :- nobs, gnorth, geast, north.
left :- nobs, gnorth, gwest, north.
left :- nobs, eobs, gnorth, east.
right :- nobs, wobs, gnorth, west.
left :- nobs, eobs, gnorth, north.
forward :- nobs, eobs, gnorth, west.
right :- nobs, gnorth, north.
forward :- nobs, gnorth, east.
right :- nobs, gnorth, south.
forward :- nobs, gnorth, west.
```

Listing 4.2: (part 1) The obstacle avoidance behaviour input file.
right :- eobs, geast, gsouth, east.
left :- eobs, geast, gnorth, east.
right :- eobs, nosb, geast, north.
left :- eobs, sobsb, geast, south.
left :- eobs, sobs, geast, east.
forward :- eobs, sobs, geast, north.
forward :- eobs, geast, north.
right :- eobs, geast, east.
forward :- eobs, geast, south.
right :- eobs, geast, west.
	right :- sobs, gsouth, gwest, south.
left :- sobs, gsouth, geast, south.
right :- sobs, eobs, gsouth, east.
left :- sobs, wobs, gsouth, west.
left :- sobs, wobs, gsouth, south.
forward :- sobs, wobs, gsouth, east.
right :- sobs, gsouth, north.
forward :- sobs, gsouth, east.
right :- sobs, gsouth, south.
forward :- sobs, gsouth, west.

right :- wobs, gwest, gnorth, west.
left :- wobs, gwest, gsouth, west.
left :- wobs, nosb, gwest, north.
right :- wobs, sobs, gwest, south.
left :- wobs, nosb, gwest, west.
forward :- wobs, nosb, gwest, south.
forward :- wobs, gwest, north.
right :- wobs, gwest, east.
forward :- wobs, gwest, south.
right :- wobs, gwest, west.

:- forward, left.
:- forward, right.
:- left, right.

s: 1 \{north, east, south, west\} 1.
s: 0 \{gnorth, gsouth\} 1.
s: 0 \{geast, gwest\} 1.
s: 1 \{nosb, eobs, sobs, wobs\} 4.
in: north east south west gnorth geast gsouth gwest nosb eobs sobs wobs.
out: forward left right.

Listing 4.1: (part 2) The obstacle avoidance behaviour input file.
This assumption, that the agent is always obstructed in a direction that would lead the agent closer to its goal, is not easily expressed using the aggregate syntax of CLASP. Using the aggregate

\[ 1 \{ \text{nobs, eobs, sob, wobs} \} 4. \]

we are able to specify that the agent is obstructed in at least one direction. However, we are unable to specify that the agent is obstructed in a direction towards its goal location. Therefore, there will be state-action table entries for the \text{obstacle avoidance} behaviour that contain no associated action. As described in Section 3.5.2, the agent will perform the default \text{wait} action in these circumstances. This leads to a larger than necessary state-action table for this behaviour. However, as these entries correspond to states where the agent is not obstructed, the \text{obstacle avoidance} behaviour will never be used by an agent in one of these states.

To resolve this issue, we can use constraint formulae to limit the possible combinations of state facts, as we have already done to limit the possible combinations of actions and behaviours. This is easy to achieve using the behaviour definition and CLASP, but has not been an important issue and so has not been implemented.

\subsection*{4.3.2 Demonstration}

We use Sinatra to simulate agents that are able to use the \text{simple goal-directed navigation} and the \text{obstacle avoidance} behaviour. If all of the agents are able to use the \text{obstacle avoidance} behaviour, then when agents are participating in a stationary interaction, all of the participating agents will observe that their path is obstructed. The behaviour-switch will direct these agents to follow the \text{obstacle avoidance} behaviour. Therefore, when we observe agents using the Sinatra GUI, the \text{obstacle avoidance} behaviour will mean that all of the participating agents move to resolve a stationary interaction simultaneously.

Figure 4.7 to Figure 4.10 show two agents using the \text{obstacle avoidance} behaviour to resolve a stationary interaction. Each obstructed agent is directed to turn to the side and then to move around an obstacle blocking its path. When the agent’s path is clear, the behaviour-switch directs the agent to return to following the \text{simple goal-directed navigation}.

Note that it is difficult to show the actions of the agents in Sinatra. We attempt to demonstrate agents following a behaviour by using screen shots to show the positions of the agents after each time step. However, in the Sinatra visualisation of the simulation, the agents are seen moving smoothly between these states.
Figure 4.7: Two agents in Sinatra using the **obstacle avoidance** behaviour. At this point the behaviour-switch directs the agents to start using the **obstacle avoidance** behaviour.

Figure 4.8: The agents turn to the side in order to avoid the obstacle.

Figure 4.9: The agents move forwards to move around the obstructed location.
Sinatra was used to simulate different sizes of agent population using the obstacle avoidance behaviour. As demonstrated, the obstacle avoidance behaviour is able to resolve stationary interactions involving two agents and was also observed to resolve stationary interactions involving three agents.

When agents use the obstacle avoidance behaviour, however, there are situations where the agents will not always be able to reach their goals. This occurs when the stationary interaction occurs at the periphery of the grid environment. In these circumstances, the obstacle avoidance behaviour and the simple goal-directed navigation were observed to cause the agents to participate in another type of undesirable interaction.

Rather than mutually obstructing each other’s path and so remaining stationary, the agents were now observed to obstruct each other’s path despite the agents moving to different locations. The combination of the obstacle avoidance behaviour and the simple goal-directed navigation lead to agents moving in a cyclical manner between a small number of locations within the grid environment. We refer to this as a repeated state interaction.

Figure 4.11 to Figure 4.17 show an example of a repeated state interaction. Although the agents are moving to attempt to resolve a stationary interaction or to move towards their goals, the agents repeatedly observe that they are obstructed. This undesirable interaction is caused by the proximity of the agents to the edge of the grid environment, which restricts their ability to move. Therefore, the agents enter into a cycle of moving between the same grid locations.

This undesirable interaction was not predicted when designing the directives of the obstacle avoidance behaviour. Therefore, the repeated state interaction is an example of an undesirable emergent behaviour of the agents, obvious only in hindsight. This demonstrates how Sinatra can be used to identify emergent behaviour of the agents.
Figure 4.11: Two agents in Sinatra using the **simple goal-directed navigation**. The behaviour-switch directs the agents to start using the **obstacle avoidance** behaviour.

Figure 4.12: The agents turn to the side to avoid each other, however, because they are next to the edge of the grid, both agents turn to face south.

Figure 4.13: The agents move forward to try to move around the obstructed location.
Figure 4.14: The behaviour-switch instructs the agents to use the simple goal-directed navigation. However, the agents again observe that they are obstructed.

Figure 4.15: The behaviour-switch instructs the agents to return to using the obstacle avoidance behaviour to attempt to resolve the stationary interaction. The agents both have an unobstructed direction that will bring them closer to their goal. Therefore, the obstacle avoidance behaviour instructs both of the agents to turn to face north.

Figure 4.16: The agents move forwards in the direction that is unobstructed and brings them closer to their goal.
4.3.3 Implementing the traffic law behaviour

The obstacle avoidance behaviour directs all the agents participating in a stationary interaction to move in order to resolve the interaction. However, it is not necessary for all of the agents to move in order for the interaction to be resolved. For example, in a stationary interaction involving two agents, only one of the agents needs to move out of the way in order for both agents to be released from the stationary interaction.

Therefore, we consider the use of traffic laws to resolve stationary interactions, which can be used to instruct agents to give way to oncoming agents under certain circumstances. We specify the following traffic law.

Agents travelling north or west take priority when agents interact. Therefore, agents travelling south or east must give way to agents travelling north or west.

This traffic law is a norm, consisting of a system norm in the first sentence and an agent-specific norm in the second sentence [CS08, Ser08]. By implementing the agent specific norm in terms of directives, the agent is able to take the agent-specific norm into account when determining its actions.

The result of implementing this traffic law as a behaviour is that not all of the agents participating in a stationary interaction will take action in order to resolve the interaction. Agents travelling south or east will move to resolve the interaction, whereas agents travelling north or west will wait for the other agent or agents to move.

We now describe the implementation of the traffic law behaviour. We begin by focussing on the modifications required for a Sinatra agent to be able to use the traffic
law behaviour. We then describe the implementation of the traffic law behaviour itself.

**Traffic law agents perceptions**

Using the perception methods north, east, south and west, the Sinatra agents are able to determine in which directions they are obstructed. New sensory capabilities are required to allow an agent to identify when it must give way.

To determine whether it must give way, the agent must first be able to perceive whether it is obstructed by another agent. The agent may also be obstructed by a physical obstruction in the environment or by the edge of the environment itself. Therefore, the sensory capabilities of the agent must be extended to allow the agent to perceive if an obstruction is another agent. In addition, the agent must also be able to perceive the direction of travel of an obstructing agent.

By combining these sensory capabilities, the Sinatra agent is able to determine if it must give way. An agent must give way if it is obstructing an agent travelling north or west. Therefore, the agent uses these additional capabilities to test if an obstruction to the south or east is an agent travelling north or west respectively.

The perception methods required by the Sinatra agents in order to use the traffic law behaviour are shown in Table 4.5. For clarity, these perception methods include the existing agent perceptions used to allow the agent to identify when it is obstructed.

<table>
<thead>
<tr>
<th>Perception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>north, east, south and west</td>
<td>Allows the agent to perceive the grid locations in these directions, adjacent to its current position. The agent can determine if a location is empty or obstructed in some manner.</td>
</tr>
<tr>
<td>giveWay</td>
<td>Returns whether the agent needs to give way to another agent. Called whenever the agent is obstructed to the south or the east. This method identifies whether the agent is obstructed in these directions by another agent and the direction of travel of this agent.</td>
</tr>
</tbody>
</table>

Table 4.5: The perception methods for the traffic law behaviour.

New tokens must be added to the agent’s dictionary of tokens, in order for the agent to be able to use the traffic law behaviour. Table 4.6 describes these tokens. The tokens introduced for agents following the obstacle avoidance behaviour, which are also required for the traffic law behaviour, are included here for completeness.

The state facts of the agent for the traffic law behaviour will use the tokens to describe the directions that the agent is obstructed in, as well as tokens to describe
nobs, eobs, sobs, and wobs

<table>
<thead>
<tr>
<th>Token</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>nobs, eobs, sobs, and wobs</td>
<td>The results of the north, east, south and west perception methods respectively, specifying the directions where the agent is obstructed.</td>
</tr>
<tr>
<td>giveaway</td>
<td>The result of the giveaway perception method when the agent is obstructing an agent to the south or east and must give way to this agent.</td>
</tr>
</tbody>
</table>

Table 4.6: The dictionary of tokens for the traffic law behaviour.

the direction the agent is facing. The giveaway token is used by the agent to determine when to switch to using the traffic law behaviour.

**Traffic law behaviour-switch method**

A behaviour-switch is used by the agent to determine when to use the traffic law behaviour. An agent should use the traffic law behaviour when it has identified that it must give way to another agent using the giveaway perception method. Listing 4.2 shows the behaviour-switch for an agent that is able to use the simple goal-directed navigation and the traffic law behaviour.

```
newgoal :- atgoal.
trafficlaw :- giveaway.
goaldirected :- .

:- newgoal, trafficlaw.
:- goaldirected, trafficlaw.
:- newgoal, goaldirected.

s: 0 {giveaway, atgoal} 2.

in: giveaway atgoal.

out: trafficlaw goaldirected newgoal.
```

Listing 4.2: The behaviour-switch input file for the traffic law behaviour.

This behaviour-switch is similar to the behaviour-switch defined for agents using the obstacle avoidance behaviour. Using this behaviour-switch, or a similar behaviour definition for when a different navigation behaviour is used to direct the agent towards its goal, the agent is able to switch to using the traffic law behaviour when it observes that it must give way.

Listing 4.3 shows the state-action table resulting from applying the STAG process to the behaviour-switch shown in Listing 4.2. The trafficlaw token is defined in
the agent’s dictionary of tokens to represent the traffic law behaviour.

\[
giveaway \atgoal \rightarrow \newgoal
\]
\[
giveaway \rightarrow \trafficlaw
\]
\[
\atgoal \rightarrow \newgoal
\rightarrow \goaldirected
\]

Listing 4.3: The state-action table generated from the behaviour-switch for the traffic law behaviour.

**Traffic law behaviour**

The traffic law behaviour is called by an agent when it observes that it must give way to another agent. In order to obstruct an agent travelling north or west, the agent using the traffic law behaviour must itself be obstructed to the south or to the east. The agent occupying the grid location to the south or to the east of an agent using the traffic law behaviour will be travelling north or west respectively and so cause the agent to need to give way.

Listing 4.4 shows the traffic law behaviour input file. The traffic law behaviour only contains directives for agents that are obstructed to the south or to the east, although the agent may additionally be obstructed in other directions. Therefore, the traffic law behaviour assumes that the agent using the behaviour needs to give way to another agent.

Unlike the obstacle avoidance behaviour, however, we are able to express the assumption made by the traffic law behaviour using the aggregate syntax of CLASP. The aggregates

\[
s: 1 \{eobs, sobs\} 2.
\]
\[
s: 0 \{nobs, wobs\} 2.
\]

specify that the agent is always obstructed to the south or to the east and possibly in both of these directions, while the agent may or may not also be obstructed to the north or to the west.

The aggregates used in the traffic law behaviour mean that we are able to define the correct state facts for when the traffic law behaviour will be used by an agent. Therefore, there are no redundant entries in the state-action table for the traffic law behaviour.
wait :- nobs, eobs, sobs, wobs.
left :- eobs, sobs, south.
left :- eobs, sobs, east.
forward :- eobs, sobs, north.
right :- eobs, east.
forward :- eobs, south.

left :- sobs, wobs, west.
left :- sobs, wobs, south.
forward :- sobs, wobs, east.
right :- sobs, south.
forward :- sobs, west.

:- forward, left.
:- forward, right.
:- forward, wait.
:- left, right.
:- left, wait.
:- right, wait.

s: 1 {north, east, south, west} 1.
s: 1 {eobs, sobs} 2.
s: 0 {nobs, wobs} 2.

in: north east south west nobs eobs sobs wobs.
out: forward left right wait.

Listing 4.4: The traffic law behaviour input file.

4.3.4 Demonstration

We use Sinatra to simulate agents that are able to use the simple goal-directed navigation and the traffic law behaviour. When participating in a stationary interaction, the behaviour-switch will direct the agents that must give way to resolve the stationary interaction by using the traffic law behaviour. The agents travelling north or west, who are not expected to give way, will continue to follow the simple goal-directed navigation. Therefore, these agents are unable to move until the stationary interaction is resolved.

Figure 4.18 to Figure 4.21 show two agents participating in a stationary interaction, where the agents are implemented to be able to use the traffic law behaviour. The red agent that is obstructed to the east is directed to move to the side by the traffic law behaviour. The blue agent that is obstructed to the west continues to follow the simple goal-directed navigation. When the red agent has moved as directed
by the traffic law behaviour, both agents will find that they have a clear path to reach their goal locations and are not required to give way to another agent. Therefore, the red agent returns to following the simple goal-directed navigation.

Figure 4.18: Two agents in Sinatra participating in a stationary interaction. At this point the behaviour-switch directs the red agent to start using the traffic law behaviour as it must give way to the blue agent travelling west.

Figure 4.19: The red agent turns to the side, away from the edge of the grid. The blue agent continues to follow the simple goal-directed navigation and is unable to move.

Figure 4.20: The red agent moves forwards, in order to give way to the blue agent.
Figure 4.21: The agents are no longer obstructed, nor required to give way and so
the behaviour-switch directs both agents to use the simple goal-directed
navigation.

Sinatra was used to simulate different sizes of agent population (between two and
six agents) using the traffic law behaviour. As demonstrated, the traffic law behaviour
is able to resolve stationary interactions involving two agents and was also observed
to resolve stationary interactions involving three agents.

When agents use the traffic law behaviour, however, it is possible for the agents
to find themselves in situations where the stationary interaction cannot be resolved.
This can occur when an agent that must give way is unable to move. An example of
such a situation is when the stationary interaction involves an agent in the north-
west corner of the grid environment, where this agent is unable to move to the south
or to the east. Figure 4.22 shows this situation.

Figure 4.22: A stationary interaction that cannot be resolved by the traffic law be-
haviour. The red agent is being directed to give way to the blue agent
and to the green agent, but is also being prevented from moving out of
the way by these agents.

The red agent in the north-west corner is using the traffic law behaviour, as it
must give way to the agents travelling north and west. However, the red agent
cannot move outside of the grid environment and so is unable to move to resolve
the stationary interaction. Therefore, the red agent is directed to wait by the traffic
law behaviour, as shown by the first directive in Listing 4.4. By waiting, the hope is that some other event may occur to allow the stationary interaction to be resolved.

The two other agents are still directed by the behaviour-switch to follow the simple goal-directed navigation. Therefore, these agents are still attempting to move forwards but are unable to move, meaning that these agents will not act to resolve the stationary interaction. In this situation, there is no opportunity for the agents to be able to resolve the stationary interaction using the traffic law behaviour.

4.3.5 Summary

We have presented two examples of behaviours that can be used to resolve stationary interactions between agents. A behaviour-switch was required to allow the agents to use these new behaviours in conjunction with the simple goal-directed navigation, or any other navigation behaviour. Agents switched to using the new behaviours when they observed specific conditions in their current state. These conditions suggested that the agent was participating in a stationary interaction and was in a situation where the new behaviours were applicable.

The obstacle avoidance behaviour attempted to resolve stationary interactions by moving an agent around an unknown obstacle in the environment. This obstacle may be another agent, in which case the obstacle avoidance behaviour allows the agents participating in a stationary interaction to move around each other and so resolve the undesirable interaction.

The traffic law behaviour attempted to resolve stationary interactions by implementing a traffic law within the multi-agent system. The traffic law specified that when agents interact, agents travelling north or west should take priority over agents travelling south or east. Agents that are obstructing an agent travelling north or west must move to the side to allow this agent to continue travelling. Therefore, the traffic law behaviour allows agents participating in a stationary interaction to resolve the interaction by specifying conditions when the agents must give way.

By observing agents following these behaviours using the Sinatra GUI, we were able to test whether the behaviours worked as intended and to identify situations where they were unsuccessful. We were able to verify that both behaviours can be used to resolve stationary interactions. We also observed, however, that agents may misidentify a stationary interaction and be directed to move unnecessarily.

The behaviour-switch directs the agents to use the obstacle avoidance behaviour or the traffic law behaviour when certain conditions hold in their current state. These conditions are used to identify that an agent is participating in a stationary interaction. However, due to the limited local perception of the agents, the agents may incorrectly identify a stationary interaction.

This can occur for both behaviours where an agent is obstructed by (and if appropriate, must also give way to) another agent that is not itself obstructed. For
example, this situation arises when an agent meets another agent that is just about
to turn and move in a new direction towards its goal, rather than continue to attempt
to move forwards in the same direction. In this situation, the behaviour-switch will
instruct the obstructed agent to switch to using the obstacle avoidance behaviour
or the traffic law behaviour, as appropriate. These behaviours will then direct the
obstructed agent to move, even though no stationary interaction is occurring and
the interaction would resolve itself in time.

By using Sinatra to observe the agents, we were also able to identify situations
where the stationary interactions were not resolved correctly. In the case of the
obstacle avoidance behaviour, these situations resulted in repeated state interactions.
This new undesirable interaction was not predicted when developing the obstacle
avoidance behaviour and so demonstrates the use of Sinatra to identify unexpected
emergent behaviour of the agents.

4.4 Conclusion

We now discuss the results of our experiments in terms of using Sinatra to implement
a norm-governed multi-agent system, where the agents are implemented to follow
behaviours. We also highlight some general issues related to the use of behaviours
to manage agent interactions.

4.4.1 Using Sinatra to implement behaviours

We have demonstrated the implementation of new behaviours in Sinatra, where
agents following the directives of these behaviours can be observed and their inter-
actions replayed using the Sinatra GUI. These behaviours are designed to manage
the interactions of agents and form the implementation of norms used by the agents.

For each of the four behaviours developed in this chapter, we have discussed the
modifications required to the Sinatra agents in order for them to be able to use the
new behaviours. In all cases, these modifications appear to be reasonable additions
to the agent capabilities, as they require methods that would be easy to implement
in a simple robotic device.

- We have described the implementation of new perception methods and sensory
capabilities, where necessary. These methods are used to determine the current
state of the agent when consulting the state-action table for the new behaviour
and are encoded in the agent’s native programming language, in this case Java.

- We have defined the new tokens that are required in the agent’s dictionary of
tokens to allow the agent to understand the state-action table entries for the
new behaviour.
We have described the modifications required to a behaviour-switch in order to allow the agents to use the new behaviour. In addition, new perception methods, sensory capabilities and tokens may be required to allow the behaviour-switch to identify when the new behaviour should be used.

We have also shown the behaviour input file that is used by the STAG process to generate the state-action table for each behaviour. Having made the required modifications to the Sinatra agents, the generated state-action table can be given directly to the agents, allowing the agents to start to follow the directives of a new behaviour immediately. Therefore, the Sinatra test bed allows a behaviour to be implemented and tested efficiently.

Updates and modifications to a behaviour can also be made in a timely manner. For example, to allow the Sinatra agents to switch from using the original strict traffic lanes behaviour to the strict traffic lanes behaviour including the exception case, the strict traffic lanes behaviour input file was modified to include the extra directives. The modified input file was then given as input to the STAG process. The resulting state-action table could be used immediately by the Sinatra agents, allowing the process of verifying and testing the modified behaviour to start instantly. This rapid implementation and testing cycle means that Sinatra is an efficient tool for the development of behaviours.

Having implemented a behaviour, Sinatra allows the user to observe the effects of the behaviour by simulating agents following these directives. We have used these simulations to demonstrate agents using the new behaviours described in this chapter. For the behaviours to resolve stationary interactions, we have chosen to demonstrate these behaviours for agents that are also implemented to use the simple goal-directed navigation. Using the simulation visualisation in the Sinatra GUI, the agents are seen to act as directed by the new behaviours, including switching between a new behaviour and the simple goal-directed navigation, when appropriate.

By observing agents in Sinatra using the new behaviours, we have been able to identify situations where the behaviours operate successfully and situations where the behaviours are not successful. This has allowed us to observe unanticipated emergent behaviour of the agents that was caused by the new behaviours. For example, we were able to identify the repeated state interaction, which was not foreseen when implementing the obstacle avoidance behaviour.

In the case of the repeated state interaction, the replay method of the Sinatra GUI proved particularly useful for analysing the interactions. By pausing the simulation when a repeated state interaction has occurred, the replay method allows the user to step forwards and backwards between the actions of the agents during this interaction. This allows the user to concentrate on the actions of each agent individually and so the different current states and active behaviours of the agents involved in
the interaction can be identified. When the simulation is running in a continuous manner, there is often not sufficient time to observe all of this information.

Executing the simulation in a continuous manner using the Sinatra GUI, however, is useful for allowing the user to observe other aspects of agents using the new behaviours. The visualisation animates the actions of the agents, showing the agents physically moving between different grid locations and the concurrent actions of agents during each time step. By observing a simulation of agents using a particular behaviour in this way, the user is able to identify whether a behaviour generally works as intended and to gain an understanding of the frequency of undesirable agent interactions that occur while the agents use this behaviour.

From these experiments, we have shown that Sinatra can be used as a fully functional tool for the development and testing of behaviours. The modifications required for the Sinatra agents to be able to use a new behaviour can be clearly identified. Having made these modifications, the Sinatra test bed allows a behaviour to be implemented and tested efficiently, where agents are able to start using a newly generated state-action table immediately. The Sinatra GUI and replay methods allow the user to verify if the behaviour directs the Sinatra agents to act as intended and to identify if any modifications are required.

Therefore, we have been able to demonstrate the use of Sinatra to implement behaviours. These behaviours allow the agents in Sinatra to take norms into account when determining their actions. We are able to use Sinatra and, in particular, the Sinatra visualisation of a simulation, to investigate what happens when different behaviours are adopted by the agents in the multi-agent system.

4.4.2 Using Sinatra to implement fallible agents

An agent may fail to follow the directives of a behaviour because of intentional and unintentional violations. In this section, we describe how Sinatra can be used to implement these different types of violations and, therefore, to explore their effects on the multi-agent system.

An agent may intentionally fail to act as directed by a behaviour because of a higher priority behaviour or action that the agent executes instead. This intentional violation is implemented in Sinatra using a behaviour-switch, where the priority order between the directives is used to allow the agent to determine which actions to perform. By changing the relative priority of the directives in the behaviour-switch, Sinatra can be used to investigate what happens when agents select certain actions over others.

It should be noted that the agents in Sinatra are not all required to follow the same set of behaviours. Therefore, a different behaviour-switch can be implemented for different agents, allowing agents using alternative priority orders to be simulated in the same multi-agent system.
There are a range of possible reasons why an agent may unintentionally fail to follow the directives of a behaviour. Unintentional violations include

- Mistakes in the implementation of the behaviour by the system designer. This can cause the agent to execute an unexpected action for a particular agent state.

- Mistakes in the implementation of the agent’s perception methods. This can cause the agent to misidentify its current state.

- Mistakes in the implementation of the agent’s action methods. This can cause the agent to fail to execute a particular action successfully.

- Problems during the execution of the agent’s action methods. For example, the agent’s wheels slipping on the floor surface. This can cause the agent to fail to execute a particular action successfully.

Mistakes in the implementation of a behaviour can be implemented in Sinatra by using the Stag process to generate a state-action table for a behaviour that has been modified in some way. This can be done by changing some of the actions associated with the directives, by introducing new directives or removing existing directives, or by changing the relative order of the directives within the behaviour. As Sinatra agents are able to switch to using a state-action table for a modified behaviour immediately, this type of violation can be easily implemented and investigated.

The remaining instances of unintentional violations can be classified together as being due to faulty perception and action methods. From the point of view of an observer of the Sinatra simulation, the reason for the error cannot be distinguished. It is just observed that the agent’s methods are sometimes or always unreliable.

Sinatra is able to implement faulty perception and action methods by introducing a random chance of failure into these method implementations. A random number is generated whenever a perception or action method is called by an agent. If the random number is below a certain threshold, then the method will execute as normal. If the random number is above this threshold, the method will immediately return, preventing the agent from perceiving the environment using this method or from executing an action.

The value of the threshold for each method reflects the type or frequency of the violation that is being simulated. In an extreme case by setting a threshold of 0, where a method will never execute, methods that are completely unable to function can be investigated. By varying the threshold, Sinatra can be used to investigate the effect of errors that manifest as common and intermittent violations on the part of the agent.
4.4.3 Managing undesirable agent interactions

We have shown the development of behaviours to prevent stationary interactions and behaviours to resolve stationary interactions. In this section, we use these experiments to draw some conclusions about using behaviours, implemented using directives, to manage undesirable agent interactions.

Behaviours to prevent undesirable agent interactions

Behaviours that prevent undesirable agent interactions must be used in place of any other behaviour that allows the undesirable interaction to occur. This means that the behaviours must also direct the agent to carry out the functionality implemented in the replaced behaviours. For example, the highway behaviour and strict traffic lanes behaviour were used in place of the simple goal-directed navigation. Therefore, these behaviours were also required to direct the agent to reach its goal.

If an agent does not comply with the highway behaviour or the strict traffic lanes behaviour, then the behaviours may not be successful at preventing stationary interactions from occurring. For example, as shown by the modified strict traffic lanes behaviour, allowing an agent not to follow the traffic lanes results in the possibility for stationary interactions to occur. It appears that in order for a strategy to prevent undesirable agent interactions from occurring to succeed, all of the agents in the system must follow the same strategy.

Based on these examples, behaviours that prevent undesirable agent interactions appear to be effective but only in a limited manner. Behaviours can be implemented that generally allow an agent to achieve its goals and also reduce the occurrence of an undesirable agent interaction. However, designing a behaviour that is able to prevent all occurrences of an undesirable agent interaction, whilst also implementing the functionality of the behaviour or behaviours that it is replacing, is a difficult challenge.

Behaviours to resolve undesirable agent interactions

Behaviours that resolve undesirable agent interactions are used only when the agent perceives that it is participating in the undesirable interaction. Therefore, to implement a behaviour to resolve an undesirable interaction, one option is for the system designer to add the resolution directives to an existing behaviour. The resolution directives must be given higher priority than the original behaviour directives, so that the resolution directives are used in preference to the regular actions of the agent.

We described this implementation strategy when we introduced the exception for the strict traffic lanes behaviour. However, this implementation strategy can quickly
lead to a very large and complicated behaviour definition, as well as the requirement to modify every behaviour that needs to use a particular resolution behaviour.

Therefore, we implemented behaviours to resolve undesirable agent interactions as separate behaviours. This means that the other behaviours used by the agent remain unchanged. For example, the obstacle avoidance behaviour and the traffic law behaviour were used in conjunction with the simple goal-directed navigation. Instead of modifying the existing behaviours, a behaviour-switch is used to tell the agent when to switch to using the resolution behaviour. This implementation strategy leads to simpler behaviours, that are more modular and can be used together with fewer dependencies.

If an agent does not comply with the obstacle avoidance behaviour, then the behaviour may still be able to resolve a stationary interaction successfully. This is because the obstacle avoidance behaviour will direct all agents participating in a stationary interaction to move to resolve the interaction. Therefore, as long as some of the participating agents comply with the obstacle avoidance behaviour, the stationary interaction can still be resolved.

If an agent does not comply with the traffic law behaviour, however, then the behaviour may not be able to resolve stationary interactions. As only some of the participating agents are directed to use the traffic law behaviour, if these agents do not comply with the behaviour, then the traffic law behaviour may be unable to resolve the interaction. This was shown in Figure 4.22, where the agent following the traffic law behaviour is unable to move and the stationary interaction cannot be resolved.

Therefore, it appears that for a behaviour to be able to resolve undesirable agent interactions, some but not necessarily all of the participating agents must comply with the behaviour. How many participating agents are required to comply with the behaviour will depend on the particular resolution strategy that is implemented.

Based on these examples, behaviours that resolve undesirable agent interactions also appear to be effective but only in a limited manner. Behaviours can be implemented that allow agents to resolve most occurrences of an undesirable interaction. However, there are always instances of the undesirable interaction that the behaviour is unable to resolve successfully. There is also the potential for emergent behaviour to cause different undesirable agent interactions to occur. Therefore, designing a behaviour that is able to resolve all occurrences of an undesirable agent interaction is also a difficult challenge.

### 4.4.4 When behaviours are unsuccessful

We have identified, at least for the behaviours that have been developed in this chapter, that designing directives that are able to manage all instances of stationary interactions is a very difficult challenge. While it appears to be relatively simple
to implement a behaviour that is able to manage stationary interactions to some extent, developing a behaviour that is able to manage all occurrences of a stationary interaction, whilst also allowing the agents to achieve their goals, has not been possible.

From our observations of the behaviours using the Sinatra GUI, it appears that behaviours are unsuccessful at managing undesirable agent interactions when something contributes to the undesirable agent interaction to make it more complex. An example of such a factor may be that there are additional agents participating in the interaction, beyond the minimum two agents. Alternatively, the undesirable agent interaction may be taking place close to the periphery of the environment, where the movement of agents is more restricted.

A behaviour provides a general strategy for agents to manage an undesirable agent interaction. If the undesirable agent interaction is more complicated, then the general strategy may not be effective in this case and so the behaviour is unable to handle the agent interaction.

A potential solution to this problem requires the system designer to identify the situations where a behaviour is not operating successfully. The system designer can then either develop additional directives to add to the behaviour or implement a new behaviour to handle the situations not successfully handled by the current behaviour.

**Using additional directives**

We have demonstrated the use of additional directives defined to handle a situation not correctly managed by the *strict traffic lanes* behaviour. We identified that agents using the *strict traffic lanes* behaviour can become trapped in the south-east corner of the grid environment. Three new directives were introduced as an exception case, to allow agents using this behaviour to be able to move out of the south-east corner.

Using these additional directives, we have demonstrated that the *strict traffic lanes* behaviour is now able to resolve the situation when an agent is trapped in the south-east corner. However, we also demonstrated that the additional directives can lead to further undesirable agent interactions occurring, as the stationary interaction in Figure 4.6 shows.

**Using additional behaviours**

An additional behaviour could be used to handle situations not correctly managed by the *traffic law* behaviour. We identified that agents using the *traffic law* behaviour can fail to resolve a stationary interaction when the agent that must give way is unable to move.

To handle this situation, agents that are travelling north or west could monitor
their own actions. If an agent travelling north or west finds that it has been unable to move for a specified number of time steps, then the agent switches to using the **obstacle avoidance** behaviour to attempt to resolve the stationary interaction.

The use of this additional behaviour is implemented using a behaviour-switch. The token **waited** is used to specify when the agent is travelling north or west and has been unable to move for a specified number of time steps. For example, the system designer may specify that an agent travelling north or west should not be obstructed for more than three time steps.

Using this token, a new behaviour-switch rule is created to specify when an agent travelling north or west should switch to using the **obstacle avoidance** behaviour. The new behaviour-switch for agents using the **traffic law** behaviour becomes

\[
\begin{align*}
\text{newgoal} & \leftarrow \text{atgoal} \\
\text{trafficlaw} & \leftarrow \text{giveaway} \\
\text{obstacle} & \leftarrow \text{waited} \\
\text{goaldirected} & \leftarrow \\
\phantom{\text{goaldirected}} & \leftarrow \text{newgoal, trafficlaw} \\
\phantom{\text{goaldirected}} & \leftarrow \text{goaldirected, trafficlaw} \\
\phantom{\text{goaldirected}} & \leftarrow \text{obstacle, trafficlaw} \\
\phantom{\text{goaldirected}} & \leftarrow \text{newgoal, goaldirected} \\
\phantom{\text{goaldirected}} & \leftarrow \text{obstacle, goaldirected} \\
\phantom{\text{goaldirected}} & \leftarrow \text{newgoal, obstacle}
\end{align*}
\]

Using this behaviour-switch, the agents shown in Figure 4.22 will be able to resolve the stationary interaction. The blue agent travelling north and the green agent travelling west will both switch to using the **obstacle avoidance** behaviour after they have been stationary for three time steps. A stationary interaction such as that shown in Figure 4.18 to Figure 4.21 will be resolved as normal, however, using the **traffic law** behaviour only.

By introducing this modified behaviour-switch, the **obstacle avoidance** behaviour allows stationary interactions not handled by the **traffic law** behaviour to be resolved. However, further undesirable agent interactions can occur when a new behaviour is introduced.

We have demonstrated that repeated state interactions can occur when agents use the **obstacle avoidance** behaviour. This is also true for when the **obstacle avoidance** behaviour is used in the behaviour-switch above. As situations where the **traffic law** behaviour is unable to resolve the stationary interaction are likely to occur at the periphery of the grid environment, there is a greater probability that agents now using the **obstacle avoidance** behaviour will also participate in repeated state interactions.
Limitations of local rules

It appears from our experiments that using behaviours to manage undesirable agent interactions has only a limited utility. A behaviour can be created to manage some, even most, undesirable agent interactions but is unable to manage all undesirable agent interactions without introducing further undesirable situations that need managing.

Further directives or new behaviours can be developed specifically for these situations and can be seen to reduce the number of undesirable agent interactions that are not successfully managed. However, it does not appear possible to eliminate undesirable agent interactions altogether. Indeed, our experiments have shown that, again, further undesirable agent interactions are likely to emerge.

Directives appear to be successful in a general situation but unable to guarantee a global behaviour in all cases, particularly when the situation is more complicated. The reason for this difficulty is because of the design of directives themselves. Directives are local rules intended to bring about global behaviours amongst a group of agents. Therefore, the directives are applied by independently acting agents, based on only their local field of perception.

This means that individual agents involved in an undesirable agent interaction may misinterpret the situation because they are relying only on their individual perception of the interaction. Independently acting agents, without communication facilities, are unable to perceive the full extent of an interaction that is occurring. This is true even if the agent’s perceptions were extended to encompass the whole environment, as the agent is still unable to predict the autonomous actions that will be performed by the other agents in the domain until they have been executed.

It appears that directives are inherently fallible. By introducing additional directives or behaviours to resolve situations that are not correctly handled by an existing behaviour, the remaining undesirable agent interactions become less frequent but more complex. As undesirable agent interactions become more complex, finding a solution using directives becomes more difficult and the chance for mistakes to occur on the part of the agent or the system designer increases.

We must conclude that, using only locally applied directives, there will be situations where agents are unable to act as desired by the system designer.
5 Local rules and centralised control

The difficulty of undesirable agent interactions not successfully handled by a set of directives, as described in the previous chapter (Section 4.4.4), was a persistent feature of our implementation experiences in Sinatra. Directives appear unable to manage all agent interactions successfully. As may seem obvious when considering the implementation of norms, local rules combined with the local perceptions of agents are unable to realise globally desired behaviours in all circumstances. Continually developing more specific directives to handle agent interactions not successfully managed by the existing directives is neither a practical nor a complete solution to the problem.

The resulting premise, that norms may not be a complete solution to managing agent interactions in multi-agent systems, has already been suggested by others. As noted by Grossi et al. when considering enforcement as a method for norm compliance, a paradox can be said to exist when implementing norms.

“In order to implement norms, it is likely to need more norms” [GGT10] p.221.

The existence of this paradox suggests that local rules are insufficient to manage all agent interactions. Instead, a global control mechanism may be required.

Grossi et al. describe a possible solution to the paradox that they identified. An additional compliance strategy is proposed, to be used in conjunction with the existing norms. Grossi et al. outline the use of full regimentation or automatic enforcement mechanisms as potential additional compliance strategies [GGT10]. However, both of these strategies would require global control of the multi-agent system.

The use of norms to manage all agent interactions has also been suggested as impractical. Vanhée et al. highlight the potential for an explosion of extra rules when considering ways to implement norms for BDI agents. They note that

“[i]n practice, most of these rules are rarely triggered, because their activation condition can be very rare. In reasoning about norms on the fly, we should avoid the design of many rules for improbable situations” [VAD11] p.16.

Vanhée et al. recommend careful consideration of the norms that are implemented in a multi-agent system.
“Caution has to be exercised since defining a system that integrates all norm aspects described in this [their] paper will greatly increase the complexity and amount of code. It is therefore important to check, in the design phase, which aspects are really needed, what their cost is and the arising technical issues” [VAD11] p.16.

Creating norms for every situation can require a large amount of time to implement, while a simpler set of norms may be implemented much faster and be just as effective in most circumstances. In addition, the design and implementation of a simpler set of norms helps to keep the work required to implement a multi-agent system to a manageable level.

Therefore, it would appear that the use of norms to handle interactions not successfully managed by existing norms can quickly lead to diminishing returns. Having developed an initial set of norms, new norms that are created to fill the gaps left by the initial set are often rarely used. In addition, all norms have the potential to require further norms, or an alternative mechanism, to manage resulting (emergent) undesirable interactions.

For norms implemented using directives, the development of new directives to handle interactions not successfully managed by existing directives can be a time consuming process. First, the situations where the agents interact in an undesirable manner must be identified. Observation of the undesirable interaction using the Sinatra GUI is required to allow the system designer to understand how the situation came about and which behaviours are interacting to make the situation undesirable. The system designer must then determine how the agents should resolve or prevent this interaction in the future and define this in terms of the actions and perceptions available to the agents. These actions are implemented as an ordered set of directives, which are added to an existing behaviour or created as a new behaviour specifically for this situation.

This process must be repeated to identify and resolve each undesirable interaction that may occur. As more agent interactions are handled successfully, the occurrence of the remaining undesirable agent interactions becomes rarer. However, undesirable interactions that do still occur may be complicated by the variety of different behaviours the agents are following.

The limited local perception of the agents means that they may find it hard to identify which undesirable agent interaction is occurring. The actions of the other agents are also less predictable because there are more behaviours available to the agents. As a consequence, the directives designed to handle these interactions must be highly specific and carefully created to ensure that they work as designed. When many behaviours are in use within the multi-agent system, there is an increased risk that the directives themselves will lead to new undesirable interactions occurring.
Therefore, using directives to resolve undesirable agent interactions that occur despite an existing set of behaviours in Sinatra also appears to have quickly diminishing returns, in the manner described by Grossi et al. and Vanhée et al. It will lead to a large number of extra directives, most of which will be rarely used and which may easily lead to new undesirable agent interactions or cause existing undesirable agent interactions to become more complicated. Most importantly, it would appear that local rules must eventually be replaced by a control mechanism with a global perspective, in order to achieve the desired global result.

In this chapter, we look at a new strategy for using directives to manage agent interactions. A centralised control mechanism, or controller, is used for situations where the existing directives are unable to manage agent interactions successfully. The controller uses a global perspective of the multi-agent system, or relevant fragment thereof, in order to manage an interaction. We begin by outlining this new strategy and the requirements for its implementation. We then illustrate the implementation of a controller using two examples.

5.1 Centralised control mechanism

Instead of relying solely on directives, let us consider a pragmatic strategy for using behaviours to manage agent interactions. Behaviours should be developed with the intention of being able to manage the majority of agent interactions, where a simple set of directives, similar to those already described in Chapter 4, are very effective. Situations where the directives are not successful at managing the agent interactions should be expected. In these situations, a centralised control mechanism is used to prevent or to resolve the undesirable agent interactions. Control then returns to the individual agents.

A centralised control mechanism will require a controller in order to be implemented. The controller may be implemented as an internal entity, potentially another type of agent within the system; or as an external entity, possibly the system designer, directing the agents. In order to integrate the controller within Sinatra, we will be experimenting with implementing the controller as a separate agent or entity within the multi-agent system. The controller will have a global view of the multi-agent system, or some relevant fragment thereof.

A controller is implemented by the system designer in order to handle different types of agent interactions or undesirable states, depending on the requirements of the multi-agent system. In the experiments discussed in this chapter, a controller will be implemented to prevent or to resolve a particular type of undesirable agent interaction. The controller has a global view of the interaction and the ability to issue instructions to the agents involved, telling them how to act. Therefore, the agents involved in the undesirable interaction must give up part of their autonomy.
in order for the controller to manage the interaction.

The implementation of the centralised controller can be completed in a number of different ways. The following key implementation points must be decided.

- **What undesirable agent interaction the controller will manage.** The particular interaction chosen will influence the implementation of the remaining points.

- **How the controller is invoked.** Whether the agents notify the controller that they believe they are participating in an undesirable interaction or whether the controller watches the agents while they act.

- **How the controller identifies an undesirable interaction.** Being the mechanism used by the controller to identify whether an undesirable agent interaction is really taking place. As the agents have only a local perception of their environment, they may invoke the controller mistakenly believing that they are participating in an undesirable interaction, when in fact the situation will resolve itself without the intervention of the controller.

- **How the controller resolves an undesirable interaction.** Being the mechanism used by the controller to decide how to instruct the agents.

- **How the agents respond to the controller.** Being the mechanism by which the agents receive instructions from the controller and act upon them.

- **How the agents are released from the controller.** At what point the controller stops issuing instructions to the agents and when the agents can determine their own actions.

To allow the agents and the controller to interact, a communication mechanism must be implemented. This communication mechanism will allow agents to invoke the controller and will allow the controller to issue instructions to the agents. A *standardised message passing interface* will be used by all agents, including the controller. Simple text-based messages can be sent to agents using this interface by using a reference or ID number to identify the recipient.

We assume that the agents maintain a reference to the controller. Using this reference, an agent is able to send a message to the controller, which is used as a signal that the agent wishes to invoke the controller. The content of the message to invoke a controller will contain the agent’s ID number, which is used to allow the controller to identify the agent. Using an agent’s ID number, the controller is able to issue instruction to an agent via the message passing interface. The content of these messages will be one or more tokens that can be understood by the agent as actions for the agent to perform. Therefore, the centralised controller uses the
same token system as described for behaviours, where tokens are related to action methods and behaviours by the dictionary of tokens stored in the agent’s internal memory.

In the following examples, the implementation of two different centralised controllers are presented. To resolve the repeated state interactions, described during the demonstration of the obstacle avoidance behaviour (Section 4.3.2), the repeated state controller is developed. To resolve the stationary interactions where agents mutually prevent each other from moving, common with all the behaviours described so far (Figure 4.1, Figure 4.3, Figure 4.6, Figure 4.22), the stationary controller is developed.

5.2 Example: Using a controller to manage repeated state interactions

To demonstrate the implementation and use of a controller designed using our centralised control mechanism, we use the concrete example of a controller to resolve repeated state interactions (Section 4.3.2). In this experiment, it is not our intention to develop or make claims for the resolution mechanism used to resolve repeated state interactions. Instead, we focus on the integration of the controller within the multi-agent system.

5.2.1 Implementing the repeated state controller

We now describe the implementation of the controller and the adaptations required for the Sinatra agents to interact with the controller. The implementation described will contain features that will allow agents to interact with any controller, as well as details that are specific to the repeated state controller. This discussion follows the implementation points highlighted above (Section 5.1).

How the repeated state controller is invoked

For the implementation of the repeated state controller – in this example – the agents themselves will monitor their own actions, rather than have the controller monitor all the agents. From an individual agent’s point of view, a repeated state interaction involves the agent alternating between the same two grid locations.

An agent cannot determine the global reason for its actions as it is only responding to the local situation that it perceives. In the case of the repeated state interaction, the agent is alternating between different behaviours but always finds that the path to its goal is blocked by an obstruction (in this case, another agent). Therefore, in order for an agent to identify a repeated state interaction, the agent must remember its previous grid locations.
We are supposing that the agents can perceive their location \((x, y)\) within the grid environment. To remember its previous grid locations, the agent must be extended with sufficient memory capacity to include location history information. The minimum information required for an agent to know that it has been alternating between the same two grid locations is a history of four locations. Therefore, this is taken as the history length that will be stored by the agents in this example.

Using this location history, an agent is able to identify when it has alternated between the same two grid locations. This is the condition used by the agents to determine whether they need to invoke the repeated state controller. The directive for an agent to invoke the controller can be written as

\[
\text{notifyrep} \leftarrow \text{repeating}
\]

where \text{repeating} is a token representing that the agent has been alternating between the same two states and \text{notifyrep} is a token representing the action of the agent sending a message to the repeated state controller.

As already stated, the agents follow different behaviours when participating in a repeated state interaction. In the case of the example repeated state interaction (Section 4.3.2), the agents use both the simple goal-directed navigation and the obstacle avoidance behaviour during the interaction. This directive can be added to either or both of these behaviours in order for the agent to invoke the controller. However, it is simpler to add the directive to the agent’s master behaviour-switch.

Listing 5.1 shows the modifications (highlighted in bold) to the agent’s behaviour-switch to allow the agent to invoke the repeated state controller. The new directive does not have to be given highest priority, in fact it can have any priority amongst the existing directives, because the action \text{notifyrep} is carried out concurrently with the execution of any behaviour also prescribed by the behaviour-switch.

When the agent invokes the repeated state controller, the agent will send a message to the controller containing the information that the agent used when trying to determine whether it is participating in a repeated state interaction. In this experiment, the agent will send the coordinates of the two grid locations it has been alternating between. The agent also sends its agent ID number, which allows the controller to identify and to send instructions to this agent.

**How the controller identifies a repeated state interaction**

The agents that invoke the repeated state controller may have misidentified a repeated state interaction. It is possible for an agent to observe that it has been alternating between the same two grid locations without a repeated state interaction taking place. Therefore, when the repeated state controller is invoked, it must determine whether a repeated state interaction is really taking place and the agents that are involved in the interaction.
Listing 5.1: The behaviour-switch input file to invoke the repeated state controller.

From the global point of view of the controller, a repeated state interaction involves multiple agents that alternate between grid locations, forming a line of agents within the grid that move together between these locations. The controller detects a repeated state interaction when an agent alternates between grid locations that are next to those that are being alternated between by another agent. Repeated state interactions require a minimum of two agents to occur but can involve more.

The repeated state controller acts in the same time step as it was invoked by the agents. The controller waits until all the agents have finished acting and so the controller has received all of the invocations that it will receive for that time step. To identify a repeated state interaction, the controller attempts to build up sets of agents that are alternating between locations adjacent to another agent in the set. Each set corresponds to a particular repeated state interaction. If a set of agents are participating in a repeated state interaction, then they will all have sent messages to invoke the controller during this time step.

The controller compares the locations that the agents send when they invoke the controller. If the controller finds a set of agents that are alternating between locations that are adjacent to another agent in the set, then the controller has found a repeated state interaction. The controller will issue instructions to agents in this set to try to resolve the interaction. If the controller finds an agent that believes it is participating in a repeated state interaction but is not next to any other agents that have invoked the controller, then the controller has not found a repeated state interaction. The controller will issue no instructions to this agent.

In a suitably populated multi-agent system, there is the potential that multiple
repeated state interactions may be occurring at the same time. For the controller, a repeated state interaction includes all of the adjacent agents who are alternating between grid locations, forming a line of agents within the grid that move together between these locations. Two separate lines of agents moving between grid locations in different parts of the grid will form two distinct repeated state interactions. When the controller identifies sets of agents participating in a repeated state interaction, the requirement that each agent is adjacent to another agent in the set ensures that the controller is able to identify multiple repeated state interactions correctly. In this way, the controller can determine how many repeated state interactions are occurring and can separate the agents involved into different sets.

**How the controller resolves a repeated state interaction**

The repeated state controller uses a simple heuristic to manage the interaction. For a set of obstructing agents, the repeated state controller finds the key agent or agents within the set. The key agents are the agents that obstruct the most other agents. In the case of the repeated state interaction, the key agents are found by using the direction of each agent’s goal to identify the adjacent agent that is blocking this direction. The controller records how many other agents each agent obstructs in this way, in order to identify the key agents.

Having identified the key agents, the controller finds the agent that is obstructing each key agent and instructs this agent to perform the wait action for a number of time steps. The idea is that if a key agent is released from the repeated state interaction by now being able to move towards its goal, then the greatest number of other agents will also be released from the interaction. This will either resolve the repeated state interaction or the number of agents involved will be reduced and the controller can attempt to resolve the interaction fully when invoked again.

This simple heuristic will fail when there are only two agents involved in the repeated state interaction, because both agents will be instructed to wait. The repeated state controller can check for this situation because it knows the size of the set of obstructing agents. If there are only two agents in the set of obstructing agents, then the controller randomly selects one of the two agents to instruct to wait.

The controller issues multiple wait action instructions to the selected agent or agents. This is because, originally, when designing the repeated state controller, the agents were implemented to invoke the controller by adding a directive to the simple goal-directed navigation and the obstacle avoidance behaviour, rather than to the agent’s behaviour-switch. When agents invoke a controller, it is possible that the agents that are following the instructions of the controller are unable to invoke the controller again. By following the instructions of the controller, the agent may no longer perceive that it is participating in the undesirable interaction, or may
no longer execute the behaviour that originally directed the agent to invoke the controller. This means that the controller would not be able to identify the same undesirable interaction in a future time step. Therefore, the controller must issue all the necessary instructions to resolve the interaction in one go.

In this example, however, it would be possible for the agents to invoke the repeated state controller successfully, even while one or more of the agents is following instructions received from the controller. This is due to the combination of the method used by the agents to identify when a repeated state interactions is occurring, the directive to invoke the controller being in the agent’s master behaviour-switch where it can always be executed and the method used by the controller to resolve the interaction. Therefore, the ability to invoke the controller multiple times for the same interaction is a coincidence of the implementation of the repeated state controller and is not a feature of the controller implementation in general.

We artificially impose this constraint on the behaviour-switch for the repeated state controller and describe how Sinatra agents can be implemented to follow a series of instructions over multiple time steps. In Section 5.4.3 we discuss what happens when a controller does not issue all the instructions to resolve an interaction at once and how a controller can be invoked multiple times to resolve the same interaction.

The agents will execute one instruction message from the controller per time step. As the repeated state interaction requires the key agent to stop obstructing the path of other agents in order to be resolved, the agent that the controller instructs to wait must wait until the key agent has moved out of the way. In this example, using the simple goal-directed navigation and the obstacle avoidance behaviour to move the agents, it is sufficient for the controller to instruct the agent to wait for three time steps.

How the agents respond to the repeated state controller

In order for the controller to instruct the agents, the agents are extended to include sufficient memory to record a set of actions issued by the controller. The controller issues instructions to the agents by sending messages containing the required actions.

For the implementation of the repeated state controller, the controller will send the agent a number of messages equal to the number of time steps where the controller will instruct the agent to act. Each message contains the action or actions for the agent to perform during a single time step. In this example, the controller will send three messages to the agent, each containing the \texttt{wait} token. In this way, the controller is able to instruct the agent how to act for three time steps.

The agent records the tokens contained within the messages received from the controller in the order that the messages are received. The first message received from the controller is used to instruct the agent how to act in the next time step. If a message from the controller contains multiple tokens, then the message instructs the
agent to execute multiple concurrent actions during a single time step. Therefore, the agents store the messages received from a controller as a two dimensional record of tokens.

This structure allows the agent to record the series of actions to be performed over multiple time steps, as well as time steps where multiple actions must be executed concurrently. For each time step after the agent has received instructions from the controller, the agent will execute (or attempt to execute) all the actions specified for that time step by consulting this record of instructions.

At the start of the next time step after invoking the controller, an agent that has received instructions from the controller will now perform the action or actions specified by the first message it received. In order for the agents to respond to the instructions of the controller, a new directive must be added to the agent’s master behaviour-switch.

\[
\text{controller} \leftarrow \text{instructions}
\]

The \textit{instructions} token specifies that the agent has tokens received from a controller and recorded in its memory. The agent may not have received these instructions in the previous time step of the simulation. Nevertheless, the agent has instructions stored in its memory that it has yet to execute. The behaviour-switch, therefore, directs the agent to follow the actions specified by the controller, signified by the \texttt{controller} token.

The updated master behaviour-switch, to allow agents to invoke and to follow the instructions (highlighted in bold) of the repeated state controller, is shown in Listing 5.2. Constraint formulae are used to prevent the agent from following another behaviour while following the instructions of the controller, as well as to prevent the agent invoking the controller again.

The \texttt{actionSwitch} method must be modified to allow the agent to follow the instructions of the controller. Having identified the \texttt{controller} token returned by the \texttt{lookUpAction} method, the \texttt{actionSwitch} method must identify the token or tokens stored in the agent’s memory to be executed in this time step. The \texttt{actionSwitch} method calls itself for each token identified in this way, allowing the appropriate action method to be executed. This allows the agent to act as instructed by the controller.

Having identified the tokens prescribed for the current time step, the \texttt{actionSwitch} method must also remove these tokens from the agent’s memory. This allows the agent to identify the tokens that should be executed for each time step. For each successive time step, the tokens to be executed will be the first set of tokens found in the agent’s memory, as the messages received from the controller are stored in the order that they were received.
controller :- instructions.
notifyrep :- repeating.

newgoal :- atgoal.
obstacle :- obstructed.
goaldirected :- .

:- newgoal, obstacle.
:- goaldirected, obstacle.
:- newgoal, goaldirected.
:- notifyrep, controller.
:- goaldirected, controller.
:- obstacle, controller.

s: 0 \{obstructed, atgoal\} 2.
s: 0 \{repeating\} 1.
s: 0 \{instructions\} 1.

in: obstructed atgoal repeating instructions.

out: obstacle goaldirected newgoal notifyrep controller.

Listing 5.2: The behaviour-switch input file to follow the instructions of the repeated state controller.

How the agents are released from the repeated state controller

The repeated state controller only issues one set of instructions to the agents. The agent’s behaviour-switch instructs the agent to follow these instructions during the following time steps. As described above, the actionSwitch method removes instructions for the current time step from the agent’s memory after they have been identified. When the agent has finished following the instructions of the controller, there will be no more tokens stored in the agent’s memory. Therefore, the behaviour-switch rule to follow the instructions of the controller will no longer trigger. This allows the agent to return to determining its own actions autonomously, as the behaviour-switch will now direct the agent to follow a behaviour.

5.2.2 Demonstration

Figure 5.1 to Figure 5.9 show the repeated state controller resolving a repeated state interaction. Figure 5.1 to Figure 5.5 show the agents moving between the two repeated grid locations, at which point in Figure 5.6 the repeated state controller is called to resolve the undesirable agent interaction. As the controller finds only two agents in the set of obstructing agents, the red agent is randomly selected to be instructed to wait, so that the blue agent can be released.
Figure 5.1: Two agents participating in a repeated state interaction.

Figure 5.2: The agents alternate between the two grid locations. At this point each agent’s location history can be thought of as the set of grid locations \{X, Y, P1, P2\}, where X and Y are the previous agent locations from before the repeated state interaction started.

Figure 5.3: Each agent’s location history will now become \{Y, P1, P2, P1\}. 

167
Figure 5.4: The agents are instructed by the behaviour-switch to now follow the **obstacle avoidance** behaviour.

Figure 5.5: At this point each agent’s location history will become \{P1, P2, P1, P2\}. The two agents notify the repeated state controller that they believe they are participating in a repeated state interaction.

Figure 5.6: The repeated state controller identifies that a repeated state interaction is occurring and the agents that are involved. The controller instructs the red agent to wait for 3 time steps. The blue agent receives no instructions from the controller and so follows the **simple goal-directed navigation**.
Figure 5.7: The repeated state controller issued multiple wait instructions to the red agent, as the behaviours take multiple time steps to move the blue agent.

Figure 5.8: The red agent follows the final instruction issued by the controller, while the blue agent is released from the repeated state interaction.

Figure 5.9: The two agents are now both directed by the behaviour-switch to continue using the simple goal-directed navigation to reach their goals. Therefore, the red agent can again autonomously determine its own actions.
5.2.3 Summary

Using the example of a repeated state controller, we have demonstrated that our centralised control mechanism can be used to resolve repeated state interactions, which are not successfully handled by the existing behaviours. The repeated state controller uses a simple heuristic method to resolve the interaction. The repeated state controller does not send instructions to agents that directly resolve the repeated state interaction; instead, the controller sends instructions that allow the other agents to resolve the interaction themselves.

We have demonstrated that the repeated state controller is able to identify a repeated state interaction correctly. During observations of agents that are capable of invoking the repeated state controller, agents were seen invoking the controller when there was no repeated state interaction taking place. In these situations, the controller correctly identified that the agent was not participating in a repeated state interaction and issued no instructions to the agent. Similarly, the controller was never observed to identify a repeated state interaction mistakenly when one was not taking place.

Figure 5.1 to Figure 5.9 demonstrate that, despite the simple heuristic approach used to resolve the interactions, the repeated state controller is able to resolve repeated state interactions successfully. The controller has also been observed resolving repeated state interactions involving three agents. Repeated state interactions involving four or more agents are very uncommon in Sinatra, as they require all of the agents involved to have their goal locations in the same row or column of the grid.

When there are more than two agents participating in the repeated state interaction, it was observed that the controller may only reduce the number of agents participating in the interaction, rather than resolving the interaction completely. The repeated state controller was invoked again within a few time steps and was then able to resolve the smaller interaction. This shows that, in situations where the controller is unable to resolve a repeated state interaction fully, the repeated state controller is able to reduce the number of participating agents, making it possible for the interaction to be resolved later.

Having invoked the controller, when the agent received no instruction from the controller, for whatever reason, the agent was observed to follow its next expected behaviour. When an agent invoked the controller and received instructions for how it should act, the agent was observed to follow these instructions for the appropriate number of time steps before switching to following a behaviour. Therefore, the behaviour-switch allows the agents to follow the instructions of the controller and to regain their autonomy after the controller has been invoked. This allows the repeated state controller to control the agents for only the length of time needed to
manage the interaction.

5.3 Example: Using a controller to manage stationary interactions

In this example we demonstrate the development of a controller to resolve stationary interactions, where two or more agents mutually prevent each other from being able to move. Stationary interactions have been observed occurring with all behaviours that have been described so far (Figure 4.1, Figure 4.3, Figure 4.6, Figure 4.22).

The aim of this example is to illustrate the implementation of a controller to resolve a different type of undesirable interaction in order to highlight the modular nature of the centralised control mechanism. For this reason, much of the implementation details are the same as those described when implementing the repeated state controller. Again, it is not our intention to develop or make claims for the resolution mechanism used to resolve stationary interactions.

5.3.1 Implementing the stationary controller

We now describe the implementation of the controller, making note of the agent based implementation features that are the same for the stationary controller and the repeated state controller. These features are those that allow agents to interact with any controller based on the centralised control mechanism.

How the stationary controller is invoked

Similarly to the repeated state controller, the agents themselves will monitor their own actions to determine whether they are participating in a stationary interaction. From an individual agent’s point of view, a stationary interaction occurs when the agent is directed to move forwards, towards its goal, but is prevented from doing so because the grid location is already occupied by another agent.

For the traffic law behaviour, we required that the sensory capabilities of the Sinatra agents were extended to allow an agent to perceive if an obstruction is another agent (Section 4.3.3). For the stationary controller, we now also require that the sensory capabilities of the agents are extended to allow an agent to identify the ID number of an obstructing agent.

Using these perceptions, the agent is able to identify when the path to its goal is being obstructed by another agent. This is the condition used by the agent to determine whether it needs to invoke the stationary controller.

The directive for the agent to invoke the controller can be written as

\[ \text{notifystat} \leftarrow \text{obstructedagent} \]
The **obstructedagent** token represents that the agent is preventing from moving towards its goal by another agent and the **notifystat** token represents the action of the agent sending a message to invoke the stationary controller.

A stationary interaction can occur when an agent uses any behaviour but was a particular problem for the **simple goal-directed navigation**. Unlike a repeated state interaction, however, a stationary interaction does not involve the agent switching between different behaviours. During a stationary interaction the agent is unable to move and so the current state of the agent does not change. Therefore, the agent’s behaviour-switch will direct the agent to keep following the same behaviour.

The directive for the agent to notify the controller that it is participating in a stationary interaction can be added to the agent’s master behaviour-switch. Instead, if the directive is to be added to the agent’s other behaviours, then it must be added to all the existing behaviours for the controller to be invoked properly.

Listing 5.3 shows the modifications (highlighted in bold) to the agent’s behaviour-switch to allow the agent to invoke the stationary controller. In this behaviour-switch, we assume that the agent is only able to follow the **simple goal-directed navigation**.

```
notifystat :- obstructedagent.

newgoal :- atgoal.
goaldirected :- .

:- newgoal, goaldirected.

s: 0 {atgoal} 1.
s: 0 {obstructedagent} 1.
in: atgoal obstructedagent.

out: goaldirected newgoal notifystat.
```

Listing 5.3: The behaviour-switch input file to invoke the stationary controller.

The new directive does not have to be given highest priority because the action notifystat is carried out in conjunction with the execution of any behaviours also prescribed by the behaviour-switch. In the case of a stationary interaction, this behaviour will again prescribe an attempt to move forwards, which the agent will be unable to perform.

When the agent invokes the stationary controller, the agent will send a message to the controller containing the information that the agent used when trying to determine whether it is participating in a stationary interaction. In this example, the agent will send the controller its agent ID number and the ID number of the obstruct-
ing agent. The agents invoke the stationary controller using the same mechanism that the agents used to invoke the repeated state controller. The standardised message passing interface allows the agents to invoke different controllers by specifying the recipient controller using a reference stored by the agent.

How the controller identifies a stationary interaction

The agents that invoke the stationary controller may have misidentified a stationary interaction. A stationary interaction involves agents that \textit{mutually} prevent each other from moving. It is possible for an agent to observe that it has been prevented from moving by another agent without a stationary interaction taking place. For example, this can occur when an agent is obstructed by another agent that is currently changing direction. As turning $90^\circ$ takes a whole time step, an agent may be obstructed for multiple time steps until the turning agent has moved on. Therefore, when the stationary controller is invoked, it must determine whether a stationary interaction is really taking place and the participating agents.

From the global point of view of the controller, a stationary interaction involves multiple agents that are mutually preventing each other from progressing towards their goals. The controller perceives a stationary interaction when an agent reports that it is obstructed by an agent that also reports that it is being obstructed. Stationary interactions require a minimum of two agents to occur but can involve more agents, where together the agents mutually prevent each other from moving.

Similarly to the repeated state controller, the stationary controller acts in the same time step as it was invoked by the agents, after all of the agents have finished acting. To identify a stationary interaction, the controller attempts to build up sets of agents that are mutually obstructing each other. This is done by comparing the ID numbers of agents that have invoked the controller to report that they are being obstructed and the agents that they report as obstructing them. If a set of agents are mutually obstructing each other, then they will have all sent a message to invoke the controller during this time step.

If the controller finds a set of agents that are obstructing each other, then the controller has found a stationary interaction. The controller will issue instructions to agents in this set to try to resolve the interaction. If the controller finds an agent that believes it is participating in a stationary interaction but the agent reported as obstructing this agent has not invoked the controller, then the controller has not found a stationary interaction. The controller will issue no instructions to this agent.

In a suitably populated multi-agent system, there is the potential that multiple stationary interactions may be occurring at the same time. Two separate sets of agents, where no agent is obstructed by an agent in the other set, will form two distinct stationary interactions. Similar to the repeated state controller, the requirement that the agents in a stationary interaction are obstructing and obstructed by
another agent in the set ensures that the controller is able to identify multiple sta-
tionary interactions successfully. This allows the stationary controller to determine
how many stationary interactions are occurring and to separate the agents involved
into different sets.

**How the controller resolves a stationary interaction**

The controller uses a simple heuristic to manage the interaction, in the same manner
as the repeated state controller. For a set of mutually obstructing agents, the sta-
tionary controller finds the key agent or agents within the set, being the agents that
obstruct the most other agents. As each agent invokes the controller by sending the ID number of the agent that is obstructing it, the stationary controller can easily
identify the key agents within a set of agents.

The idea for resolving the stationary interaction is to allow the key agents to be
released from the interaction. This will allow the greatest number of other agents
to also be released from the stationary interaction. The controller releases the key
agents by sending instructions to the agents that were reported as obstructing the
key agents.

Having identified an agent to send instructions to, the stationary controller searches
the four grid locations that this agent can move to from its current location in order
to find an empty grid location. The controller will send appropriate instructions to
turn the agent towards this empty location and then for the agent to move forwards
into this space. Therefore, the controller will send one or two turn left or turn right
tokens to the agent, followed by a move forward token.

This simple heuristic has some unintended behaviour in the case where there are
only two agents involved in the stationary interaction. If both agents are directed
to move to grid locations that are still next to each other, then the stationary
interaction will not be resolved. For example, both agents may be instructed to
move to the empty grid location immediately north of their current location. Having
completed executing the instructions of the controller, the agents are likely to again
find that they are mutually obstructing each other from reaching their goals and
will invoke the stationary controller again. To avoid this possible situation, in the
special case where a stationary interaction involves only two agents, the stationary
controller will randomly select to send instructions to one of the two agents.

The controller must issue all of the instructions to the agents during the time
step that it is invoked, because the controller is only invoked once for the stationary
interaction. Once an agent starts to follow the instructions received from the con-
troller, this agent will no longer perceive that it is being obstructed by another agent.
Therefore, the agents involved in the interaction will not all invoke the controller
once some of them start following the controller’s instructions. This will prevent the
controller from being able to identify the stationary interaction again.
The agents will execute one instruction message from the controller per time step. The number of messages the stationary controller sends to the agents may vary depending on the number of turn left or turn right actions required to move an agent into the empty grid location. Therefore, if there are multiple key agents, one key agent may be released from the stationary interaction before another.

**How the agents respond to the stationary controller**

The agents respond to the instructions received from the stationary controller using the same mechanism that was described for the implementation of the repeated state controller. The controller issues instructions to the agents by sending messages containing tokens. The agents parse these messages and store the tokens in the order that the messages were received. Again, the controller uses a standardised message passing interface within our centralised control mechanism, in this case allowing the agents to receive instructions from the controller.

The agents store the instructions received from the controller in their internal memory. The updated master behaviour-switch to allow the agents to invoke and to follow the instructions (highlighted in bold) of the stationary controller is shown in Listing 5.4.

```prolog
controller :- instructions.
notifystat :- obstructedagent.
newgoal :- atgoal.
goaldirected :- .

:- newgoal, goaldirected.
:- notifystat, controller.
:- goaldirected, controller.

s: 0 \{atgoal\} 1.
s: 0 \{obstructedagent\} 1.
s: 0 \{instructions\} 1.

in: atgoal obstructedagent instructions.
out: goaldirected newgoal notifystat controller.
```

Listing 5.4: The behaviour-switch input file to follow the instructions of the stationary controller.

The master behaviour-switch for the stationary controller is modified in the same manner as described for the behaviour-switch for the repeated state controller. Using this behaviour-switch, the modified actionSwitch method will allow the agent to
act as instructed by the stationary controller.

**How the agents are released from the stationary controller**

The agent is released from the stationary controller in the same manner as an agent is released from the repeated state controller. When the agent has identified the token or tokens that is has received from the controller for this time step, the `actionSwitch` method deletes these tokens from the agent’s memory of received instructions. When the agent has finished executing all of the instructions from the controller, there will be no more instructions stored in the agent’s memory. The behaviour-switch will then direct the agent to follow a behaviour.

### 5.3.2 Demonstration

Figure 5.10 to Figure 5.13 show the stationary controller resolving a stationary interaction. As the controller finds only two agents in the set of obstructing agents, the red agent is randomly selected to receive instructions from the controller, in order to release the blue agent.

![Figure 5.10: Two agents participating in a stationary interaction. The agents notify the stationary controller that they believe they are participating in a stationary interaction.](image)

176
5.3.3 Summary

Despite the simple heuristic method used to resolve the stationary interactions, the stationary controller was able to resolve stationary interactions when they occurred.
We have demonstrated the stationary controller resolving a stationary interaction involving two agents. Observation of the stationary controller also showed that it was able to resolve stationary interactions involving three and four agents.

The stationary controller was observed to be able to handle situations where agents invoke the controller mistakingly and to identify a stationary interaction correctly when one was occurring. In addition, the behaviour-switch correctly allows the agents to follow the instructions of the controller and to regain their autonomy after the controller has been invoked. Therefore, the stationary controller was observed to function correctly.

The main aim of the stationary controller example, however, was to demonstrate the implementation of an alternative controller using our centralised control mechanism. In the next section, we compare the implementation of the stationary controller against the repeated state controller.

5.4 Conclusion

In this section, we describe the key implementation points for the centralised control mechanism, from the perspective of the agent and the controller. We begin by describing the modifications required to allow agents to perceive when to invoke the example controllers developed in this chapter and to allow the agents to follow instructions received from these controller. We highlight the impact of these modifications on the agents that are implemented in Sinatra. We then compare the implementation of the stationary controller and the repeated state controller in order to highlight the common aspects of their implementations that are due to the centralised control mechanism. We finish by discussing some general issues related to the use of our centralised control mechanism.

5.4.1 Agents using the centralised control mechanism

The modification of the behaviours to allow the agents to invoke the example controllers required only minimal changes. For each controller, a single new directive is added to the master behaviour-switch, or any other behaviour that is being used to invoke the controller. The relative priority of this directive compared to the existing directives is unimportant in these examples, because the act of sending a message to the controller is carried out concurrently with the execution of any behaviour or action also prescribed by the behaviour.

By implementing the agents to invoke the example controllers, we have been able to demonstrate Sinatra agents with behaviours that guide them to perform two actions within a single time step. In these examples, the two actions are to invoke a controller and to execute another behaviour, which in turn directs the agent to
execute one of the agent’s existing movement actions (turn left, turn right or move forwards).

Having been invoked, the repeated state controller and the stationary controller both use a simple heuristic method to resolve their undesirable interaction. However, the modular design of the controller means that, if desired, this simple approach can be replaced with a more sophisticated resolution mechanism. From the point of view of the individual agents, the mechanisms used to interact with the controller would remain unchanged and the agents would still respond to the instructions received.

In order to follow the instructions of the controller, the agents are required to maintain a record of instructions that have been received. The agent may also be required to store other information such as a record of its previous locations, as used to invoke the repeated state controller. This means that the minimum requirements for an agent must now include this additional memory capacity in order for a controller to be used. This is a relatively large increase on the previous memory requirements for an agent, although the information itself may be stored in just a few bytes.

The addition of directives to invoke and to follow the instructions of a controller both lead to double the number of state-action table entries in the modified behaviours. This means that the state-action table for a behaviour may quickly become very large if an agent is implemented that can invoke a variety of different controllers. The exponential growth of entries in a state-action table is a factor the system designer must be aware of when implementing new controllers.

5.4.2 Implementing the centralised control mechanism

The stationary controller has many implementation features that are similar to the implementation of the repeated state controller. These example controllers allow the modular nature of the centralised control mechanism to be highlighted. Many of these similarities are due to the standardised nature of the centralised control mechanism implementation, particularly in terms of how the agents and a controller communicate.

The example controllers developed in this chapter demonstrate the design of a controller as an autonomous agent or entity within the multi-agent system, possessing the ability to construct a global view of an agent interaction based on the information it receives. A controller does not have access to information that is not available to the other agents; instead, the controller has the ability to combine the local perceptions of the individual agents in order to develop a global perception of what is occurring in the multi-agent system. Using this global view, the controller is able to understand when an undesirable interaction is occurring, to identify the agents involved and to instruct the agents how to act.

Some of the similarities between the example controller implementations, however,
are not due to the centralised control mechanism. These similarities are due to properties of the undesirable interactions that are being resolved in these examples or assumptions that have been made by both controllers about how to resolve the interactions.

Based on the information sent by the agents when they invoke the controller, the controller is able to determine whether a particular undesirable interactions is occurring. Similarities between a repeated state interaction and a stationary interaction allow both example controllers to identify their undesirable agent interaction by building up a set of agents that mutually obstruct each other. Other techniques for identifying undesirable interactions can be used when implementing a controller. For example, a machine learning algorithm could be used to train a controller to recognise the type of undesirable interaction it will handle.

Both example controllers use a simple heuristic method for identifying the agent or agents to instruct. Again, this similarity is due to the undesirable interactions that the two controllers are handling, rather than being a feature of the centralised control mechanism. Identifying key agents that obstruct the most other agents and then allowing these key agents to move away from the undesirable interaction is a possible method for resolving both interactions. Different strategies or tools can be used to implement the resolution mechanism for an undesirable interaction. For example, in Chapter 6 we use the universal multi-agent planning framework UMOP to implement the resolution mechanism of a controller.

The resolution mechanisms of the example controllers only send instructions to some of the agents participating in the undesirable interactions. While this has been shown to work in these examples, these resolution mechanisms are based on a potentially critical assumption. The resolution mechanisms rely on the other agents involved in the interactions following the behaviours expected by the controllers. Agents that deviate from this expected pattern of behaviour may cause the controllers to be unable to resolve their undesirable interactions successfully.

In particular, the success of the repeated state controller relies entirely on the independent behaviour of the other agents falling within the parameters expected by the controller. By only instructing an agent to wait, the repeated state controller relies on the behaviours used by the other agents to resolve the repeated state interaction without further direction from the controller. Therefore, the use of such a simple resolution mechanism must be considered carefully by the system designer to ensure that the assumptions made by the controller are valid.

Having identified an agent or agents to issue instructions to, however, by whatever means used to implement the controller, the controller sends a series of messages to the agents. These messages contain tokens that represent actions for the agents to carry out. A standardised message passing interface, consisting of simple text-based messages, is used by the controllers to send these instructions to the agents, as
well as to allow the agents to invoke the controllers. Therefore, this communication mechanism of the centralised controller implementation allows an agent to interact with any controller and to understand the instructions received, regardless of the specifics of the controller implementation.

5.4.3 Further questions for the centralised control mechanism

We now explore some general issues related to the use of our centralised control mechanism, including different implementation strategies and what happens when there are multiple controllers.

What changes when agents or the controller monitor actions?

We have described the modifications required to the Sinatra agent implementation to allow the agents to invoke the controller and to follow the instructions received. While most of these changes required only simple modifications to the existing behaviours and methods, the agents require improved memory and sensory capabilities in order to determine when to invoke the controllers. This is because both controllers were implemented to respond to messages from the agents, rather than to monitor the actions of the agents themselves.

In some situations, however, it is desirable for a controller to actively monitor the agent’s actions. Such a controller may be appropriate when handling undesirable interactions that occur in a specific location.

For example, imagine a corner controller, which monitors the corner areas of the grid environment and sends instructions to agents passing through these areas. The corner controller will aim to manage undesirable agent interactions caused by agents obstructing each other and being restricted in their movements due to their proximity to the edge of the grid.

To implement the corner controller, the controller will have to be able to observe agents in a particular area of the grid environment. For example, the controller could monitor all agent action within the 3x3 square of grid cells that includes the corner cell. The controller must be able to perceive the location and orientation of all agents within this area. This could be implemented by allowing the controller to observe this part of the grid or by implementing the agents to send information to the controller automatically when they are in this monitoring area. Although these messages can use the same message passing mechanism already used by the agents, this is different to the agents monitoring their own actions. In this situation, the agents would be sending the information to the controller without any processing of this information being carried out by the agent.

Using the information gathered by monitoring the agents, the controller will attempt to manage undesirable interactions within the corner area, being situations
where the agents obstruct each other. Different techniques for managing the undesirable interactions could be used. The corner controller could be implemented to try to predict when undesirable interactions will occur and so prevent them from happening by issuing instructions to the agents before the situation arises. This could be achieved by using information about the goal locations and intended paths of the agents to predict when agents might interfere with each other’s progress. Alternatively, the controller could be implemented to watch for when an undesirable interaction is occurring and to use its knowledge of the restricted movement in the corner area to allow the interaction to be resolved successfully.

There is a range of possible controller implementations that could be developed and tested, depending on the desired balance between direction from the controller and autonomous behaviour of the agents. The controller may be implemented to issue the minimum number of instructions in order to attempt to allow the agents to handle the interactions themselves, or the controller may be designed to issue instructions to all agents whenever they enter a specific area, or some method in between.

**What happens when an agent receives instructions from a controller without first invoking the controller?**

If the controller monitors the actions of the agents, then when the controller observes an undesirable interaction, the controller will issue instructions to agents that have not themselves invoked the controller. This situation can also occur for other reasons; for example, a controller may need to instruct more agents to act than just those that invoke the controller. This latter situation is discussed in relation to the UMOP controller in the next chapter (Section 6.3).

From the point of view of any agent that did not invoke a controller, the agent unexpectedly receives instructions from the controller. Therefore, an agent that was otherwise autonomously determining its own actions must now respond to unsolicited instructions from a controller.

For this question, we assume that the agents will obey the instructions of the controller. The issue of when agents intentionally disobey the controller will be discussed later.

When an agent invokes a controller, the agent will not necessarily receive instructions from the controller. Therefore, having invoked a controller, the agent continues to determine its actions autonomously until it receives instructions. In this way, if the agent misidentifies an undesirable interaction, the agent is not left waiting for instructions from the controller that it will not receive.

Therefore, an agent that invokes a controller does not use the fact that it has invoked the controller when determining how it should act. The agent maintains no record that it has invoked a controller and does not expect to receive instructions.
Instead, the agent responds when it does receive instructions from the controller. In situations where the agent has not itself invoked a controller, this same process can still be used by the agent to respond to instructions that it has received.

A controller issues instructions to an agent by sending messages containing tokens, representing the required actions for the agent to perform. The agent parses the tokens from these messages and saves the tokens in its internal memory. This process happens whenever the agent receives instructions from a controller and so is not affected by whether the agent invoked the controller first or whether the messages received are unexpected.

When an agent has instructions stored in its internal memory, the master behaviour-switch directs the agent to follow the token or tokens specified by the controller for the next time step. Therefore, there is no change required in the implementation of the agent to follow instructions from a controller that the agent has not invoked. The controller is able to issue instructions in the form of messages as normal and the agents are able to act based on these instructions.

**What happens if a controller does not issue all the instructions to the agent at once?**

In the example implementations of the repeated state controller and the stationary controller, the controllers were required to issue all of the instructions for resolving the undesirable interaction at once. This method meant that when the agent had finished executing the instructions from the controller then the agent was free to return to autonomously determining its own actions. In the case of the stationary controller, this method was required because the agents are unable to identify that they are (or were) participating in a stationary interaction once they start following the instructions of the controller.

As described for the repeated state controller, however, the agents participating in the repeated state interaction would still be able to invoke the controller and the controller would still be able to identify the same repeated state interaction, even after one or more of the agents has received instructions from the controller. This is because the method used by the agents to determine whether they are participating in a repeated state interaction still identifies the interaction after some of the agents have been instructed to wait. Therefore, the repeated state controller can be implemented to issue only one wait instruction to selected agents each time it identifies a repeated state interaction.

It is not a requirement of the centralised control mechanism that the controller must issue all its instructions at once. In particular, there may be undesirable interactions where the controller needs to issue multiple sets of instructions to the agents. This allows the controller to modify the instructions it sends to the agents, depending on the outcome of the previous set of instructions.
The controller can also check that an undesirable interaction has been resolved correctly and issue another set of instructions if, for some reason, the agents have not been able to act as instructed. For the example controllers developed in this chapter, this is achieved by the agents realising they are still participating in an undesirable interaction and invoking the controller a second time. However, a few time steps may be required for the agents to identify that they are still participating in an undesirable interaction.

For situations where the controller monitors the actions of the agents, then the controller can always issue further instructions to the agents. However, when the agents monitor their own actions, then the controller is only able to issue instructions when invoked by an agent.

Therefore, in situations where the controller does not issue all the instructions to resolve an interaction in one go, and cannot guarantee that it will be invoked again by the agents, the controller must have the opportunity to issue a further set of instructions to the agents. This could be achieved by the controller sending a final instruction message to an agent, specifying that the agent should invoke the controller again. This instruction to invoke the controller can also specify updated information required by the controller to determine what instructions, if any, it will now send to the agent.

When the actionSwitch method identifies that the agent must invoke the controller again during this time step, the agent is able to invoke the controller using the same message passing interface that has been described. The agent will invoke the controller again, either sending the same information originally used to invoke the controller, or updated information that the controller has requested.

When a controller is invoked for a second time, the controller will use the information contained in the new message it receives to determine whether the agent is still participating in an undesirable interaction. Therefore, if necessary, the controller can issue a further set of instructions to the agent in order to attempt to resolve the undesirable interaction, based on the updated information from the agent.

This system for invoking a controller again after an initial set of instructions means that an agent is no longer automatically released from following the directions of a controller after it has executed all of the instructions received from the controller. When the agent invokes the controller again, the agent may receive further instructions to perform over the next time steps.

Whenever an agent invokes a controller, however, it is not guaranteed to receive instructions from the controller. Therefore, as we have described above, the agent does not record that it has invoked the controller and instead responds to instructions when they are received. This means that if an agent invokes a controller as instructed but does not receive any new instructions, then the agent will return to autonomously determining its own actions as if the controller had not been invoked.
How can multiple controllers be integrated into the agent design?

Integrating multiple controllers into the agent design follows a similar pattern to the implementation of a single controller, as described in this chapter. Directives telling the agent in what circumstances to invoke each controller must be added to all appropriate behaviours (including behaviour-switches). If a behaviour contains directives to invoke different controllers, then the system designer must indicate in the behaviour whether it is appropriate for the agent to invoke all or some of the controllers at the same time. This is handled within a behaviour by using the constraint formulae to restrict the permitted concurrent actions.

What if an agent receives new instructions before it has finished an existing set of instructions?

An example of an agent receiving new instructions from a controller before it has finished executing an existing set of instructions from the same controller can occur when an agent is participating in a stationary interaction. The agent receives instructions to move away from the stationary interaction and, before it can finish completing these actions, finds itself now participating in a different stationary interaction with a different agent.

In this example situation, for the stationary controller, the agent should ignore both sets of instructions that it has received. The old set of instructions are now ‘out of date’ because of the new interaction with the different agent and the new set of instructions may not take into account the original stationary interaction. Therefore, the agent should ignore the instructions of the stationary controller and return to following its behaviours. Then, either the behaviours will resolve the undesirable interaction themselves, or the stationary controller will be invoked again. If the stationary controller is invoked again, however, it will be able to include the additional agent or agents when it identifies the undesirable interaction.

This process becomes more complicated when an agent receives instructions from two or more different controllers. If an agent is participating in multiple undesirable interactions, or there are different controllers designed to handle the same interaction, the agent must not confuse the instructions received from different controllers. This is especially important as the instructions received may be contradictory.

To handle situations where an agent receives multiple conflicting sets of instructions, the agent requires a ranking of controllers that it prefers to obey (for whatever reason). This strategy can be achieved as long as the agent knows which controller issued each set of instructions. If an agent receives instructions from a more preferred controller while already following the instructions from a less preferred controller, the agent will stop following its current instructions in order to follow the instructions of the preferred controller. Similarly, an agent will ignore instructions received
from a controller if it is already following a set of instructions received from a more preferred controller.

To implement this strategy, the agent needs to record the instructions received from different controllers separately in its memory. This allows the agent to use information on which instructions came from which controllers when it acts. The master behaviour-switch must be modified to allow the agent to select between instructions received from different controllers, based on the ranking of controllers preferred by the agent.

We have described how the directive

\[
\text{controller} \leftarrow \text{instructions}
\]

is used in the master behaviour-switch to allow the agent to follow the instructions of a controller. If multiple such directives are used for the different controllers, then the agent will be able to prioritise the instructions of its more preferred controllers.

For example, consider the following directives and constraint formula.

\[
\text{statcontroller} \leftarrow \text{statinstructions}
\]
\[
\text{repcontroller} \leftarrow \text{repinstructions}
\]
\[
\leftarrow \text{statcontroller, repcontroller}
\]

If these directives are written in descending priority order, then the directives will allow the agent to follow the instructions of the stationary controller over those of the repeated state controller, in a situation where the agent has received instructions from both controllers.

In some situations, it may also be desirable for an agent to be able to select which controllers to follow depending on its current state. Further literals can be added to the condition of the directives above to specify that an agent will only follow the instructions from a controller in certain circumstances.

In addition, if the instructions received from multiple controllers do not conflict, then this method of storing instructions from different controllers separately in memory can be used to allow an agent to follow the instructions of multiple controllers simultaneously. The constraint formulae of the behaviour-switch will specify whether the instructions from different controllers can be obeyed concurrently or not.

**What if an agent tries but fails to execute an instruction from the controller?**

An agent may fail to execute an instruction from the controller because of one-off occurrences, such as the actions of some other agent. Alternatively, an agent may persistently fail to execute an instruction. This can occur when there is a mistake in the implementation of the agent, whether deliberate or accidental.
If an agent tried but failed to be able to perform some or all of the actions specified for a single time step then, using the current implementation, the agent would not notice this error. The actions to be performed by the agent are determined using the `actionSwitch` method, which deletes the instructions for this time step once the actions to be performed have been identified.

If an agent fails to act as directed by the controller, it is possible that the undesirable interaction will not be resolved. This will result in the controller being invoked again. If the agent failed to act because of a one-off occurrence, such as interference from another agent not previously involved in the interaction, then the controller will be able to resolve the undesirable interaction in the second attempt, as long as the agent now acts as directed. If the agent failed to act because of a mistake in its implementation, then the controller may be unable to resolve the interaction by issuing instructions to this agent, depending on the nature of the mistake.

In the case of agents that fail to act due to a one-off occurrence, a possible solution may be to modify the `actionSwitch` method to delete the tokens representing actions to be performed in this time step only after the agent has been able to verify, in some manner, that the actions have been executed successfully. Depending on the actions the controller requires the agent to perform, verifying that they have been executed successfully could require the agents to be extended with additional sensory capabilities.

If the controller monitors the actions of the agents, or is invoked multiple times by the agents, then the controller itself may be able to handle the issue of agents failing to execute instructions. In these situations, there is no requirement for the controller to issue all of its instructions in one go. Therefore, the controller can issue instructions for each time step individually, or for a small number of time steps at once. By monitoring the agent’s executing these actions, or being invoked again by the agents, the controller is able to respond to agents failing to act as instructed. This modification would require no change to the mechanism used by the agents to follow the instructions of the controller.

**What if an agent intentionally disobeys a controller?**

Agents may be programmed to disobey or ignore instructions from a controller in favour of following their existing behaviours during certain situations. For example, an agent may normally follow the instructions of a controller, except when the agent is dangerously low on battery and needs to move to a recharge point. This process can be implemented by adjusting the relative priorities of the directives in the behaviour-switch. If a directive where an agent will follow a particular behaviour or execute a specific action is given higher priority than a directive where the agent follows the instructions from a controller, and it has been specified that both cannot be carried out simultaneously, then the agent will disobey the controller.
Of course, it is possible that if an agent intentionally disobeys a controller then the controller will be unable to resolve the undesirable interaction. This can result in the disobeying agent, as well as other agents in the system, being trapped in an undesirable interaction.

When implementing a controller, therefore, the system designer may need to take into account that the agents receiving instructions from the controller may not follow these instructions. A more robust resolution mechanism is required in these situations, to ensure that the controller is able to resolve the undesirable interaction. For example, the controller may be implemented to send instructions to all of the agents participating in an undesirable interaction, not just some of them.

Similarly, the system designer must be careful when specifying situations when an agent will disobey the instructions of a controller. In the scenario above, even though the agent needs to recharge its battery, the agent may be unable to remove itself from the undesirable interaction if the agent does not follow the instructions of the controller. The controller, with its global view of the undesirable interaction, is in a better position to resolve the interaction than the individual agents. Therefore, although it is possible for an agent to be implemented to disobey a controller, care should be taken in identifying situations where this is appropriate.
6 Multi-agent planning to implement centralised control

In the previous chapter we outlined our centralised control mechanism and demonstrated its implementation using two examples. Each controller uses a resolution mechanism to determine how to direct the agents to resolve the undesirable interaction. In the example controllers, however, the implementation of the resolution mechanism was relatively simple, using a fixed pre-programmed heuristic method in both cases.

There are advanced tools developed in the literature that can be used as part of the implementation of a more sophisticated controller. In particular, planning tools exist (e.g. MBP [BCP⁺01] and UMOP [JV00]) that are able to generate plans that allow agents to achieve their goals in non-deterministic domains. A non-deterministic domain allows for uncertainty when sensing the current state of the environment and in the effects of actions. This is an important property of the multi-agent grid environment in Sinatra and so it is natural to consider planning tools as a possible mechanism to direct an agent to achieve its goal.

In this chapter we investigate the use of one such tool, UMOP, to develop a resolution mechanism for undesirable interactions, allowing us to utilise multi-agent planning techniques within the controller implementation. We begin by describing UMOP and the process by which UMOP can be used to generate and simulate the running of a plan. We then describe the implementation of a controller that uses UMOP as part of its resolution mechanism.

6.1 Universal multi-agent OBDD-based planner

Jensen and Veloso have created the planning framework UMOP, which stands for Universal Multi-agent OBDD-based Planner [JV00]. UMOP is a planning system designed for non-deterministic multi-agent domains, which is capable of performing plan-generation and plan-simulation. UMOP uses plan-generation to create a plan for a particular domain. Using a previously generated plan, UMOP uses plan-simulation to execute the plan via a text-based interface.

The input to UMOP is a planning problem or domain, described using a specially designed domain description language known as NADL, or Non-deterministic Agent
Domain Language. An NADL domain description is used by UMOP to generate a plan for the specified planning problem. During plan-generation, UMOP translates the NADL planning problem into a non-deterministic finite automaton (NFA). The output of UMOP is a symbolic encoding of the NFA structure as a set of ordered binary decision diagrams (OBDDs) [JV00]. An OBDD is a rooted directed acyclic graph that allows for the efficient encoding of boolean functions.

UMOP is explicitly designed for multi-agent planning domains, comprising synchronised agents that act concurrently and a separate set of uncontrollable agents that model the environment. To achieve this distinction between agents, the agents in an NADL domain description are divided into two groups: system agents and environment agents. A plan generated by UMOP is capable of instructing the system agents how to act, but the environment agents remain uncontrollable, meaning that the actions of the environment agents cannot be part of the generated plan. In practice, the environment agents act randomly, or possibly adversarially, depending on how the domain is defined [JV00, JVB01].

The agents in UMOP, both the system and environment agents, perform actions that are assumed to be executed synchronously and for an equal duration. At each step of a plan, all the agents perform exactly one action, which is represented by UMOP as an action tuple called a joint action. A valid domain description in NADL requires that the system and environment agents constrain a disjoint set of variables, meaning that their actions cannot interact on shared objects [JV00]. This restriction means that system and environment agents are only able to interact in very limited ways.

The plans generated by UMOP are known as universal plans. A universal plan [Sch87] is a form of lookup table containing state-action rules that map every possible state of the domain to an action to be performed in that state. Universal plans are executed in a loop, where action selection from the plan is interleaved with perceiving the state of the world. This means that universal plans are suited to non-deterministic domains.

A universal plan is similar in form to the state-action tables generated from a behaviour by the STAG process and used by Sinatra agents to determine their actions. However, the universal plans generated by UMOP are in a non-human readable format and so cannot be used directly to implement Sinatra agents.

The source code for UMOP, packaged with a number of domain examples, is available from the UMOP website [JV]. Amongst the domain examples presented by Jensen and Veloso, their soccer domain [JV00] appears to have the greatest similarity to the multi-agent systems that are the focus of this thesis. The soccer domain involves two teams of system and environment agents moving around a grid of locations and a representation of a ball. The system agents attempt to coordinate their actions in order to carry the ball to the far right column of the grid. A
goal is scored if the system agents carry the ball to this column and there are no environment agents in the column. Each system agent has the ability to move north, east, south and west within the environment and to pass the ball to another member of their team. A team of environment agents form the opposition, with the ability to move north, east, south and west only. The team of environment agents attempt to prevent the system agents from scoring a goal.

The UMOP system agents contained within the soccer domain have a number of similarities to agents in the grid environment implemented in Sinatra. The system agents are able to move freely around a grid environment and have the ability to interact with other system agents. The distinction between system agents and environment agents also mirrors the concept of autonomous agents that perceive other agents in the environment as movable obstacles.

A property of the universal plans generated by UMOP also highlights why UMOP is a potential tool for implementing a controller. The universal plans generated by UMOP control all system agents together, as part of the joint action tuple. This has the effect that multiple system agents in UMOP appear to operate as a single autonomous agent with multiple effectors in the domain. This means that UMOP is capable of handling multiple concurrently acting agents under the direction of a single plan, which is exactly the behaviour we want to achieve in a controller.

We now describe the process of plan-generation and plan-simulation in more detail.

6.1.1 Plan-generation

Using an NADL domain description of a planning problem, UMOP can be used to generate a universal plan. A toy example of an NADL domain description is shown in Listing 6.1, which we use to explain properties of NADL.

An NADL domain description consists of five parts: (i) a definition of the state variables, (ii) a description of the system agents, (iii) a description of the environment agents, (iv) a specification of the initial condition and (v) a specification of the goal condition [JV00].

The state variables are used to define the valid states of the domain. State variables can be either integer variables (nat(2) value) or Boolean variables (bool light_on). For integer variables used to define the domain, the variable must take all values between 0 and 2^n-1, where n is the size specified in the variable declaration, i.e. nat(n).

The system agents are those that can be controlled by the generated plan. Each system agent must have a name (agent1) and a set of actions that it can perform (add_one). Each action is described in terms of a set of state variables, a precondition formula and an effect formula. The state variables are those that change when the action is performed; therefore, the agent’s action is said to constrain these state variables. The effect formula specifies the values that the constrained variables will
take in the next state, after the action has been performed. A primed variable notation is used to refer to the value of the variable in the next state.

The environment agents are defined similarly to the system agents, although their actions must not constrain the same state variables as the actions of the system agents. Each agent must perform exactly one action at each time step, which means that there is only one joint action tuple based on this domain description: \{add\_one, switch\}. Therefore, in every time step of the plan execution, the system agent agent1 will execute the add\_one action and the environment agent agent2 will execute the switch action.

The NADL domain description is written in a text file and given as input to UMOP for plan-generation. A planning algorithm is also specified for plan-generation. UMOP is implemented to be able to generate universal plans using a selection of different OBDD-based planning algorithms. In our investigations using UMOP, we have considered the use of three types of plans.

- **Strong plan.** When executed, a strong plan is guaranteed to achieve the goal [CPRT03].

- **Strong cyclic plan.** A strong cyclic plan is guaranteed to achieve the goal
with an iterative trial-and-error strategy [CPRT03]. This means that all execution paths of the plan have the possibility of terminating and that the goal is guaranteed to be achieved when the plan does terminate. If a strong plan exists, it is also a strong cyclic plan. However, a strong cyclic plan can be applied in situations where a strong plan does not exist due to a non-deterministic domain.

- **Strong cyclic adversarial plan.** A strong cyclic adversarial plan guarantees goal achievement independently of the actions of the environment [JVB01]. Developed specifically for UMOP, the generated universal plan takes into account the actions of the environment agents, which attempt to prevent the system agents from achieving their goal.

The planning algorithms that generate these three types of plans are designed to find the appropriate strong solution despite non-determinism in the domain. This means that non-determinism from the actions and interactions of the agents are taken into account when developing the plan.

For the NADL domain description described in Listing 6.1, the actions of the environment agent cannot be adversarial, because the goal state can only be influenced by the actions of the system agent. There are also no non-deterministic actions defined for the system or environment agents. Therefore, a strong planning algorithm is the most suitable for plan-generation in this example.

While a strong cyclic planning algorithm is suitable when non-deterministic actions have been defined, it can also be used if there is any concern of possible interactions between the UMOP agents leading to states where the goal cannot be achieved directly. As a strong plan is also a strong cyclic plan, the use of a strong cyclic planning algorithm will generate a strong plan if one exists.

Using a specified planning algorithm, the NADL domain description is translated by UMOP into an NFA. To complete the plan-generation process, UMOP encodes the NFA in a symbolic form using OBDDs, which become the generated universal plan. The output of UMOP plan-generation is a text file containing the OBDD encoding of the generated plan. This text file is in a non-human readable format; therefore, it is not possible to determine any details of the generated plan from the output file. In order to gain some insight into the nature of the plan generated by UMOP, plan-simulation must be used to simulate the execution of the plan.

### 6.1.2 Plan-simulation

The output file containing the universal plan from the plan-generation process is given as input to UMOP for plan-simulation. Via a text prompt in a shell environment, the user can select to step through solutions of the plan from different specified initial states. Listing 6.2 shows an example of UMOP plan-simulation for
A solution of the universal plan is referred to in UMOP as a *trace*. For each trace, the user is first prompted to enter the initial values of the state variables. The user then steps through the solution of the plan from this particular initial state. During each step of the execution, the joint action of the system and environment agents is specified and the current state is updated. At the end of the execution the user can request to make a new trace. If this is selected, the user will again be asked to specify an initial state. If the same initial state is specified, then the plan-simulation will output an alternative trace of the plan from the specified initial state.

Plan-simulation presents the only opportunity for the user to verify the efficacy of a generated universal plan. Using the UMOP plan-simulation, the user is able to test the generated plan from different initial states and to observe alternative executions of the plan from this initial state. However, there is no indication to the user how many alternative traces exist for each initial state or whether a given trace may be considered optimal in terms of number of steps.

For a more complicated planning problem, it is very difficult for the user to understand the strategy used by the plan to achieve the goal state. The best that can be achieved is for the user to develop a general feel for the plan strategy by repeated simulation and observation of the execution of the plan.

### 6.2 Example: Using UMOP to manage row obstruction interactions

The aim of this example is to investigate the implementation of a controller using UMOP as an alternative mechanism for resolving an undesirable interaction. UMOP seems to have the potential to allow us to implement a more advanced resolution mechanism than the fixed pre-programmed heuristic methods used for the repeated state controller and the stationary controller.

Due to restrictions encountered when implementing the UMOP controller, which will be discussed later (Section 6.2.3), the UMOP controller is only designed to resolve a very specific type of undesirable interaction. We will refer to this type of undesirable interaction as a *row obstruction interaction*. Figure 6.1 shows an example of a row obstruction interaction, which is defined as involving exactly two agents that obstruct each other along a row of the grid environment. A row obstruction interaction is a very specific form of stationary interaction and also the most commonly observed form of stationary interaction.
Trace 1 
Please enter the initial state of the trace 
value [0-3] = 0 
light_on [0-1] = 1

Step 1

Joint action of system agents: 
action of agent1=add_one

Joint action of environment agents: 
action of agent2=switch

Current state: 
value=1 light_on=0

Continue current trace ? [y/n] : y

Step 2

Joint action of system agents: 
action of agent1=add_one

Joint action of environment agents: 
action of agent2=switch

Current state: 
value=2 light_on=1

Continue current trace ? [y/n] : y

Step 3

Joint action of system agents: 
action of agent1=add_one

Joint action of environment agents: 
action of agent2=switch

Current state: 
value=3 light_on=0

Trace reached a goal state!

make a new trace ? [y/n] : n

Listing 6.2: Plan-simulation of the toy NADL domain.
6.2.1 Implementing the UMOP controller

We have already identified that the UMOP controller will be designed to manage row obstruction interactions. We now describe the implementation of the controller, making note of the implementation features that remain the same from the controllers implemented in Chapter 5 and implementation points related specifically to the use of UMOP. As the row obstruction interaction is a special case of a stationary interaction, the implementation of the UMOP controller uses many of the features described for the implementation of the stationary controller (Section 5.3.1).

How the UMOP controller is invoked

Similar to the previous controllers implemented in Chapter 5, the agents themselves will monitor their own actions to determine whether they are participating in a row obstruction interaction. As the row obstruction interaction is a specific form of stationary interaction, the same perception mechanism described for the stationary controller can be used to allow the agent to determine when to invoke the UMOP controller.

As we have specified, the UMOP controller is only designed to handle row obstruction interactions. From an individual agent’s point of view, the only perceptible difference between a row obstruction interaction and any other stationary interaction is that the agent is obstructed while facing east or west within the grid environment. An agent that is facing east or west and cannot move towards its goal due to an obstructing agent is being obstructed along a row of the grid.

The directive for the agent to invoke the UMOP controller can be written as

\[ \text{notifyumop} \leftarrow \text{obstructedrow} \]

where the \text{obstructedrow} token represents that the agent has been prevented from moving along a row of the grid environment by another agent and the \text{notifyumop}
token represents the action of the agent sending a message to invoke the UMOP controller.

This directive must be added to the appropriate behaviour or behaviours in the same manner as described for the implementation of the stationary controller (c.f. Listing 5.3). When an agent sends a message to invoke the UMOP controller, the message will contain the same information used by the agent when determining whether it is participating in a stationary interaction. Specifically, the agent will send the controller its agent ID number and the ID number of the obstructing agent.

**How the controller identifies a row obstruction interaction**

The agents that invoke the UMOP controller may have misidentified a row obstruction interaction. A row obstruction interaction only occurs when there are exactly two agents involved in the interaction. An agent that invokes the UMOP controller has only determined that it is being obstructed by another agent along a row of the grid environment; the agent does not know how many other agents might be participating in the interaction.

The UMOP controller uses the same mechanism described for the stationary controller to identify when a row obstruction interaction is taking place. The only extra requirement is that when the UMOP controller finds a set of mutually obstructing agents, the controller also checks that there are only two agents contained within the set. If the controller finds a pair of agents that are mutually obstructing each other, then it has found a row obstruction interaction. The controller will issue instructions to these agents to try to resolve the interaction. If the UMOP controller finds a set of more than two agents that are mutually obstructing each other, then the controller has not found a row obstruction interaction. The controller will issue no instructions to this set of agents.

**How the controller resolves a row obstruction interaction**

We now discuss the main aspect of the use of UMOP as part of the implementation of a controller: the implementation of the resolution mechanism for an undesirable interaction. UMOP was selected as an implementation tool because of its ability to handle multiple concurrently acting agents under the direction of a single plan. This is exactly the behaviour we want to achieve with the resolution mechanism for a centralised controller.

To use UMOP as an implementation tool for a controller, a plan is required that will direct the agents how to act. The initial state of the plan will be a row obstruction interaction and the goal state will be a state such that the undesirable interaction is resolved, meaning that the agents no longer obstruct each other. In order to create this plan, the plan-generation feature of UMOP will be used.
The generation of universal plans can be carried out in advance, similar to the state-actions tables used for behaviours. In order to generate a plan, a domain description for the planning problem must be created. Listing 6.2 shows the NADL domain description for a row obstruction interaction.

In UMOP, a generated universal plan maps each state to a joint action tuple, being a set of single concurrent actions to be undertaken by each system agent and environment agent during the next time step. In our context, the system agents are the agents whose actions are going to be managed by the UMOP controller. Therefore, the NADL domain description for a row obstruction interaction contains two system agents. This will allow the generated universal plan to control the actions of both agents participating in the row obstruction interaction.

The actions defined for the system agents allow the agents to turn left and right, to move forwards and to wait. The actions of the second system agent are the same as those for agent \(a_1\) and so have been omitted from Listing 6.2 to avoid unnecessary repetition. The precondition of the move actions ensures that the agent will not move into a location already occupied by the other system agent. The precondition also specifies that the agent must be facing in the direction that it is moving.

The wait action has been defined to allow the system agents to remain stationary for a time step. This action is necessary because the joint action tuple specified by the plan must contain an action for each agent in every step of its execution. Without the wait action, a plan may be forced to turn or to move an agent when having the agent perform no action would be preferable. In the case of the row obstruction interaction, the wait action allows an agent to wait for another agent to finish moving out of its way without having to perform unnecessary actions itself. In this way, the wait action allows more efficient plans to be generated by UMOP.

Environment agents are considered to act randomly by UMOP and are not controllable by a generated plan. Although other agents may exist in the environment, they are not participating in the row obstruction interaction. Therefore, at this point, we do not include any environment agents in the domain description. Later, we consider the addition of environment agents who may interfere with the agents participating in a row obstruction interaction (Section 6.3).

The goal state defined in the domain description specifies that the agents must be in the same column of the grid environment. As the agents in a row obstruction interaction are obstructing each other along a single row of the environment, the agents will both be released from the interaction if they move to occupy the same column. This is a very simple goal for the plan to attempt to achieve. A simple goal was selected due to restrictions encountered when implementing the UMOP controller, which will be discussed later (Section 6.2.3).

The NADL domain description of Listing 6.2 was used to generate a universal plan in UMOP. The plan-generation feature was directed to find a strong cyclic universal
VARIABLES

# Variables for agents a1 and a2.

nat(4) a1x  # The x coordinate of a1, 0 – 15.
nat(3) a1y  # The y coordinate of a1, 0 – 7.
nat(2) d1   # The direction a1 is facing, 0 – 3.
nat(4) a2x

nat(3) a2y

nat(2) d2   # north 0, east 1, south 2, west 3.

SYSTEM

# System agents, who will be directed by the universal plan.

agt: a1     # Agent a1.

turn_left1

  con: d1
  pre: d1 = 0
  eff: d1' = 3

  turn_left1

    con: d1
    pre: ¬(d1 = 0)
    eff: d1' = d1 - 1

  turn_right1

    con: d1
    pre: d1 = 3
    eff: d1' = 0

  turn_right1

    con: d1
    pre: ¬(d1 = 3)
    eff: d1' = d1 + 1

move_east1

  con: a1x
  pre: ¬(a1y = a2y ∧ a1x+1 = a2x) ∧ d1 = 1
  eff: a1x' = a1x + 1

move_west1

  con: a1x
  pre: ¬(a1y = a2y ∧ a1x-1 = a2x) ∧ d1 = 3
  eff: a1x' = a1x - 1

move_north1

  con: a1y
  pre: ¬(a1x = a2x ∧ a1y+1 = a2y) ∧ d1 = 0
  eff: a1y' = a1y + 1

move_south1

  con: a1y
  pre: ¬(a1x = a2x ∧ a1y-1 = a2y) ∧ d1 = 2
  eff: a1y' = a1y - 1

Listing 6.3: (part 1) The row obstruction interaction domain description.
wait1
con: a1x
pre: true
eff: a1x' = a1x

agt: a2  # Agent a2.
# Actions defined as agent a1, using the variables a2x, a2y and d2.
# Omitted here to avoid repetition.

ENVIRONMENT
# There are no environment agents in this domain description.

INITIALLY  # Initial state of the domain.
# The agents are next to each other along the same row of the grid.
a1y = a2y \land (a1x+1 = a2x \lor a1x-1 = a2x)

GOAL  # Goal of the universal plan.
a1x = a2x  # The agents move to be in the same column.

Listing 6.2: (part 2) The row obstruction interaction domain description.
of each agent for input to the plan-simulation. This forms the initial state used during the plan-simulation. After this, the controller specifies to step through the first solution of the plan. While this is occurring, the controller records the actions specified by the plan trace. These will be used by the controller to instruct the agents how to act.

An example of the output of UMOP plan-simulation for the universal plan generated for a row obstruction interaction is shown in Listing 6.2. The interactive input that would be supplied by the UMOP controller is highlighted in bold.

Using plan-simulation, multiple traces can be viewed of the generated plan from the same initial state to the goal state. Each different trace of the plan describes a different set of actions for the agents to perform to resolve the row obstruction interaction. However, for a given initial state, the order of the traces that are viewed using the plan-simulation feature is always the same. The UMOP controller only needs one set of instructions from the generated plan to be able to instruct the agents. Therefore, the controller uses the actions specified in the first trace to resolve a row obstruction interaction.

The UMOP controller parses the actions specified in the trace and sends these as the appropriate tokens to the two agents participating in the interaction. In the case of the example output shown in Listing 6.2, the UMOP controller will send four messages to each agent.

Trace 1

Please enter the initial state of the trace

a1x [0-15] = 8
a1y [0-7] = 7
d1 [0-3] = 1
a2x [0-15] = 9
a2y [0-7] = 7
d2 [0-3] = 3

Step 1

Joint action of system agents:
action of a1=wait1
action of a2=wait2

Joint action of environment agents:
Current state:
a1x=8 a1y=7 d1=1 a2x=9 a2y=7 d2=3

Continue current trace ? [y/n] : y

Listing 6.3: (part 1) Plan-simulation of the row obstruction interaction.
Step 2

Joint action of system agents:
  action of $a_1$ = turn_right1
  action of $a_2$ = wait2

Joint action of environment agents:

Current state:
  a1x=8 a1y=7 d1=2 a2x=9 a2y=7 d2=3

Continue current trace? [y/n] : y

Step 3

Joint action of system agents:
  action of $a_1$ = move_south1
  action of $a_2$ = wait2

Joint action of environment agents:

Current state:
  a1x=8 a1y=6 d1=2 a2x=9 a2y=7 d2=3

Continue current trace? [y/n] : y

Step 4

Joint action of system agents:
  action of $a_1$ = turn_left1
  action of $a_2$ = move_west2

Joint action of environment agents:

Current state:
  a1x=8 a1y=6 d1=1 a2x=8 a2y=7 d2=3

Trace reached a goal state!

make a new trace? [y/n] : n

Listing 6.2: (part 2) Plan-simulation of the row obstruction interaction.
How the agents respond to and are released from the UMOP controller

The UMOP controller uses the standardised message passing interface described for the centralised control mechanism. Therefore, the agents will follow the instructions received from the UMOP controller and then return to following their behaviours.

6.2.2 Demonstration

Figure 6.2 to Figure 6.7 show the UMOP controller resolving a row obstruction interaction. The universal plan used to resolve these row obstruction interactions has the goal to move the two agents so that they occupy the same column and so can move past each other. The actions available to the agents are to turn 90° to the left or the right, to move forwards and to wait. The agents perform the actions specified by the universal plan, as shown in the output of the plan-simulation feature in Listing 6.2.

![Figure 6.2: Two agents participating in a row obstruction interaction.](image)

![Figure 6.3: The UMOP controller identifies that a row obstruction interaction is taking place and consults the universal plan. The output of the plan is used by the controller to instruct the agents how to act. The first action in the plan trace was for both agents to wait.](image)
Figure 6.4: The agents follow the instructions of the UMOP controller, with the goal of moving to be in the same column.

Figure 6.5: The agents follow the instructions of the UMOP controller, with the goal of moving to be in the same column.

Figure 6.6: The two agents execute the final instructions given to them by the UMOP controller. The agents are now occupying the same column. Therefore, the goal state of the plan has been reached and so the row obstruction interaction has been resolved.
Figure 6.7: The two agents are now directed by their behaviour-switch to continue using the simple goal-directed navigation to reach their goals. Therefore, both agents can again autonomously determine their own actions.

6.2.3 Conclusion

The demonstration of the UMOP controller shows the UMOP controller is able to function as intended. The agents invoke the UMOP controller when they believe they are participating in a row obstruction interaction, the UMOP controller identifies if a row obstruction interaction is occurring and the agents follow the instructions received from the UMOP controller. As the row obstruction interaction is so specific, there is little variation in the possible row obstruction interactions that the UMOP controller will be required to handle, except the position within the grid where the interaction occurs.

The main aim of the UMOP controller example, however, was to demonstrate that an alternative implementation tool can be used to implement the resolution mechanism of a controller. UMOP is used to provide a more sophisticated resolution mechanism than the fixed pre-programmed heuristic strategy used by the repeated state controller and the stationary controller. The integration of the UMOP controller within the agents followed the standard implementation pattern that has been established for the centralised control mechanism. In addition, specific modifications required for the agents to be able to interact with the UMOP controller are the same as those described for the stationary controller. Therefore, our analysis of the implementation of the UMOP controller focusses on the use of UMOP to implement the resolution mechanism.

As will be clear from the description of the UMOP controller implementation, a number of simplifications were introduced to the undesirable interaction. These simplifications were required to be able to implement the resolution mechanism using UMOP, because of limitations with UMOP as an implementation tool for a controller.

It was initially intended that the UMOP controller would be able to handle all stationary interactions, in the manner of the stationary controller. This meant that
the UMOP controller would be able to handle stationary interactions involving two or more agents and where the stationary interactions can occur anywhere within the grid environment. This was not possible, however, as we were unable to achieve a generic NADL domain description that would describe all stationary interactions and generate a suitable plan that could be used by the controller.

The greatest limitation with using UMOP as an implementation tool for a controller comes from generating a universal plan that can be used to resolve an undesirable interaction. The number and size of the variables needed to represent a domain in NADL significantly increases the time taken for UMOP to generate a universal plan and to simulate the running of the plan. Plan-generation for the UMOP controller is carried out in advance and so the time taken to generate the plan is not, in itself, a major issue for the implementation of the UMOP controller. Plan-simulation on the other hand must be carried out at runtime every time the UMOP controller attempts to resolve an undesirable interaction. Therefore, in order to develop and test the resolution mechanism, the time taken to simulate the generated plan is an important factor.

We have supposed that there are typically up to six agents inhabiting the Sinatra grid environment. Undesirable interactions can occur between two or more of these agents. NADL domain descriptions must contain an explicit representation of all of the agents in the domain. Therefore, a different NADL domain description would be required for undesirable interactions involving different numbers of agents.

In order for the generated universal plan to control the agents that are participating in the undesirable interaction, the agents must be defined in the NADL domain description as system agents, each with their own set of variables and defined actions. This in itself leads to an increase in the time taken to generate a universal plan.

As an example, we experimented with the addition of further agents to the domain description defined in Listing 6.2. The goal remained unchanged, being to move the two initial system agents to the same column. The time taken and the size of the generated output file for different numbers of agents are shown in Table 6.1. The times shown are the plan-generation time recorded by UMOP. All tests were carried out using an Intel® Core™2 Duo CPU with 7.7 GiB RAM and running Ubuntu 11.04.

<table>
<thead>
<tr>
<th>Number of agents</th>
<th>Plan-generation</th>
<th>Output plan file</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.36 seconds</td>
<td>107 KiB</td>
</tr>
<tr>
<td>3</td>
<td>6.41 seconds</td>
<td>1 MiB</td>
</tr>
<tr>
<td>4</td>
<td>10 hours</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table 6.1: Plan-generation time and size of the generated output file for the row obstruction interaction domain containing different numbers of agents.
The results of Table 6.1 show that the number of agents in the domain leads to a dramatic increase in the time taken to generate a universal plan and the size of the output plan file. Indeed, after approximately 10 hours of execution, the plan-generation for a four agent domain terminated before a plan was generated. Instead, UMOP recorded a ‘profiling time alarm’.

Plan-simulation involves loading a generated universal plan into UMOP; therefore, the size of the universal plan has a direct impact on the time taken for plan-simulation. UMOP does not provide any internal timings for loading a plan during plan-simulation, however, it was observed to be proportional to the size of the generated plan file and to be comparable to the time taken for plan-generation. Therefore, in order to be able to develop and test a resolution mechanism implemented in UMOP, the number of agents participating in the stationary interaction was restricted to just two agents.

The requirement to restrict the number of agents that are managed by the UMOP controller to just two agents meant that the UMOP controller would only be able to handle a subset of stationary interactions. Even though the UMOP controller would clearly be unable to handle all stationary interactions, being able to resolve stationary interactions involving only two agents is still a useful exercise.

Despite this simplification to the undesirable interaction, the associated NADL domain description was again found to take a significant amount of time for plan-generation. This was because of the effect of the goal state on the time taken to generate the universal plan. When we first developed a domain description for stationary interactions involving two agents, a goal was considered where the two agents should swap places. However, this strategy for resolving the stationary interaction resulted in a universal plan that took approximately 30 minutes to generate and an output file that was 224MiB in size.

Therefore, the two agent stationary interaction to be handled by UMOP was simplified further. The stationary interaction would now involve only two agents and these agents would obstruct each other only along a row of the grid environment, which we have termed a row obstruction interaction. By restricting the UMOP controller to only handle row obstruction interactions, the simpler goal that the agents should move to be in the same column could be used to resolve the interaction, resulting in a smaller generated plan file.

It is for these reasons that the UMOP controller was only designed to handle row obstruction interactions. Having developed the NADL domain description for the row obstruction interaction, plan-simulation was used to test the efficacy of the resulting plan.

When testing the universal plan for resolving the row obstruction interaction, a number of different plan traces between the initial state and the goal state can be viewed. The first output trace using plan-simulation, shown in Listing 6.2, begins
with a joint action where both system agents perform the wait action. Therefore, it is clear that not all plan traces are optimal. In fact, using the plan-simulation feature to view multiple traces of the universal plan for a row obstruction interaction from the same initial state, the number of steps required for the agents to achieve the goal state was seen to vary from between three and seven steps.

Restricting the UMOP controller to only handle row obstruction interactions allowed the NADL domain description and generated universal plan to be developed and tested in an efficient manner. Unfortunately, the resulting resolution mechanism is also obviously very limited. For example, the UMOP controller is unable to resolve what might be called a column obstruction interaction, where two agents mutually obstruct each other from moving along a column of the grid environment. Equally, the UMOP controller is unable to resolve a row obstruction interaction involving three agents.

As a potential solution to the restricted applicability of the UMOP controller, additional domain descriptions could be developed for other specific types of stationary interaction. This might involve a different domain description and associated universal plan for stationary interactions along a row and along a column, as well as different domain descriptions for different numbers of agents participating in these undesirable interactions. The resulting collection of universal plans could be used by one or more controllers to handle a greater selection of stationary interactions. However, the specific nature of each plan, combined with the issues stated above for developing plans for more than two agents, would make this a very time consuming process.

It is clear that implementing a fully functional resolution mechanism for an undesirable interaction using UMOP is not likely to be a practical strategy. The selected undesirable interaction for the controller to handle must be subdivided by the system designer into different specific interactions that contain a fixed number of agents and that can be resolved by achieving a simple goal. Each specific instance of the undesirable interaction requires a separate domain description to be written and for the associated plan to be generated and tested. In addition, the time taken for plan-generation and plan-simulation increases dramatically for domain descriptions containing more than two agents. It appears to be very difficult to create a more generic multi-use plan using UMOP that could be used to resolve more than just a specific instance of an undesirable interaction.

6.3 Handling other agents in the environment

One of the main reasons for selecting UMOP as a possible implementation tool for the resolution mechanism of a controller was because of the distinction maintained in UMOP between system agents and environment agents. Specifically, system agents
are controllable by the universal plan, whereas environment agents are not. The environment agents have the potential to interfere with the actions of the system agents however, with the possibility of preventing them from being able to achieve their goal.

The interference caused by an independently acting agent can lead to a situation where an agent tries but fails to be able to act as instructed by the controller. This is an important quality of NADL domains because, as we have already described for the centralised control mechanism (Section 5.4.3), controllers are typically unable to handle undesirable interactions if an agent fails to act as instructed by the controller. We now discuss the possibility of using universal plans generated by UMOP to be able to overcome this issue.

### 6.3.1 Adding an environment agent

The UMOP controller was demonstrated using a domain containing only two agents. While it has not been observed, it is possible for the UMOP controller to fail to be able to resolve a row obstruction interaction if a third agent interferes with the two system agents while they are following the instructions of the controller. This can occur when an agent following the instructions of the UMOP controller is prevented from moving forwards by another agent moving into this location.

To try to allow UMOP to include this type of situation during the generation of the universal plan, an environment agent was introduced to the row obstruction interaction domain description shown in Listing 6.2. The initial state and goal state were not changed. A universal plan was generated for this updated domain description using a strong cyclic planning algorithm. The plan was generated in 6.41 seconds and the output plan file was 1MiB.

An execution trace of plan-simulation of the three-agent plan is shown in Listing 6.2. When tracing this three-agent plan during plan-simulation, the user must begin by specifying the location of the environment agent within the domain, as well as the locations of the system agents. The initial state was specified such that the environment agent is next to one of the system agents and so has the possibility to interfere with their actions. The user cannot influence the actions of the environment agent during the execution trace, however, the actions of the environment agent are specified as part of the joint action tuple.

The generated plan may contain a separate trace for all possible combinations of environment agent actions, although this has not been verified. The traces observed for the three-agent row obstruction domain always resolved the interaction by moving the agent that is not being obstructed by the environment agent. Therefore, in practice, the environment agent does not interfere with the system agents as they move to achieve their goal.
Trace 1
Please enter the initial state of the trace

\[
a_{1x} \ [0-15] = 8 \\
a_{1y} \ [0-7] = 7 \\
d_{1} \ [0-3] = 1 \\
a_{2x} \ [0-15] = 9 \\
a_{2y} \ [0-7] = 7 \\
d_{2} \ [0-3] = 3 \\
a_{3x} \ [0-15] = 8 \\
a_{3y} \ [0-7] = 6 \\
d_{3} \ [0-3] = 1
\]

Step 1

Joint action of system agents:
- action of \( a_1 = \text{wait1} \)
- action of \( a_2 = \text{turn\_left2} \)

Joint action of environment agents:
- action of \( a_3 = \text{wait3} \)

Current state:
- \( a_{1x}=8 \) \( a_{1y}=7 \) \( d_{1}=1 \) \( a_{2x}=9 \) \( a_{2y}=7 \) \( d_{2}=2 \) \( a_{3x}=8 \) \( a_{3y}=6 \) \( d_{3}=1 \)

Continue current trace? [y/n]: y

Step 2

Joint action of system agents:
- action of \( a_1 = \text{wait1} \)
- action of \( a_2 = \text{move\_south2} \)

Joint action of environment agents:
- action of \( a_3 = \text{turn\_right3} \)

Current state:
- \( a_{1x}=8 \) \( a_{1y}=7 \) \( d_{1}=1 \) \( a_{2x}=9 \) \( a_{2y}=6 \) \( d_{2}=2 \) \( a_{3x}=8 \) \( a_{3y}=6 \) \( d_{3}=2 \)

Continue current trace? [y/n]: y

Listing 6.3: (part 1) Plan-simulation of the three-agent row obstruction interaction.
Step 3

Joint action of system agents:
- action of a1=move_east1
- action of a2=move_south2

Joint action of environment agents:
- action of a3=turn_left3

Current state:
- a1x=9 a1y=7 d1=1
- a2x=9 a2y=5 d2=2
- a3x=8 a3y=6 d3=1

Trace reached a goal state!

make a new trace? [y/n]: n

Listing 6.2: (part 2) Plan-simulation of the three-agent row obstruction interaction.

6.3.2 Adding an adversarial environment agent

Adding an environment agent to the domain and placing it next to the system agents did not seem to allow the environment agent to interfere with the actions of the system agents. Even though the environment agent acts as an obstruction, the system agents were not prevented from achieving their goal by the environment agent. Therefore, the generated plan would not allow the UMOP controller to handle interference between the environment agent and the system agents. However, UMOP contains an implementation of an adversarial planning algorithm, designed specifically to be used in domains where environment agents attempt to prevent the system agents from achieving their goal.

To allow UMOP to include an adversarial environment agent, a plan was generated for the three-agent row obstruction domain description using UMOP’s strong cyclic adversarial planning algorithm. It was hoped that the use of an adversarial environment agent would allow possible interference between the environment agent and the agents participating in the row obstruction interaction to be observed and the use of the plan to implement the resolution mechanism of a controller to be investigated.

Using the strong cyclic adversarial planning algorithm, the time taken to generate the universal plan increased to 638.93 seconds and the output plan file was 2.1MiB. The initial state and the goal state were unchanged from the original row obstruction domain description. Listing 6.2 shows an execution trace of the generated three-agent adversarial plan. The initial state was specified such that the environment agent is next to one of the system agents and so has the opportunity to interfere with their actions. Again, the user cannot influence the actions of the environment...
agent during the execution trace, however, the actions of the environment agent are specified as part of the joint action tuple.

In this trace, the environment agent appears to act in a possibly adversarial manner by occupying a location that either of the system agents might want to move to in order to achieve their goal. However, the system agents are still able to achieve their goal without difficulty.

It is hard to observe via plan-simulation that the environment agent is actually acting in an adversarial manner, although the environment agent was observed to act differently to the non-adversarial environment agent in Listing 6.2 during multiple traces from the same initial state. However, in practice, the environment agent does not interfere with the system agents as they move to achieve their goal.

Trace 1
Please enter the initial state of the trace

\[
\begin{align*}
    a_{1x} [0-15] &= 8 \\
    a_{1y} [0-7] &= 7 \\
    d_{1} [0-3] &= 1 \\
    a_{2x} [0-15] &= 9 \\
    a_{2y} [0-7] &= 7 \\
    d_{2} [0-3] &= 3 \\
    a_{3x} [0-15] &= 8 \\
    a_{3y} [0-7] &= 6 \\
    d_{3} [0-3] &= 1
\end{align*}
\]

Step 1

Joint action of system agents:
    action of a1=wait1
    action of a2=turn_left2

Joint action of environment agents:
    action of a3=turn_left3

Current state:
    a1x=8 a1y=7 d1=1 a2x=9 a2y=7 d2=2 a3x=8 a3y=6 d3=0

Continue current trace? [y/n] : y

Listing 6.3: (part 1) Plan-simulation of the three-agent adversarial row obstruction interaction.
Step 2

Joint action of system agents:
  action of a1=wait1
  action of a2=move_south2

Joint action of environment agents:
  action of a3=wait3

Current state:
  a1x=8 a1y=7 d1=1 a2x=9 a2y=6 d2=2 a3x=8 a3y=6 d3=0

Continue current trace? [y/n] : y

Step 3

Joint action of system agents:
  action of a1=move_east1
  action of a2=turn_left2

Joint action of environment agents:
  action of a3=turn_left3

Current state:
  a1x=9 a1y=7 d1=1 a2x=9 a2y=6 d2=1 a3x=8 a3y=6 d3=3

Trace reached a goal state!

make a new trace? [y/n] : n

Listing 6.2: (part 2) Plan-simulation of the three-agent adversarial row obstruction interaction.

6.3.3 Observations

Neither of the generated three-agent plans appear to be able to represent possible interference from a third agent in the environment during a row obstruction interaction. Despite positioning the environment agent next to the system agents, the environment agent does not appear to interfere with the system agents, even when an adversarial planning algorithm is used.

The output of both three-agent generated plans was observed via plan-simulation for repeated traces from the same initial state. In all traces, the system agents were easily able to achieve their goal. Specifically for the adversarial environment agent, where some adversarial actions were expected, it was hard to interpret the actions of the environment agent as being adversarial or as interfering with the actions of the system agents in some way.
Plan-simulation in UMOP showed execution traces for both plans for a number of different sets of actions of the environment agent from the same initial state. Therefore, it may be true that the generated universal plan contains an execution trace for any set of actions performed by the environment agent. If it were possible for the user to specify the actions of the environment agent during the course of plan-simulation, then the generated universal plan may contain a suitable plan execution to allow the system agents to still achieve their goal. However, while the actions of the environment agent are somehow specified automatically in the joint action tuple during plan-simulation, the environment agent does not appear to interfere with the system agents.

The concept of environment agents is defined in UMOP so that the actions of these agents are uncontrollable by a generated universal plan. Unfortunately, the actions of the environment agents are also uncontrollable by the user or by a controller implemented to use a UMOP plan. It is not possible to specify how the environment agent will act and so in what manner it may interfere with the system agents. Therefore, a controller using a UMOP plan as part of its resolution mechanism cannot use the plan to respond to interference from another agent while the controlled agents are following its instructions, even though the generated universal plan may contain a trace for this specific set of environment agent actions. This means that the distinction in UMOP between system agents and environment agents does not appear to improve the implementation of a resolution mechanism for a centralised controller.

6.3.4 Other potential strategies

We are able to identify two further possible avenues of investigation that could allow a UMOP controller to handle a row obstruction interaction where there is interference from a third agent. We now describe these two additional strategies. From our experiments using UMOP, however, it appears that a number of factors contribute to making UMOP unsuitable for implementing the resolution mechanism of a controller and for handling the introduction of additional agents to the interaction. Therefore, we have not pursued the investigation of these additional strategies.

Instructing three agents

Either of the three-agent generated plans can potentially still be used as part of the resolution mechanism of the UMOP controller. The output of plan-simulation shows the generated plans being able to manage a row obstruction interaction where a third agent is close by and, if left to act independently, may interfere with the agents participating in the interaction. Therefore, the generated universal plans can be used to allow the UMOP controller to send instructions to all three agents in
order to resolve the interaction.

This strategy consists of agents monitoring their own actions for signs that an undesirable interaction is occurring, as well as the controller monitoring the actions of the agents. One implementation method is for the controller to scan the environment around an undesirable interaction for any agents that may interfere with the agents participating in the interaction. Alternatively, an agent that believes it is participating in an undesirable interaction can also monitor its immediate vicinity for any agents that may interfere with its actions when it has received instructions from the controller. The agent will inform the controller of any nearby agents when it invokes the controller.

Specifically for this example, when an agent believes that it is participating in a row obstruction interaction, the agent must also be able to identify whether there is a third agent beside it when it invokes the controller. If this is the case, then the agent will send a message to the UMOP controller containing its agent ID number, the ID number of the obstructing agent and also the ID number of the other nearby agent. The UMOP controller can use a three-agent universal plan to instruct all three agents how to act to resolve the row obstruction interaction. From the point of view of the third agent, the agent would unexpectedly receive instructions from the UMOP controller, despite the agent not having invoked the controller itself.

Unfortunately, this strategy is unlikely to help in practice to resolve undesirable interactions where other agents may interfere. Firstly, a collection of separate plans would be required in order to handle situations where different numbers of agents interfere, or may potentially interfere, with the actions of the interacting agents. We have shown in Table 6.1 that UMOP appears unable to generate a plan for the row obstruction domain containing four agents.

Secondly, an agent that is very close to an agent when it invokes the controller could potentially interfere with the ability of the controlled agent to follow instructions received from the controller. However, the strategy that has been described is unable to handle situations where another agent moves next to and interferes with a controlled agent after it has received instructions from the controller. This is exactly the situation that we were trying to handle using UMOP, where an agent interferes with an agent that is already being instructed how to act. Therefore, although this strategy may be used for situations where a third agent is already close to agents that invoke the controller, this does not appear to be a practical strategy for incorporating possible interference from an additional agent into the implementation of a resolution mechanism.

Non-deterministic agent actions

A final possibility remains for using UMOP to handle other agents in the environment that may interfere with the actions of the agents that invoke a controller. The
domain description language NADL has been designed to allow non-deterministic agent actions to be defined.

The existing move forward actions of the system agents, shown in Listing 6.2, could be defined such that they may either succeed and move the agent as intended, or fail and cause no change to the agent’s location. By defining a row obstruction domain description where the move forward actions of the system agents have non-deterministic effects, it may be possible to simulate a situation where an agent tries but fails to act as instructed by the UMOP controller.

The implementation of a row obstruction domain description containing non-deterministic actions for the system agents has not been investigated. Based on the results of our investigation of introducing a third agent into the row obstruction domain, however, it is likely that a non-deterministic domain description will also not allow the resolution mechanism of the UMOP controller to handle situations where the agent fails to be able to follow the instructions of the controller. This is because the non-deterministic effect of the system agents’ move actions will be specified during plan-simulation as part of the joint action tuple. Therefore, it will not be possible for the controller to respond to any actual interference from another agent while the system agents follow the instructions of the controller.
7 Local coordination of autonomous agents

The Sinatra examples presented in earlier chapters have allowed us to examine different strategies for using directives to manage agent interactions. The scenarios considered so far, however, have focussed on behaviours that are designed to allow agents to coexist in a shared environment. We now consider behaviours that are used to facilitate coordination between agents. Such behaviours are not merely designed to allow the agents to achieve their individual goals but also to allow the agents to work together to achieve a communal, team or global objective.

Coordination of multiple agents can be achieved using a centralised controller, examples of which have been described in Chapter 5 and Chapter 6. The global view of the system available to the centralised controller means that coordination, while often far from trivial, is a matter of issuing the correct commands to the individual agents. There are some situations, however, where the autonomy of the agents must be maintained and so coordination must be done at a local level. We investigate the use of directives for this purpose.

Directives are local rules that are complied with by each agent in the system individually. However, if a system of agents comply with a set of directives, can the directives be used to influence the individual agent actions such that collectively the agents exhibit a coherent global behaviour?

In this chapter we explore two simple illustrative examples. The first is a robot rugby domain, inspired by the RoboCup 2D Soccer Simulation League [Rob]; the second is a coordinated hide-and-seek domain, inspired by robotic search-and-rescue (e.g. CRASAR [Mur]).

7.1 Coordination through behaviours

Coordinating the actions of autonomous agents requires the coordination to be carried out locally at each agent in order for their autonomy to be maintained. Directives are a possible mechanism by which coordination of autonomous agents can be achieved. However, by definition, an autonomous agent acts independently and is unable to control the actions of the other autonomous agents. From an individual agent’s point of view, the other agents in the system act in an unknown manner.
A possible strategy for using directives to coordinate agents is to manage the unpredictable nature of the other agents. By imposing limitations on the actions of the agents, their actions become more standardised, allowing an agent to attempt to coordinate its own actions with those of the other agents in the system. We have identified two possible techniques for coordinating agent actions using this strategy.

In the first technique, directives are used to establish expected patterns of behaviour during situations when coordination is required. This allows an agent to determine its actions assuming a pre-established action will be performed by the other agents in the system.

For the second technique, agent communication is used to allow agents to exchange information. Directives specify when the agents should communicate and the information that should be exchanged. This allows an agent to have a greater understanding of the past, current and future actions of the other agents and to modify its own actions based on this information.

In the following experiments both of these techniques for coordinating agents are presented. To implement the team formations used in the robot rugby domain, expected patterns of behaviour are used to allow the agents to attack and to defend together. The ability of the team to make a specific formation successfully relies on the other agents in the team also attempting to make the same formation. To implement the coordinated search used in the hide-and-seek domain, message-exchange between agents is used to allow the searching agents to search the grid environment more efficiently.

7.2 Example: Using behaviours to coordinate agents playing robot rugby

The aim of this example is to demonstrate the use of directives that allow agents to coordinate their actions by creating expected patterns of behaviour amongst the group of agents. To do this we use the example of a team of agents playing a very simplified form of a robot rugby game. We develop a general rugby behaviour that will be used by agents to play the game. We then develop the formation behaviour, a modified version of the rugby behaviour, which can also be used to coordinate the actions of the agents within a team.

Using these behaviours, we are able to investigate the effect of expected patterns of behaviour on coordinating agent actions. We compare simulated robot rugby games where the two teams are using different behaviours - one team will use the rugby behaviour, while the other team will use the formation behaviour.

In this example, it is our intention to develop directives for coordinating agents and to determine whether these directives allow the agents to perform better than a team of uncoordinated agents. It is not our intention to develop or make claims
for the game strategy employed by either team.

### 7.2.1 Implementing the robot rugby agents

The concept of the robot rugby domain is based on the RoboCup Soccer Simulation League.

> “This is one of the oldest leagues in RoboCupSoccer. The Simulation League focus on artificial intelligence and team strategy. Independently moving software players (agents) play soccer on a virtual field inside a computer” [Rob].

The 2D Soccer Simulation League has the greatest similarity to the multi-agent domains that are simulated within the Sinatra test bed. The design of the robot rugby domain also draws inspiration from the UMOP soccer domain [JV00], described in Section 6.1.

It should be noted that the robot rugby domain is so called in order to echo the name of the RoboCup Soccer Simulation League, as well as to highlight that the game implementation is a simplification of a rugby game scenario. It is not intended that the robot rugby domain should attempt to simulate a realistic human rugby game; instead, the domain should imitate certain aspects of the game. Similarly, we do not consider how the actions available to the rugby robots may be implemented in practice, although these capabilities are based on those possessed by physical robots that take part in the RoboCup competition. For example, we do not consider how a physical robot may be able to ‘tackle’ another player.

The robot rugby domain consists of a grid environment populated by two teams of agents and a ball object. The leftmost and rightmost columns of the grid environment represent the goal-lines. Both teams of agents attempt to score by carrying the ball over their opponent’s goal-line. Agents are capable of passing the ball, tackling a ball-carrier and attempting to block an opponent. In addition, the agents must handle non-deterministic effects of actions while playing the game, such as dropping the ball when attempting to pass or tackle. We now describe the implementation of the robot rugby domain, focussing on the modifications required to the Sinatra agents and the development of the different behaviours.

#### Robot rugby agent perceptions

We are supposing that the Sinatra agents are able to perceive their location \( (x, y) \) position within the grid environment. To respond to the game that is happening the Sinatra agent must be extended to allow the agent to perceive information about the ball.
First, the agent must be able to perceive when it is carrying the ball. This can be used, together with the agent’s current location, to allow the agent to identify when it has scored a goal. For situations when an agent is not carrying the ball, the agent must be able to perceive the location of the ball, whether the ball is being carried and the team that has possession of the ball. Using these perception methods, the agent is able to identify situations when the ball has been dropped, as well as modify its actions depending on whether another member of the agent’s team has the ball (the agent is attacking) or whether the opposing team has the ball (the agent is defending).

We are also supposing that a Sinatra agent is able to perceive the immediately adjacent grid locations that it is able to move to, so that the agent can determine whether these locations are occupied. For the original Sinatra agents, the agents are only able to move north, east, south or west and so the agent is also only able to perceive these four locations. The movement capabilities of the Sinatra agents are extended for the robot rugby domain to allow the agents to move diagonally. Therefore, the agents’ sensory capabilities are similarly increased to encompass the eight grid cells that surround the agent.

For use when other agents move into these adjacent locations, the perception methods of the agent are extended further in order to determine useful information about an adjacent agent, such as its team. This allows an agent to be able to identify opportunities to interact with the adjacent agent; for example, for a ball-carrier to pass the ball to a teammate or for a defending agent to tackle the ball-carrier of the opposite team. In addition, the Sinatra agent is extended to be able to perceive the location of the nearest opposing team agent and the location of its nearest teammate, even if these agents are not immediately adjacent to the agent.

Table 7.1 describes the additional agent perception methods that are introduced to the Sinatra agents for the robot rugby domain. Higher level perception methods are achieved by combining the results of other perceptions. While these perception methods are significantly more advanced than those defined for the original Sinatra agents, these methods implement realistic capabilities for a physical soccer playing robot, such as those used in RoboCup [Rob].

The additional information that the agent can perceive about the game is intended to provide the agent with enough information to respond to the game that is happening, without providing each agent with a global view of the environment. The agent is aware of the ball and, therefore, whether it should be currently attacking or defending. The agent is aware of its nearest opponent, which represents the closest threat to a ball-carrying agent or an opportunity to ‘mark’ a member of the opposing team. The agent is also aware of opportunities to pass the ball to a teammate and to tackle an opposing ball-carrier in order to imitate the game-play of an actual rugby game.
<table>
<thead>
<tr>
<th>Perception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasBall</td>
<td>True if the agent is holding the ball object.</td>
</tr>
<tr>
<td>scored</td>
<td>True if hasBall is true and the agent is in the first or last column of the grid environment (depending on the agent’s team). Uses the agent’s current y position to perceive whether the agent is in the appropriate column.</td>
</tr>
<tr>
<td>teamBall and oppositionBall</td>
<td>True if a teammate of the agent is holding the ball or an opponent is holding the ball respectively.</td>
</tr>
<tr>
<td>ballDropped</td>
<td>True if hasBall, teamBall and oppositionBall are all false.</td>
</tr>
<tr>
<td>ballLocation</td>
<td>Returns the x, y position of the ball object within the grid environment, regardless of whether the ball has been dropped or is being carried.</td>
</tr>
<tr>
<td>ballCell</td>
<td>True if the agent is in the same grid location as a ball that has been dropped. Uses the ballLocation and the agent’s current x, y position perceptions to identify when an agent is able to pick up the ball.</td>
</tr>
<tr>
<td>north, east, south, west, ne, nw, se and sw</td>
<td>Allows the agent to perceive the eight adjacent grid locations to its current position. The agent can determine if a location is empty or occupied and the team of an adjacent agent.</td>
</tr>
<tr>
<td>canPass</td>
<td>Uses the adjacent grid location perceptions to identify opportunities for a ball-carrier to pass the ball to a teammate. The ball-carrier agent may only pass the ball to an adjacent agent that is beside or diagonally behind the ball-carrier.</td>
</tr>
<tr>
<td>canTackle</td>
<td>Uses the adjacent grid location perceptions, combined with the oppositionBall and ballLocation perceptions, to identify opportunities for an opposing team agent to tackle the ball-carrier.</td>
</tr>
<tr>
<td>closeOpponent</td>
<td>Uses the adjacent grid location perceptions to identify when an agent is next to an opposing team agent.</td>
</tr>
<tr>
<td>nearestOpponent</td>
<td>Returns the x, y position of the nearest opponent to the agent. Used to identify the nearest opposing team agent when closeOpponent is false.</td>
</tr>
<tr>
<td>nearestTeammate</td>
<td>Returns the x, y position of the nearest teammate of the agent. Used during the implementation of the formation behaviour but included now for completeness.</td>
</tr>
</tbody>
</table>

Table 7.1: The perception methods for the robot rugby domain.

**Robot rugby agent actions**

To facilitate the free movement of the robot rugby agents within the grid environment, the original Sinatra agent is extended to allow the agents to be able to turn and move during the same time step. In addition, the agents are now able to move diagonally from their current location, as well as north, east, south or west. Therefore, in a single time step, the Sinatra agents are now able to move to any of the eight adjacent locations from its current position.
These modifications are achieved by creating high-level movement actions that will be decomposed by the agents during their execution into the \texttt{turnLeft}, \texttt{turnRight} and \texttt{moveForwards} action methods that are already part of the Sinatra agents. The turn and move forwards actions are the basic movement actions implemented in the Sinatra agent implementation. In order to achieve the high-level movements using these existing action methods, the turn actions must also be modified to allow the agent to move diagonally.

As with the original turn and movement actions of the Sinatra agent, the enhanced movement capabilities of the robot rugby domain are actions that will always succeed as long as the location that the agent is moving to is unoccupied. If two agents both attempt to move into the same location during the same time step, then only one of them will succeed. The other agent will remain in its original location.

In order to imitate the game-play of an actual rugby game, a number of additional action methods are added to the Sinatra agents. These actions include specific rugby moves, as well as additional high-level movement actions. These high-level actions allow the agents to move towards a specific goal or away from an opposing team agent. By design, some of the additional actions have an inbuilt chance of failure, making the agents’ actions non-deterministic. This introduces an element of chance into the robot rugby game.

Table 7.1 describes the additional agent action methods that are introduced for the robot rugby domain. Agents can perform one of the actions described in Table 7.1 per time step of the simulation. The specific rugby actions allow the agent to play the game, by passing the ball and tackling other players. The new movement actions allow the robot rugby agents to move around the environment more freely, which allows a greater number of interactions to occur between the agents.

While we do not consider how these action methods will be implemented in practice, we believe that these action methods correspond to actual implementable actions for a simple physical robot. For the physical RoboCup Soccer leagues, techniques for self-localisation, perception of the ball and perception of other players are combined to allow similar capabilities to be implement for soccer playing robots [Rob].

**Robot rugby behaviour**

The agents use the relatively simple rugby behaviour in order to play the game. In general, the rugby behaviour instructs the ball-carrier agent to move towards the opposing team’s goal-line (\texttt{moveToGoal}). If the ball-carrier meets an opposing team agent, it will attempt to \texttt{dodge} past the agent or to \texttt{pass} the ball if there is a teammate nearby. If another member of an agent’s team is carrying the ball, then the agent attempts to move towards (\texttt{moveToOpponent}) and then try to \texttt{block} the nearest opposing team agent. If the opposing team has the ball, then the agent also
<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
<th>Possible outcomes</th>
</tr>
</thead>
<tbody>
<tr>
<td>moveNorth, moveEast, moveSouth, moveWest, moveNE, moveNW, moveSE and moveSW</td>
<td>Directs the agent to turn to face the appropriate direction and to move forwards into this location. Used by the moveToLocation and dodge methods described below.</td>
<td>Movement will succeed unless the grid location becomes occupied by another agent.</td>
</tr>
<tr>
<td>moveToLocation</td>
<td>Directs the agent to move directly towards a specified grid location. Used by the more specific moveTo* methods, as well as by the formation behaviour.</td>
<td>Movement will succeed unless the grid location becomes occupied by another agent.</td>
</tr>
<tr>
<td>moveToBall</td>
<td>Directs the agent to move directly towards the ball’s location. This is used when the ball has been dropped.</td>
<td>Movement will succeed unless the grid location becomes occupied by another agent.</td>
</tr>
<tr>
<td>moveToGoal</td>
<td>Directs the agent to move directly towards the opposing team’s goal-line. This is used by the ball-carrier agent.</td>
<td>Movement will succeed unless the grid location becomes occupied by another agent.</td>
</tr>
<tr>
<td>moveToOpponent</td>
<td>Directs the agent to move directly towards the nearest opposing team member.</td>
<td>Movement will succeed unless the grid location becomes occupied by another agent.</td>
</tr>
<tr>
<td>pickUpBall</td>
<td>When the ball is dropped it is moved to the nearest unoccupied grid location. An agent that moves into the same grid location as the ball can pick up the ball.</td>
<td>Always succeeds.</td>
</tr>
<tr>
<td>pass</td>
<td>The ball-carrier agent can attempt to pass the ball to a teammate that is in a grid location immediately to the side or diagonally behind the agent.</td>
<td>1/3 chance the ball will be dropped, otherwise pass is successful.</td>
</tr>
<tr>
<td>dodge</td>
<td>An agent can attempt to move around an opposing team agent. This is used by the ball-carrier agent.</td>
<td>Movement will succeed unless the grid location becomes occupied by another agent.</td>
</tr>
<tr>
<td>tackle</td>
<td>An opposing team agent can attempt to tackle the ball-carrier if it is in an adjacent grid location.</td>
<td>Equal chance that the tackle is successful, the tackle is unsuccessful or the ball will be dropped.</td>
</tr>
</tbody>
</table>

Table 7.2: (part 1) The action methods for the robot rugby domain.
An agent can attempt to hinder the progress of an opposing team agent. Movement will succeed unless the grid location becomes occupied by another agent.

<table>
<thead>
<tr>
<th>block</th>
<th>An agent can attempt to hinder the progress of an opposing team agent.</th>
<th>Movement will succeed unless the grid location becomes occupied by another agent.</th>
</tr>
</thead>
<tbody>
<tr>
<td>celebrate</td>
<td>Used to signify that a goal has been scored. This is used by the ball-carrier agent when it has crossed the goal-line.</td>
<td>Always succeeds.</td>
</tr>
</tbody>
</table>

Table 7.1: (part 2) The action methods for the robot rugby domain.

Attempts to block the nearest opposing team agent, as well as to try to tackle the ball-carrier. If the ball is dropped, then all of the agents will try to move to the ball location (moveToBall) and pick up the ball (pickUpBall).

Listing 7.1 shows the rugby behaviour input file. The behaviour requires relatively few directives in order to be implemented. This is because of the high-level actions that are prescribed. Relatively few constraints are required because very few of the directives can be triggered in the same agent state. The first aggregate reflects this property of the possible agent states.

A snapshot of the robot rugby domain implemented in Sinatra is shown in Figure 7.1. The ball is shown in the Sinatra GUI by a small white square. This is a start configuration of a game, showing the red ball-carrier starting behind its own team’s goal-line and the members of both teams in their respective halves of the environment. A point will be scored by red if a member of red team can hold the ball behind the blue goal-line, and conversely for blue.

![Figure 7.1: Two teams in the robot rugby domain.](image)

The rugby behaviour allows the agents to play the game but does not allow the agents within a team to coordinate their actions. A centralised controller is not appropriate for this type of domain as we are attempting to represent some situations that may appear in a rugby game in practice. We propose that expected patterns of behaviour can be used to allow the agents to coordinate their actions. We investigate
celebrate :- scored.

pass :- hasball, opponentclose, canpass.
dodge :- hasball, opponentclose.
togoal :- hasball.

block :- teamball, opponentclose.
toopponent :- teamball.

tackle :- oppositionball, cantackle.
block :- oppositionball, opponentclose.
toopponent :- oppositionball.

toball :- balldropped.
pickup :- ballcell.

:- pass, dodge.
:- pass, togoal.
:- dodge, togoal.
:- tackle, block.
:- tackle, toopponent.
:- block, toopponent.

s: 1 {scored, hasball, teamball, oppositionball, balldropped, ballcell} 1.
s: 0 {opponentclose} 1.
s: 0 {canpass, cantackle} 1.

in: scored hasball teamball oppositionball balldropped ballcell opponentclose canpass cantackle.

out: celebrate toball togoal toopponent pickup pass dodge tackle block.

Listing 7.1: The rugby behaviour input file.

direct this by developing the formation behaviour, which attempts to direct the agents to arrange themselves into different formations depending on the circumstances of the game.

7.2.2 Implementing the formation behaviour

The formation behaviour attempts to allow a team of agents to coordinate their actions in order to attack and defend more successfully during the game of robot rugby. The formation behaviour is a modified version of the rugby behaviour, which directs the agents to use additional behaviours and high-level actions to move into the different formations. We begin by defining the different formations that will be used and describing how they are implemented, before we define the formation
behaviour itself.

**Wedge formation**

When a team of agents are attacking, one of the agents will be the ball-carrier, who will attempt to take the ball to the opponent’s goal-line as quickly as possible. For the formation behaviour, the other agents in the team attempt to arrange themselves into a wedge shape behind the ball-carrier and to maintain this formation while the ball-carrier moves towards the goal-line. Figure 7.2 shows the blue team in a wedge shape formation.

![Figure 7.2: The blue team in a wedge formation.](image)

The wedge formation is used when agents are attacking, which the agents are able to perceive using the teamBall perception method. Therefore, the directive for the agents to attempt to form the wedge formation can be written as

\[ \text{wedge} \leftarrow \text{teamball} \]

The token teamball represents that teamBall is true and the token wedge represents the wedge formation behaviour or an equivalent action method implementation. We described in Section 3.5.4 how behaviours can be implemented in terms of other behaviours or as high-level action methods. Whether the wedge formation is implemented as a behaviour or as an action method depends on the choice of the system designer.

For this example, we demonstrate the implementation of the wedge formation using a behaviour, shown in Listing 7.2. The associated dictionary of tokens is shown in Table 7.2.

The agents following the wedge formation behaviour will attempt to move into the wedge formation and then to maintain this formation as the ball-carrier progresses towards the goal-line. By forming a wedge shape, the attacking team are all in close proximity to the ball-carrier. This allows the ball-carrier’s teammates to support the ball-carrier if it meets an opposing team agent, by making it harder for the opposing team agent to get close enough to tackle the ball-carrier.
tonw :- red, northball.
tosw :- red.
tone :- blue, northball.
tose :- blue.
:- tonw, tosw.
:- tone, tose.

s: 1 {red, blue} 1.
s: 0 {northball} 1.
in: red blue northball.
out: tonw tosw tone tose.

Listing 7.2: The wedge formation behaviour input file.

<table>
<thead>
<tr>
<th>Token</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>red and blue</td>
<td>The agent’s team, which is stored as a fact in the agent’s internal memory.</td>
</tr>
<tr>
<td>northball</td>
<td>The result of comparing the y coordinates returned by the agent’s location and the ballLocation perception methods. Specifies whether the agent is to the north of the ball.</td>
</tr>
<tr>
<td>tonw, tosw,</td>
<td>The moveToLocation action method. However, before calling this method, the target grid location must be identified based on the result of the ballLocation method. The target locations are north-west, south-west, north-east and south-east of the ball-carrier respectively.</td>
</tr>
<tr>
<td>tone and tose</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.2: The dictionary of tokens for the wedge formation behaviour.

The wedge shape also means that the ball-carrier has the opportunity to pass the ball to one of its teammates, as they will be diagonally behind the ball-carrier. If the ball is successfully passed to a teammate, then the other attacking agents will now be directed to form a wedge shape behind the new ball-carrier.

**Line formation**

When a team of agents are defending, the opposing team ball-carrier will attempt to reach the defending team’s goal-line as fast as possible (moveToGoal), before the defending team can reach and tackle the ball-carrier. For the line formation, the defending team of agents attempt to arrange themselves into a line in front of their goal-line and to maintain this formation until the ball-carrier moves into a position where it can be tackled. Figure 7.3 shows the blue team in a line formation.
The line formation is used by agents that are defending, which the agents are able to perceive using the oppositionBall perception method. Therefore, the directive for the agents to attempt to form the line formation can be written as

\[
\text{line } \leftarrow \text{oppositionball}
\]

The token oppositionball represents that oppositionBall is true and the token line represents the line formation behaviour or an equivalent action method implementation.

For this example, we demonstrate the implementation of the line formation by describing the formLine action method. In this action method, the agents use the \( x \) coordinate returned by the nearestTeammate method and the \( y \) coordinate of their current location as the target \( x, y \) coordinates for the moveToLocation method.

Unlike the wedge formation, having achieved the line formation, the agents are not required to move further while maintaining this formation. Therefore, the agents maintain the line formation until they perceive an opposing team agent to block or tackle.

The line formation improves the ability of the team of defending agents to monitor the grid environment for the ball-carrier. If the ball-carrier manages to slip past the defending agents, then it is very difficult for the defending team to prevent the ball-carrier from scoring. Therefore, the line formation allows the defending team to observe the majority of a column of the grid environment and so make it more likely that they will be able to spot the ball-carrier.

**Formation behaviour**

The formation behaviour modifies the rugby behaviour to direct the agents to attempt to move and to maintain the wedge formation and line formation when they are attacking and defending respectively. Listing 7.3 shows the necessary additions to the rugby behaviour (highlighted in bold). The token wedge represents the wedge formation behaviour, while the token line represents the formLine action method.
celebrate :- scored.

pass :- hasball, opponentclose, canpass.
dodge :- hasball, opponentclose.
togoal :- hasball.

**wedge :- teamball.**

tackle :- oppositionball, cantackle.
block :- oppositionball, opponentclose.
**line :- oppositionball.**

toball :- balldropped.
pickup :- ballcell.

:- pass, dodge.
:- pass, togoal.
:- dodge, togoal.
:- tackle, block.
:- **tackle, line.**
:- **block, line.**

s: 1 \{scored, hasball, teamball, oppositionball, balldropped, ballcell\} 1.
s: 0 \{opponentclose\} 1.
s: 0 \{canpass, cantackle\} 1.

in: scored hasball teamball oppositionball balldropped ballcell opponentclose canpass cantackle.

out: celebrate toball togoal pickup pass dodge tackle block **wedge line.**

Listing 7.3: The formation behaviour input file.

### 7.2.3 Observations and results

Figure 7.2 and Figure 7.3 show different formations resulting from when agents follow the formation behaviour. The Sinatra GUI also shows the agents maintain the wedge formation as the ball-carrier agent moves towards the opponent’s goal-line. Therefore, the agents are able to coordinate their actions in order to achieve and then to maintain the two formations that have been implemented.

For interest, repeated simulation of the robot rugby domain was used to determine whether the ability to coordinate their actions offers any advantages to the team of agents. Each game of robot rugby was timed to last for 300 time steps (approximately 5 minutes). The game starts with a randomly selected team of agents having possession of the ball. The ball-carrier must start behind its team’s goal-line,
while the other attacking and defending agents are randomly positioned within their halves of the grid.

When a ball-carrier agent manages to score by crossing the opposing team’s goal-line, a point is awarded to the ball-carrier’s team. The agents are then reset to their halves of the grid for the start of a new round. The opposing team is given possession of the ball and so a member of the opposing team will become the ball-carrier and will start the new round behind its goal-line. At the end of 300 time steps the simulation is stopped and the scores of the two teams are recorded. The scores are classified as a red win, a blue win or a draw.

Two sets of repeated simulations of the robot rugby domain were carried out. In the first set of games (R1), both the red team and the blue team use the rugby behaviour. This is the control group, which aims to establish that neither team has an advantage when both teams use the same behaviour to determine their actions. In the second set of games (R2), the red team use the rugby behaviour, while the blue team use the formation behaviour. There were 150 games played for each set, the results of which are summarised in Table 7.3.

<table>
<thead>
<tr>
<th>Game set</th>
<th>Red win</th>
<th>Blue win</th>
<th>Draw</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>54</td>
<td>62</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>36.0%</td>
<td>41.3%</td>
<td>22.7%</td>
</tr>
<tr>
<td>R2</td>
<td>20</td>
<td>104</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>13.3%</td>
<td>69.3%</td>
<td>17.3%</td>
</tr>
</tbody>
</table>

Table 7.3: Overall results of the robot rugby simulations, showing the actual and percentage distribution of game results.

The results show that when both teams of agents use the rugby behaviour there is a similar distribution between game wins for the red team and the blue team, with some games ending in a draw. When the blue team switches to using the formation behaviour, there is a dramatic increase in the number of game wins for the blue team, with the red team winning significantly fewer games. Therefore, it appears that the coordination of the agent actions due to the formation behaviour provides a real advantage to the blue team in the robot rugby domain, allowing them to achieve the team objective of winning the robot rugby game more easily.

7.2.4 Summary

Using the example of the robot rugby domain we have demonstrated how directives can be used to coordinate agent actions by establishing expected patterns of behaviour amongst the agents. The formation behaviour allows agents to form and to maintain certain shapes by directing the agents under the assumption that the other agents will act in a predictable manner.
The formation behaviour attempted to achieve the wedge formation by instructing an attacking agent to move to a diagonal position behind the ball-carrier, implemented in the wedge formation behaviour. If the other attacking agent also acts as directed by the wedge formation behaviour, then it is possible for the team of attacking agents to achieve the wedge formation.

The formation behaviour attempted to achieve the line formation by instructing all of the defending agents to move to the same column, implemented in the formLine action method. If all of the other defending agents also execute the formLine action, then it is possible for the team of defending agents to achieve the line formation.

As shown in Figure 7.2 and Figure 7.3, agents using the formation behaviour are able to coordinate their actions in order to form the different formations. However, the agents are not always able to achieve these formations. In most cases, this is due to the dynamic nature of the robot rugby domain and the physical restriction that only one agent can occupy a grid location, which can mean that the agents do not always act as directed.

The wedge formation behaviour may fail to allow the agents to achieve the wedge formation if both of the agents are on the same side of the ball-carrier agent. For example, if both agents are to the north of the ball-carrier, then the agents will both aim to move to the north diagonal position. Therefore, the wedge formation behaviour does not implement a guaranteed strategy for the agents to achieve the wedge formation.

More significantly, however, interference from the other team may also prevent the attacking agents from being able to achieve the wedge formation. If an opposing team agent is in the grid location diagonally behind the ball-carrier, then this will prevent an attacking agent from being able to occupy this location. Similarly, as the attacking agents move in the wedge formation, an opposing team agent may move to occupy a grid location diagonally behind the ball-carrier.

7.3 Example: Using behaviours to coordinate hide-and-seek agents

The aim of this example is to illustrate the use of directives that allow agents to coordinate their actions using some form of agent communication. To do this we use the example of communication between two seeking agents who are attempting to find a third hiding agent. We develop seeking agents that use a simple independent search behaviour to search the grid environment in a systematic manner, utilising an A* search algorithm. We then demonstrate a coordinated search behaviour, where the agents communicate information about their current search.

Using these behaviours, we are able to investigate the effect of agent communication on coordinating agent actions. We compare repeated simulations of the
hide-and-seek domain, where the seeking agents either search independently or using the **coordinated search** behaviour.

In this example, it is our intention to develop a behaviour for coordinating agents and to determine whether the behaviour allows the agents to achieve a global objective better than a team of uncoordinated agents. It is not our intention to develop or make claims for the specific search strategy employed by the seeking agents.

### 7.3.1 Implementing the seeking agents

The concept of the hide-and-seek domain is based on an idealised form of robotic search-and-rescue, where autonomous or manually controlled robots are used to aid rescue workers after disasters. The hide-and-seek domain is a simple version of a search-and-rescue scenario, where two searching agents attempt to find another agent as quickly as possible.

The hide-and-seek domain consists of a grid environment populated by two seeking agents and one or more other agents, one of whom is the hiding agent. The seeking agents use the **independent search** behaviour to search the environment. The other agents in the simulation, including the hiding agent, follow the **simple goal-directed navigation** to reach goal locations within the environment. The hiding agent is selected at random by the simulation from the agents following the **simple goal-directed navigation**. It should be noted that the hiding agent in this example is not deliberately attempting to hide from the seeking agents. We now describe the implementation of the hide-and-seek domain, focussing on the modifications required to the Sinatra agents and the development of the different behaviours.

**Seeking agent search strategy**

The seeking agents are based on the original Sinatra agent model. Therefore, the seeking agents cannot move diagonally and are only able to perceive the adjacent grid locations to the north, east, south and west. This means that the seeking agents are only able to perceive the hiding agent when it is in one of these four grid locations. A search strategy must be used in order for the seeking agents to be able to find the hiding agent.

The seeking agents divide the grid environment into different zones. In this example, the hide-and-seek domain is implemented in Sinatra as a 9x9 grid. The seeking agents divide the environment into nine zones, each consisting of a 3x3 block of cells. The centre cell of each zone is used as a **zone marker** and stored in the agent’s internal memory in order to record the location of each zone.

A time stamp system is used by the seeking agents to select which zone to search. As well as storing the nine zone markers in its internal memory, the agent also associates a time stamp with each zone marker. The time stamp records the simulation
time when the agent last visited this zone marker. The seeking agent selects the zone with the smallest time stamp when determining which zone to search next.

Having identified a zone to search next, this zone marker is set as the agent’s goal. An A* search method is used to find a path for the seeking agent to the selected zone marker. The A* search method returns a sequence of actions that will direct the agent to follow the path found by the algorithm. Having reached a zone, the seeking agent follows a basic pre-programmed search pattern to ‘search’ the zone for the hiding agent. Once the zone has been searched, a new zone marker is selected and this process is repeated.

The seeking agents are able to find the hiding agent at any point, whether travelling between zones or while searching a particular zone. When one of the seeking agents has found the hiding agent, the simulation can be stopped and the time taken to find the hiding agent recorded, or a different hiding agent can be selected and the seeking agents begin a new search.

The A* search method and the pre-programmed search pattern are implemented as the `aStarSearch` and `searchZone` methods respectively. Both of these methods return a sequence of tokens that represent actions for the agent to perform. These sequences consist of the actions to turn 90° to the left or right and the `moveForwards` action. As the agents will need to follow this sequence of actions over a number of time steps, the tokens are stored in the agent’s internal memory. The same strategy described for storing a sequence of tokens received from a controller as part of the centralised control mechanism (Section 5.2.1) can be used to store this sequence of tokens.

When implementing the centralised control mechanism in Section 5.2.1, a directive was added to the agent’s behaviour-switch to direct the agent to follow a stored list of instructions. For the hide-and-seek domain, however, a directive is added to the independent search behaviour to instruct the agent to follow a sequence of tokens stored in memory.

\[
\text{followinstructions} \leftarrow \text{storedinstructions}
\]

This directive is added to the independent search behaviour, rather than to the master behaviour-switch, as this is helpful when we come to implement the coordinated search behaviour.

The `actionSwitch` method uses the same process as described for following the instructions received from a controller to allow the agent to follow the series of tokens stored in its memory. The `actionSwitch` method selects the first token (or set of tokens) stored in the agent’s internal memory and executes the action method associated with this token. Having identified the required action method, the `actionSwitch` method also deletes this token from the agent’s memory. This
moves the next token in the sequence to the front of the list, so that it will be used to identify the action to be performed in the next time step.

**Implementing the A* search**

The seeking agent uses the `aStarSearch` method to find a path from the agent’s current location to a particular zone marker, based on an A* search. The `aStarSearch` method returns a list of actions that direct the agent to follow this path.

The A* search algorithm is a best-first search algorithm, using an heuristic to find the lowest cost path between an initial location and a goal location. The heuristic has the form

\[ f(x) = g(x) + h(x) \]

where \( g(x) \) is the lowest cost to reach the current location \( x \) and \( h(x) \) is the estimated cost from the current location to the goal location. The value \( f(x) \) is the estimated cost of the cheapest solution that passes through location \( x \).

If the heuristic \( h(x) \) is admissible, meaning that it does not overestimate the cost to reach the goal, then A* search is guaranteed to find the optimal, smallest-cost path to the goal location. Also if the heuristic \( h(x) \) is consistent, meaning that it also satisfies the condition

\[ h(x) \leq d(x, y) + h(y) \]

where \( d(x, y) \) is the cost of travelling between adjacent locations \( x \) and \( y \), then the A* algorithm can be implemented in a more efficient manner, by recording a list of locations already visited by the algorithm.

The heuristic used in the `aStarSearch` method is the Manhattan distance between grid locations. Agents can only travel north, east, south or west from a given location; therefore, the heuristic counts a cost of one to move to each of these locations. The heuristic counts a cost of two to move from the agent’s current location to an adjacent diagonal location, as two movement steps will have to be taken. The heuristic does not include the additional cost of turning that the Sinatra agents may require in order to move between locations.

To carry out an A* search, two sets of locations are maintained, referred to as the *open set* and the *closed set*. The open set contains the set of locations to be traversed, initially containing only the start location. The closed set contains the set of locations that have already been traversed and so do not need to be considered again. The use of a closed set is only possible when the heuristic \( h(x) \) is consistent, as is the case for Manhattan distance, allowing the implementation of the algorithm to be more efficient.
For each location $x$ stored in the open set and the closed set, the values of $f(x)$, $g(x)$ and $h(x)$ currently associated with the location are also stored. These values are modified as the algorithm progresses and finds different, possibly shorter paths for reaching the same location. A *parent location* is also associated with each location stored in the open set and the closed set. The parent location is a record of the previous location along the path that was used to reach this location.

Listing 7.4 shows the steps that are carried out during each iteration of the algorithm. The A* search algorithm is guaranteed to find the shortest path to the goal location if one exists. If the open set becomes empty and the goal location has not been found, then there is no path to the goal location and the algorithm terminates with a failure.

1. The location $x$ with the lowest $f(x)$ value is selected from the open set.
   - If $x$ is the goal location, then the algorithm terminates and the path to the goal is returned.
   - Otherwise, the location $x$ is removed from the open set and added to the closed set.

2. For each location $y$ adjacent to $x$
   - If $y$ is in the closed set, then it can be ignored.
   - Otherwise
     a) If $y$ is not already in the open set, then add $y$. Set the value of $g(y)$ stored by $y$ to some maximum value (Integer.MAX_VALUE) and set the parent location of $y$ to be $x$.
     b) Calculate the $g(y)$ value. If the value $g(y)$ is less than the currently stored value, then update the values of $g$ and $f$ stored by $y$ and update the parent location of $y$ to be $x$.

Listing 7.4: The A* search algorithm.

When a goal location is found, the record of parent locations is used to reconstruct the path from the start location to the goal. Therefore, the path is a list of grid locations that the agent can move between in order to reach the goal location. This list is translated by the `aStarSearch` method into a list of tokens that represent the action methods to turn and move the agent between these locations. Using these actions, the agent is able to follow the path found by the A* search algorithm to the selected zone marker.

**Seeking agent search behaviour**

The seeking agents use the *independent search* behaviour to search the grid environment for the hiding agent. In general, the *independent search* behaviour instructs a seeking agent to find a path from its current location to a selected zone marker.
The seeking agent then follows the sequence of actions returned by the `aStarSearch` method. When the agent reaches the zone marker, the `independent search` behaviour instructs the agent to search the zone using the `searchZone` method. Having completed this search, the agent selects a new zone marker to be its goal location and repeats this process.

Listing 7.5 shows the `independent search` behaviour input file. The behaviour is very simple because of the high-level actions that are contained in the directives. Constraint formulae for impossible concurrent actions are not required because the aggregate defining the possible states specifies that the different state facts cannot be true at the same time. However, it is possible for the agent to find itself in a situation where `newzone` and `atzone` hold at the same time, meaning that a new zone is selected and the agent is already at the appropriate zone marker. This situation can occur at the very start of the simulation. Therefore, the implementation of the `currentState` method for the seeking agent must ensure that `atzone` is checked before adding `newzone` to the agent’s current state.

```
astarsearch :- newzone.
searchzone :- atzone.
followinstructions :- storedinstructions.
selectzone :- searchedzone.

s: 1 {newzone, atzone, storedinstructions, searchedzone} 1.
in: newzone atzone storedinstructions searchzone.
out: astarsearch searchzone followinstructions selectzone.
```

Listing 7.5: The `independent search` behaviour input file.

The `independent search` behaviour is only designed to direct the agents how to search the grid environment. The behaviour does not handle the actions of the agent when one of the seeking agents finds the hiding agent. Instead, the seeking agents constantly monitor the environment for the hiding agent as they act. When the hiding agent has been located, the simulation can select a new hiding agent from the agents following the `simple goal-directed navigation` and the seeking agents can begin a new search by setting all of the time stamps for the zone markers to 0. This allows the hide-and-seek simulation to be run in a continuous manner. Alternatively, the simulation can be stopped and the time taken for the seeking agents to find the hiding agent recorded.

A snapshot of the hide-and-seek domain implemented in Sinatra is shown in Figure 7.4. The red and blue agents are the seeking agents, following the `independent search` behaviour. The green agent is the hiding agent, following the `simple goal-directed navigation` behaviour.
directed navigation. The highlighted squares around the red and blue agents show the area of the grid that they are able to perceive.

Figure 7.4: Three agents in the hide-and-seek domain. The red and blue agents are the seeking agents trying to find the green hiding agent.

The independent search behaviour allows the seeking agents to search the grid environment but does not allow the seeking agents to coordinate their actions. We now consider how agent communication can be used to allow the agents to coordinate their actions. We investigate this by developing the coordinated search behaviour, which attempts to allow the seeking agents to search together by communication information about their individual search to each other.

7.3.2 Implementing the coordinated search behaviour

The coordinated search behaviour attempts to allow the seeking agents to coordinate their actions in order to search the grid environment for the hiding agent more efficiently. By directing agents to share information about their current search, agents do not spend time searching an area that was recently searched by the other seeking agent. The coordinated search behaviour is an extension of the independent search behaviour, which introduces additional actions to be performed concurrently while the agent follows the existing directives of the independent search behaviour. These additional actions require the seeking agents to be able to communicate. We begin by describing the communication capabilities required by the seeking agents to allow them to send messages to each other and the type of information the agents will share. We then define the coordinated search behaviour itself.
Seeking agent communication

By extending the seeking agents to be able to send messages to each other, the seeking agents will be able to share information about their current search. This allows a seeking agent to have a greater understanding of the current and future actions of the other seeking agent. The seeking agents take this information into account when determining their own actions.

To implement this coordination the seeking agents must be aware of each other, with the ability to send and receive messages. We reuse the simple message passing interface used to allow agents and controllers to communicate as part of the centralised control mechanism (Section 5.2.1). When implementing this communication mechanism, the agent ID number was used to allow the controller to send messages to an agent. Therefore, the seeking agents must know the ID number of the other seeking agent.

Using an agent’s ID number, another agent is able to send text-based messages to this agent. The information that the seeking agents will share are the time stamps associated with each zone marker recorded in their internal memory. These time stamps form the search log of the agent, being a record of the last time that the seeking agent searched these zones.

We require that the agents store the zone markers in their internal memory in a fixed order. For example, the pair-wise ascending order of the $x$, $y$ coordinates for each zone marker. Therefore, the order of zone markers is the same for each agent. To share an entire search log, an agent only needs to send a list of time stamps ordered in this way. The first time stamp will correspond to the first zone marker stored in the agent’s memory. To share information about a specific zone marker, the agent must send the specific $x$, $y$ coordinates along with the time stamp, to allow the receiving agent to identify which zone marker to update.

Table 7.4 describes the agent communication methods that are introduced for the coordinated search behaviour. Agents can perform one or more of these communication actions at the same time as a single search action method (aStarSearch, searchZone or followInstructions).

The opportunity for seeking agents to coordinate their search occurs when the seeking agents determine which zone to search next. By sharing information about when different zones have been visited, the seeking agents can prioritise zones that have not been recently searched by either seeking agent.

The seeking agent must request information from the other seeking agent when it needs to select a new zone to search. This occurs when a seeking agent has finished following the actions that were specified by the searchZone method. The directive for the seeking agent to request the search log from the other seeking agent can be written as
<table>
<thead>
<tr>
<th>Communication</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestLog</td>
<td>Sends a message to an agent via its ID number that is understood as a request for the agent’s search log. Contains the sending agent’s ID number so that it can be sent the reply.</td>
</tr>
<tr>
<td>sendLog</td>
<td>Sends a list of time stamps to an agent via an ID number specified in a requestLog message. The list of time stamps is ordered according to the order of zone markers in the agent’s internal memory.</td>
</tr>
<tr>
<td>sendUpdate</td>
<td>Sends a message to an agent via its ID number that is understood as a request for the agent to update its search log with new information. Contains the x, y coordinates of a zone marker and a time stamp to be entered into the search log.</td>
</tr>
<tr>
<td>updateLog</td>
<td>Directs the agent to update the time stamp of a specified zone marker in response to a sendUpdate message. Not a communication method but included for completeness.</td>
</tr>
</tbody>
</table>

Table 7.4: The communication methods for the coordinated search behaviour.

\[ \text{requestlog } \leftarrow \text{searchedzone} \]

The token searchedzone represents that the seeking agent has finished following the actions specified by the searchZone method. The token requestlog represents the requestLog communication method.

When an agent receives a request for a copy of its search log, the agent sends the set of time stamps that it has recorded for the different zones to the requesting agent. The directive for the seeking agent to send another agent a copy of its search log can be written as

\[ \text{sendlog } \leftarrow \text{receivedrequest} \]

The token receivedrequest represents that the agent has received a request for its search log from the other seeking agent. The token sendlog represents the sendLog communication method.

Having requested and received a copy of the other agent’s search log, the first seeking agent must update its own search log. The agent selects the highest time stamp for each zone between its own recorded time stamp and the time stamp received from the other seeking agent. This time stamp represents the most recent time that the zone was searched by either agent. Using this updated search log, the first seeking agent selects the zone with the smallest time stamp to be the zone that it will search next.

\[ \text{selectzone } \leftarrow \text{receivedlog} \]
Having selected a new zone to search, the agent sends a message to inform the other seeking agent. This message specifies the selected zone marker and the current time stamp. The directive for the seeking agent to send this update to the other seeking agent can be written as

\[ \text{sendupdate} \leftarrow \text{newzone} \]

The other seeking agent then associates the received time stamp with the specified zone marker in its internal memory. The directive for the seeking agent to update its internal memory can be written as

\[ \text{updatelog} \leftarrow \text{receivedupdate} \]

In this way, the seeking agents are able to share information about their current and future actions. Having selected a zone to search, the other seeking agent is notified so that it does not also decide to search this zone. Therefore, the first seeking agent updates the other agent by sending the current time stamp, even though the agent has not yet searched this zone.

This additional update is necessary because seeking agents often need to select new zones to search within a few time steps of each other. As the seeking agents share their search logs, if the first seeking agent has not yet reached its selected zone, this zone marker will still be the zone with the smallest time stamp. Therefore, the other seeking agent will also select to search this zone. By sending the current time stamp to the other seeking agent, the first seeking agent is notifying its intention to search this zone next. When the other seeking agent selects a zone to search, this zone will not have the smallest time stamp and the agent will select a different zone.

**Coordinated search behaviour**

The coordinated search behaviour extends the independent search behaviour with additional communication actions for the agents to perform. In addition to the search actions, the coordinated search behaviour also directs the seeking agents to request information from the other seeking agent, to share information when requested and to share information about the agent’s future search. Listing 7.6 shows the necessary modifications to the independent search behaviour (highlighted in bold).

An agent may send or receive a message to update a time stamp, or receive a request for its search log, while at the same time executing any of the search actions in the behaviour. This means that the coordinated search behaviour leads to state-action table entries where the seeking agent is directed to perform more than one action simultaneously. These actions will be one of the original search actions (\text{astarsearch, searchzone, followinstructions}), as well as one or more of the new communication actions.
astarsearch :- newzone.
searchzone :- atzone.
followinstructions :- storedinstructions.
selectzone :- receivedlog.

requestlog :- searchedzone.
sendlog :- receivedrequest.

sendupdate :- newzone.
updatelog :- receivedupdate.

s: 1 {newzone, atzone, storedinstructions, searchedzone receivedlog} 1.
s: 0 {receivedrequest receivedupdate} 1.
in: newzone atzone storedinstructions searchedzone receivedrequest receivedupdate receivedlog.
out: astarsearch searchzone followinstructions selectzone requestlog sendlog sendupdate updatelog.

Listing 7.6: The coordinated search behaviour input file.

7.3.3 Observations and results

The Sinatra GUI makes it possible to observe the agents in the hide-and-seek domain. Using both the independent search behaviour and the coordinated search behaviour, the seeking agents are able to find the hiding agent. However, the search strategies of the agents using the independent search behaviour and the agents using the coordinated search behaviour were observably different.

Observation of the seeking agents using the independent search behaviour showed that they would sometimes search the same zone of the grid at the same time and would often repeat zones recently searched by the other seeking agent. In contrast, the agents using the coordinated search behaviour were always observed to search different areas of the grid. The agent communication used by the coordinated search behaviour was also observed to allow the seeking agents to search the different zones of the grid environment with greater regularity.

However, the time taken for the seeking agents to find the hiding agents was found to vary significantly during the observed simulations, both when the independent search behaviour and the coordinated search behaviour were used. Sometimes the initial spawn positions of the agents meant that the hiding agent spawned next to or very close to a seeking agent. Other times the seeking agents were unable to find the hiding agent for more than 60 time steps (approximately 1 minute).

It was also observed that all of the agents in the hide-and-seek domain had the
potential to participate in undesirable agent interactions, specifically stationary interac-
tions, where agents mutually prevent each other from moving. Stationary interac-
tions can significantly affect the time taken for the seeking agents to find the hiding agent. Therefore, it was necessary to introduce a behaviour for resolving stationary interactions (Section 4.3) to the behaviours that can be followed by the agents in the hide-and-seek domain.

For interest, repeated simulation of the hide-and-seek domain was used to deter-
mine whether the \textit{coordinated search} behaviour offers any advantages to the seeking agents. The hide-and-seek domain was populated by two seeking agents and one hiding agent. The agents start the simulation from a randomly selected spawn position. Each simulation was timed until one of the seeking agents finds the hiding agent, at which point the simulation was stopped and the current simulation time recorded.

Two sets of repeated simulations of the hide-and-seek domain were carried out. In the first set of simulations (S1), the seeking agents use the \textit{independent search} behaviour to search the grid environment. In the second set of simulations (S2), the seeking agents work together to search the grid environment using the \textit{coordinated search} behaviour. There were 200 simulations carried out for each set, the results of which are summarised in Table 7.5.

<table>
<thead>
<tr>
<th>Minimum time</th>
<th>Number of tests</th>
<th>Average time steps taken to find hiding agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All tests</td>
<td>200</td>
<td>20.2</td>
</tr>
<tr>
<td>$t &gt; 1$</td>
<td>185</td>
<td>22.3</td>
</tr>
<tr>
<td>$t &gt; 5$</td>
<td>152</td>
<td>26.5</td>
</tr>
<tr>
<td>$t &gt; 10$</td>
<td>118</td>
<td>31.9</td>
</tr>
<tr>
<td>$t &gt; 30$</td>
<td>41</td>
<td>56.7</td>
</tr>
<tr>
<td>$t &gt; 60$</td>
<td>10</td>
<td>101.3</td>
</tr>
<tr>
<td>S2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All tests</td>
<td>200</td>
<td>18.5</td>
</tr>
<tr>
<td>$t &gt; 1$</td>
<td>180</td>
<td>20.4</td>
</tr>
<tr>
<td>$t &gt; 5$</td>
<td>142</td>
<td>25.0</td>
</tr>
<tr>
<td>$t &gt; 10$</td>
<td>114</td>
<td>29.3</td>
</tr>
<tr>
<td>$t &gt; 30$</td>
<td>36</td>
<td>51.2</td>
</tr>
<tr>
<td>$t &gt; 60$</td>
<td>9</td>
<td>78.4</td>
</tr>
</tbody>
</table>

Table 7.5: Overall results of the hide-and-seek simulations, showing the average number of time steps taken for the seeking agents to find the hiding agent. Results are shown for all tests, as well as for different minimum values of $t$, where $t$ is the time taken for the seeking agents to find the hiding agent.

As already noted, the time taken for the seeking agents to find the hiding agent has the potential to vary significantly. To try to demonstrate these differences in the time taken, the results in Table 7.5 separate the simulations into those that lasted
longer than a minimum time $t$, for different values of $t$. In all cases, the results show the number of simulations that lasted longer than $t$ and the average number of time steps taken by these simulations.

The results show that when the seeking agents use the coordinated search behaviour and so share information about their actions, the agents are able to find the hiding agent slightly faster than when the independent search behaviour is used. While the simulations in set $S2$ consistently had a shorter duration than the simulations in set $S1$, the reduction in the time taken for the seeking agents to find the hiding agent is very small. Therefore, it appears that while the coordination of the agent actions using the coordinated search behaviour leads to an observable difference and a presumed improvement in the search strategy of the seeking agents, the actual benefit of this coordination in this instance of the hide-and-seek domain, if any, is only minor. For a larger hide-and-seek domain, the improved search time for agents using the coordinated search behaviour appeared to be much more significant, although timed experiments in this larger domain have not been carried out.

7.3.4 Summary

Using the example of the hide-and-seek domain, we have demonstrated how directives can be used to coordinate agent actions by allowing agents to communicate in order to share information. The coordinated search behaviour allows the seeking agents to search the grid environment in a more efficient manner by directing the agents to share information about their independent searches.

The coordinated search behaviour attempted to allow the seeking agents to modify their individual searches based on their joint experiences of searching for the hiding agent. The seeking agents are directed to send messages to each other in order to share information about the most recent time that zones have been searched by either agent and to notify the other seeking agent of which zone they intend to search next. This allows the agents to take into account the search carried out by the other seeking agent and to modify their own actions based on this information.

Observations of the seeking agents confirmed that the agents behaved differently when using the independent search behaviour compared to the coordinated search behaviour. Agents using the independent search behaviour were often observed to search zones recently searched by the other seeking agent. Agents using the coordinated search behaviour were always observed to search different zones of the grid and were able to search all of the zones more regularly. Therefore, the coordinated search behaviour allowed the seeking agents to coordinate their actions successfully.

We have described how stationary interactions were observed to occur between the agents and the need to include an additional behaviour to resolve stationary interactions between the agents in the hide-and-seek domain. Stationary interactions are more likely to occur between seeking agents who are using the independent search behaviour.
behaviour. This is because these agents are more likely to search the same zone or to search a zone that the other seeking agent has only recently searched. Therefore, the seeking agents are more likely to be in close proximity and so obstruct each other.

The coordinated search behaviour, however, avoids situations where the seeking agents are searching the same zone and so reduces the occurrence of stationary interactions. As the stationary interactions take a few time steps to resolve, which would otherwise be spent searching, the coordinated search behaviour can be said to provide some improvement to the efficiency of the seeking agents’ search.

7.4 Conclusion

We have demonstrated the use of directives to coordinate the actions of autonomous agents. The directives attempted to manage the unpredictable nature of the other agents by imposing limitations on the actions of the agents so that their actions become more standardised. By using these directives, an agent is able to determine its own actions, whilst taking into account some (possibly assumed) knowledge of how the other agents in the system are going to act. Therefore, the agent is able to select actions to perform that will coordinate with the actions of the other agents in the system.

We identified two possible techniques for implementing this strategy for coordinating agents. Using the example of a robot rugby domain, we have shown the development of the formation behaviour to coordinate agents by establishing expected patterns of behaviours amongst the agents. Using the example of a hide-and-seek domain, we have shown the development of the coordinated search behaviour to coordinate agents by utilising agent communication to share individual information.

7.4.1 Coordination through expected patterns of behaviour

Expected patterns of behaviour allow agents to attempt to coordinate their actions by assuming that the other agents in the system will act in a pre-established manner. By assuming that the actions of the other agents are constrained in a particular way, directives allow an agent to determine its actions so that it can coordinate with this assumed knowledge.

We designed the formation behaviour to allow a team of agents to arrange themselves into different formations by each agent assuming that the actions of its teammates would complement their own. The wedge formation behaviour directed an agent to attempt to find and to maintain its position in a wedge shape behind the ball-carrier. However, a wedge formation was only successfully if the other agent on the team was also able to move into position behind the ball-carrier. Similarly, the formLine action method allowed an agent to move into the same column as its
nearest teammate but a line formation was only successful if the other agents on the
team also moved into this column.

The formation behaviour was observed to allow a team of agents to arrange them-

selves into and to maintain different formations. However, the agents were not
always able to achieve a desired team formation. The individual agents were always
seen to attempt to act as directed by the formation behaviour but were often unable
to complete a formation.

Assuming that an agent is implemented correctly, an agent may fail to comply
with a behaviour because it is prevented from doing so by the actions of another
agent or some factor in the environment, or because it intentionally disobeyed the
directives of the behaviour. If an agent fails to comply with the formation behaviour,
then collectively the team of agents will be unable to achieve the current formation.
This is due to the nature of the wedge and line formations, where every agent (except
the ball-carrier) must move to the correct position for the formation to be achieved.

The agents are unable to achieve a formation when another agent fails to act as
expected because the agents possess only a limited view of the environment. Without
a global view, the agents are often unable to perceive the actual positions of their
teammates and any obstructions that might prevent them from successfully reaching
their place in the formation. Therefore, the agents act based on the knowledge (or
assumption) that the other agents intend to act in an expected manner and are
unable to resolve a situation when this is not the case.

Despite the inability of the team of agents to achieve a formation in all situations,
the formation behaviour was shown to provide a significant benefit to a team of
robot rugby agents. Therefore, based on the example of the formation behaviour,
using directives to coordinate agent actions based on expected patterns of behaviour
appear to be an effective potential method for implementing the local coordination
of agents. The expected actions of the other agents are sufficient to allow the agents
to coordinate their actions and to improve the ability of the agents to achieve a
global objective, despite the coordination itself not always being successful.

7.4.2 Coordination through agent communication

Agent communication allows agents to share information about their past, current
and future actions. By sharing information, the agents are able to develop a greater
understanding of the actions of the other agents. Using this information, directives
allow an agent to determine its actions so that it can coordinate with the other
agents.

We designed the coordinated search behaviour to allow seeking agents to search the
grid environment for a hiding agent, whilst also specifying when the agents should
communicate and the information that should be exchanged. The coordinated search
behaviour directed seeking agents to share only very general information about their
actions by sending messages to each other.

The message passing interface used by the hide-and-seek agents was based on the existing communication model that has been demonstrated for the centralised control mechanism (Section 5.1). For the centralised control mechanism, the messages received from the agents allowed the controller to develop a global view of the system, or some relevant fragment, in order to manage an undesirable agent interaction. For allowing agents to coordinate their actions, however, the messages exchanged between agents allow them to build a shared view of part of the system, but one that is still local to each agent.

The sharing of information allows the coordinated search behaviour to direct the seeking agents to coordinate their actions so that the agents are able to search the grid environment together. The coordinated search behaviour was observed to allow the seeking agents to ensure that they always search different zones of the grid and to search all of the zones more regularly.

If, for whatever reason, the agents fail to comply with the coordinated search behaviour, then the information shared by the agents may be incorrect or incomplete. Therefore, the agents will be less able to coordinate their actions successfully until their shared view of the system has been updated with accurate information from each coordinating agent.

The coordinated search behaviour was not able to provide a clear benefit in terms of the time taken for the seeking agents to find the hiding agent. However, the observable behaviour of the seeking agents was different to those using the independent search behaviour, showing that the coordination was being carried out. Therefore, based on the example of the coordinated search behaviour, using agent communication to coordinate the actions of agents appears to be an effective potential method for implementing the local coordination of agents. The exchange of messages allows the agents to augment their local perception of the system with the perceptions of other agents, creating a shared view of part of the system. The information shared by the agents significantly increased their understanding of the actions of the other agents, allowing the agents to work together towards a global objective.

7.4.3 Local coordination using behaviours

The techniques that have been presented for using directives to coordinate the actions of agents are only suggestions of possible techniques. We have only presented a single example of using expected patterns of behaviour and agent communication to allow the actions of agents to become more predictable. In both examples, limitations were encountered with the effectiveness of the directives in each domain, which corresponds to the conclusion put forward in Section 4.4.4 that directives are unable to manage all agent interactions. Nevertheless, the overall results of our experiments suggest that some coordination of agent actions has been achieved.
A better understanding of the effectiveness of these techniques can be achieved by experimenting with other multi-agent system scenarios where coordination of agents is desirable or there is a global objective for the agents to achieve. Directives that use these techniques to coordinate the agents can be implemented and the Sinatra GUI used to observe the agents following these directives in order to determine whether the coordination is occurring and in what situations it is successful. In particular, a coordination domain where both expected patterns of behaviour and agent communication are implemented will be necessary to compare their relative effectiveness. The robot rugby domain may be suitable for this task. The robot rugby domain will also allow teams of agents using different coordination techniques to be compared at the same time.
8 Case study: Autonomous robot assistants

We have shown the development of example directives for managing agent interactions and helping agents coordinate their actions. We have also investigated the use of a centralised control mechanism to handle situations that are not correctly managed by a set of directives. We now consider how Sinatra and the different behaviours that have been developed so far, together with our greater understanding of possible strategies for their use, can be applied in a potential case study.

We consider the development of autonomous robot assistants to be used in, say, the Department of Computing, using directives to implement their reasoning processes. We begin by outlining the case study scenario and the capabilities of the robots that will be simulated. We then develop a behaviour that allows these robots to travel safely around the department. Next, a series of additional behaviours are developed that allow the robots to assist visitors and members of staff in different tasks. Finally, we evaluate the use of Sinatra and directives to implement robots in this scenario, before suggesting a general conclusion that can be drawn from this case study.

The simulation of robot assistants in Sinatra has not been built. Instead, this chapter presents an argument for how the simulation of the robot assistants in Sinatra and the development of directives for these robots could be carried out, by providing a detailed account of how this would be made to work. In other words, if this system were to be built and it were to be based on norms, then this chapter presents a recommendation for the implementation, based on the findings of the previous chapters.

8.1 Case study outline

We imagine that the Department of Computing is considering the purchase of a fleet of mobile robots. These robots are intended to act as autonomous robot assistants, with the ability to travel freely about the department, fulfilling tasks assigned to them by visitors and members of staff. There are many robotics challenges that will need to be resolved in order for the robots to be able to act as robot assistants. For example, how the robots are able to open doors, how the robots are able to pick up
and carry items and how the robots are able to interact with humans. We do not consider these issues.

Instead, we look at how the implementation of the reasoning processes of these robots, in the form of directives, can be developed and tested using Sinatra. Sinatra has been designed such that the behaviours instructing simulated agents could be used to instruct actual physical robots. Therefore, behaviours developed in Sinatra can be used to determine if the robots can, in principle, be implemented to act in the desired manner.

In addition to providing a means to develop behaviours, the Sinatra simulation can be used to identify potential problems that the robots may encounter and to resolve these problems in advance of deploying the robots. For example, the agents in Sinatra may have difficulty navigating a particular intersection within the simulated department environment. A potential solution to this problem, such as the implementation of a controller to manage the movement of agents through this intersection, can be developed and tested using Sinatra, before deployment of the physical robots.

Besides the original perception and action capabilities of the Sinatra agents (outlined in Section 3.2), further capabilities of the robots are required for the assistance tasks that they will perform. We now describe the general capabilities of the robots and details of how they will be simulated.

We assume that the robots are able to sense their local environment, allowing them to perceive obstructions in their immediate vicinity, as well as to identify nearby humans and other robots. We assume that the robots are also able to perceive their location and orientation within the department, based on a map of the department stored in their internal memory.

We assume that the robots are able to act by turning on the spot and moving forwards. In addition, to allow the robots to assist members of staff, we assume that the robots are able to pick up and carry light objects.

We assume that the robots are able to communicate using simple text-based messages. These messages can be transmitted using the college Wi-Fi, which can also provide a clock to synchronise the actions of the robots.

Tasks are assigned to the robots using an automated task manager. A centrally stored database of robots is maintained, which specifies whether a robot has been assigned a task to perform, meaning that the robot has one or more goal locations to reach. If a robot does not currently have a goal, the robot will be recorded in the database as unoccupied. When a new task is created, the task manager selects an unoccupied robot from the database and assigns the task to this robot by sending a message to inform the robot of its new goal or goals.

To simulate the robots moving around the department a representation of the department as a grid of locations is implemented in the Sinatra GUI. Each area of
the department, such as a section of corridor, an atrium, an office or a lecture theatre, is created as a separate set of grid locations, and so depicted in a similar manner to that shown in the existing Sinatra GUI. Robots are able to move between different areas by travelling to specific grid locations that represent intersections of corridors and doorways of rooms in the department. These grid locations are referred to as goal markers, similar to the zone markers described for use by the seeking agents in the hide-and-seek domain (Section 7.3).

By travelling to a goal marker a robot is able to move to a different area of the department. As each area will be a discrete set of grid locations, many of the behaviours that have already been developed, such as the highway behaviour and the obstacle avoidance behaviour, can be used to direct robots how to act within each area.

A similar representation will form the map of the department stored in each robot’s internal memory. However, instead of including the grid of locations for each area, the map stored by the robots will represent the network of goal markers. Therefore, rather than being able to perceive its exact grid location within the department, a robot will be able to perceive the goal markers that it is between. As the goal markers are placed in the robot’s map to correspond with the intersections of corridors and the doorways of rooms in the department, the robot will be able to travel around the department by moving between the goal markers.

To simulate humans moving around the department and other uncertain dynamic features of the environment, grid locations within the environment will be randomly selected to become ‘obstructed’ for a few time steps. This obstruction may represent a human or some other unexpected obstruction, such as a chair or piece of litter. The robots must be implemented to respond to different obstructions appropriately. We do not attempt to simulate the realistic movement of humans around the department.

8.2 Moving around the department

Before the robots can be used as robot assistants, the robots must be able to move freely and safely around the department. We develop a new behaviour that allows the robots to reach their goal locations, while moving safely around the department and avoiding stationary interactions. We then develop a behaviour-switch that allows robots to use this behaviour to travel to arbitrary locations within the department using an A* search to plan their route.

8.2.1 Safe robot movement behaviour

The robots must be able to navigate safely through the various corridors and atriums within the department. While travelling around the department, the robots will encounter people, as well as other robots. It is vitally important that the robots
are able to avoid all collisions, while still being able to reach their goal locations. Therefore, we develop the safe robot movement behaviour.

If the robots are implemented to avoid collisions at all costs, then the safest action for a robot to perform when confronted with an obstacle in its path is to remain stationary. When the obstruction is a person, the person can easily step around the stationary robot and continue towards his destination.

\[ \text{wait} \leftarrow \text{human} \]

When the obstacle is an inanimate object, such as an item of furniture or a piece of litter, the robot must find a path around this obstruction. We have designed the obstacle avoidance behaviour (Section 4.3.1), which will allow the robot to navigate around an unspecified obstruction.

\[ \text{obstacle} \leftarrow \text{obstructed} \]

If the obstruction is another robot, however, then the interaction becomes more complicated. We have described how the obstacle avoidance behaviour has the potential to cause repeated state interactions to occur (Section 4.3.2). Therefore, we do not wish to use the obstacle avoidance behaviour for this situation. However, if the robots perform the wait action, the robots will continue to mutually obstruct each other. This is a stationary interaction.

We have considered the use of the highway behaviour and the strict traffic lanes behaviour in order to prevent stationary interactions occurring (Section 4.2). While the strict traffic lanes behaviour was able to prevent all stationary interactions, the traffic lanes themselves are not visible to people in the department. This means that the movement of the robots will be unintuitive to a human observer and may cause more collisions to occur. Therefore, the highway behaviour will be used by the robots to move between their goal markers.

\[ \text{highway} \leftarrow \]

For a particular corridor or atrium that the robot needs to travel through, the robot will move along the appropriate highway for its direction of travel. In a corridor, the highways will result in a two lane traffic system, similar to a road. In an atrium, or other large area, the highways will result in organised lanes for use by the robots in crossing the space. This allows people in the atrium to coexist with the robots, as the movement of the robots will normally only be along these established highways, which the people in the atrium can avoid.

We observed in Section 4.2.2 that the highway behaviour was able to organise the movement of agents as they travel towards their goals, but that stationary interactions were still a frequent occurrence. Therefore, a behaviour that can resolve stationary interactions must also be used.
We have considered the use of the obstacle avoidance behaviour and the traffic law behaviour in order to resolve stationary interactions (Section 4.3). Both of these behaviours were able to resolve stationary interactions, however, as mentioned before, the obstacle avoidance behaviour also has the potential to cause repeated state interactions. Therefore, the robots will use the traffic law behaviour to resolve stationary interactions. Robots that must give way will switch to using the traffic law behaviour, whilst robots that have priority will wait for their path to be clear.

\[
\text{trafficlaw} \leftarrow \text{robot, giveaway}\\
\text{wait} \leftarrow \text{robot}
\]

The traffic law behaviour, however, is unable to resolve all stationary interactions. Specifically, the traffic law behaviour is unable to resolve stationary interactions where the agent that must give way is unable to move.

At this point, we have reached the limit of what can reasonably be achieved using these existing behaviours to manage the interactions of the robots (Section 4.4.4). While further modifications and additional behaviours can be introduced, in most cases these changes will unnecessarily complicate the implementation, without providing a significant benefit. Therefore, to resolve these remaining stationary interactions, a centralised control mechanism with a global perspective will be used.

The stationary controller will be used to resolve stationary interactions that cannot be resolved by the existing behaviours (Section 5.3). Robots that are obstructed by another robot and have been unable to move for three time steps will notify the stationary controller that they believe they are participating in a stationary interaction and that they are unable to resolve the interaction themselves. The stationary controller will determine whether a stationary interaction is occurring based on the robots that invoke the controller. If the stationary controller identifies a stationary interaction, the controller will send instructions to the robots to direct them to move away from and to resolve the stationary interaction.

\[
\text{statcontroller} \leftarrow \text{statinstructions}\\
\text{notifystat} \leftarrow \text{robot, waited}
\]

The stationary controller is able to resolve stationary interactions that occur between the robots that invoke the controller. However, in a busy area of the department, the stationary controller may struggle to resolve a stationary interaction because of interference from people and other robots (Section 5.4.3). One such area is the main entrance to the Department of Computing. In this area, many people and robots will interact in a confined space. Therefore, a specific entrance controller will be used to manage the interactions of robots in this area.

Instead of waiting to be invoked by a robot, the entrance controller monitors all of the people and robots in the entrance area. The entrance controller can send
instructions to every robot as it moves within the entrance area, allowing the controller to manage the movement of the robots and prevent undesirable interactions. If the controller detects that a robot is on a collision course with a person or another robot, the entrance controller may instruct the robot so that it follows a different path around this obstruction. In addition, if the entrance controller identifies multiple robots that are travelling to the same location, the controller may direct these robots to travel along the same path and adjust the route of this path in response to the shifting groups of people in the entrance area.

\[
\text{entrancecontroller} \leftarrow \text{entranceinstructions}
\]

Together these directives allow the robots to navigate safely around the department. They are combined to form the safe robot movement behaviour, which is shown in Listing 8.1. The tokens used in the safe robot movement behaviour are defined in Table 8.1.

<table>
<thead>
<tr>
<th>Token</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>human, obstructed and robot</td>
<td>The robot has perceived that it is being obstructed by a person, an inanimate object and a robot respectively.</td>
</tr>
<tr>
<td>giveaway</td>
<td>The robot perceives that it must give way to another robot.</td>
</tr>
<tr>
<td>waited</td>
<td>The robot has been unable to move for three time steps.</td>
</tr>
<tr>
<td>entrance-instructions and statinstructions</td>
<td>The robot has tokens recorded in its memory, which have been received from the entrance controller and the stationary controller respectively.</td>
</tr>
<tr>
<td>wait</td>
<td>The wait action, where the robot performs no action for this time step.</td>
</tr>
<tr>
<td>highway</td>
<td>The highway behaviour.</td>
</tr>
<tr>
<td>obstacle</td>
<td>The obstacle avoidance behaviour.</td>
</tr>
<tr>
<td>trafficlaw</td>
<td>The traffic law behaviour.</td>
</tr>
<tr>
<td>notifystat</td>
<td>The action of the robot sending a message to invoke the stationary controller.</td>
</tr>
<tr>
<td>entrancecontroller and statcontroller</td>
<td>During this time step the robot should follow the next action or actions specified by the entrance controller and the stationary controller respectively.</td>
</tr>
</tbody>
</table>

Table 8.1: The dictionary of tokens for the safe robot movement behaviour.

The directive specifying that a robot should wait when obstructed by a human is given highest priority in order to ensure the safety of people in the department. After this, the two controllers are given next highest priority as their global view of the environment allows them to resolve interactions that are difficult for the robots. As there are two controllers, some of the issues discussed in Section 5.4.3 must be handled for this case study. In particular, the question of when the robots will be directed to follow the instructions of either controller must be considered.
wait :- human.
entrancecontroller :- entranceinstructions.
statcontroller :- statinstructions.
otifystat :- robot, waited.
trafficlaw :- robot, giveaway.
wait :- robot.
obstacle :- obstructed.
highway :- .

:- wait, entrancecontroller.
:- wait, statcontroller.
:- wait, notifystat.
:- wait, trafficlaw.
:- wait, obstacle.
:- wait, highway.
:- entrancecontroller, statcontroller.
:- entrancecontroller, notifystat.
:- entrancecontroller, trafficlaw.
:- entrancecontroller, obstacle.
:- entrancecontroller, highway.
:- statcontroller, trafficlaw.
:- statcontroller, obstacle.
:- statcontroller, highway.
:- trafficlaw, obstacle.
:- trafficlaw, highway.
:- obstacle, highway.

s: 0 {human, obstructed, robot} 3.
s: 0 {entranceinstructions, statinstructions} 2.
s: 0 {waited, giveaway} 2.

in: human obstructed robot waited giveaway entranceinstructions statinstructions.

out: wait highway obstacle trafficlaw notifystat entrancecontroller statcontroller.

Listing 8.1: The safe robot movement behaviour input file.

The entrance controller should be given priority over the stationary controller, as
the entrance controller is able to resolve stationary interactions that occur in the
entrance area itself. If the robot receives instructions from the entrance controller
while instructions from the stationary controller are still stored in its memory, the
robot will delete the instructions from the stationary controller.

While the entrance controller and the stationary controller cannot be used to-
gether, a robot can be in a situation where it is directed to follow the instructions
of the stationary controller and at the same time to notify the stationary controller
that it is being obstructed by a robot. This can occur when a third robot joins a stationary interaction that the stationary controller has previously identified as involving only two robots, where the third robot prevents the robots from following the instructions of the controller. By notifying the stationary controller again, the robots allow the stationary controller to identify that the stationary interaction now involves three robots and to issue instructions accordingly.

After the directives for responding to the controllers, the directives for when a robot is obstructed by another robot allow the robots to attempt to resolve stationary interactions themselves. If the robot must give way, then the traffic law behaviour is used; otherwise, the robot waits for its path to be cleared. If the traffic law behaviour is unable to resolve the stationary interaction, the robots invoke the stationary controller.

If the robot is obstructed by an inanimate object, then the robot is directed to follow the obstacle avoidance behaviour in order to move around this obstruction. Finally, by default, the robots follow the highway behaviour in order to travel to a goal location. Using this safe robot movement behaviour, a robot should be able to travel to its goal location without being involved in any collisions or becoming stuck in any stationary interactions.

8.2.2 Robot movement behaviour-switch

The robots may be required to go to a number of different goal locations in a specific order. Therefore, each robot stores a list of goal locations in its internal memory, where each goal location is a goal marker. This is the goal destination list of the robot. However, the robot needs to determine how to reach each of these goal destinations.

The robots use a map of the department to navigate to different locations, using goal markers to separate the areas for the robots to travel between. By travelling to a list of goal markers in order, a robot is able to navigate to a specific goal location. The robots use an A* search to find a route to their first goal destination from their current location, in terms of adjacent goal markers.

\[
\text{astarsearch} \leftarrow \text{newgoal}
\]

The route found by the A* search is stored as a second list of goal markers in the robot’s internal memory, referred to as the goal marker list. The goal marker list specifies the required goal markers that the robot must travel between to reach its first goal destination. The robots use the safe robot movement behaviour to travel to each of these goal markers in order. In this way the robots are able to navigate to any location within the department.

\[
\text{nextgoalmarker} \leftarrow \text{atgoalmarker, storedgoalmarkers}
\]
\[
\text{saferobotmovement} \leftarrow \text{storedgoalmarkers}
\]
Listing 8.2 shows the behaviour-switch that implements this movement of the robots. The tokens used in the behaviour-switch are defined in Table 8.2.

\begin{verbatim}
nextgoal :- atgoal.
nextgoalmarker :- storedgoalmarkers, atgoalmarker.
saferobotmovement :- storedgoalmarkers.
astarsearch :- newgoal.

wait :- .

:- nextgoal, wait.
:- nextgoalmarker, saferobotmovement.
:- nextgoalmarker, wait.
:- saferobotmovement, wait.
:- astarsearch, wait.

s: 0 \{newgoal, storedgoalmarkers, atgoal\} 1.
s: 0 \{atgoalmarker\} 1.

in: atgoal newgoal storedgoalmarkers atgoalmarker.

out: wait astarsearch nextgoal nextgoalmarker saferobotmovement.
\end{verbatim}

Listing 8.2: The robot movement behaviour-switch input file.

At any point, each robot will either have a new goal destination where it must plan a route to this goal (newgoal), have planned a route to this goal and is following this route of goal markers (storedgoalmarkers), or have reached this goal destination (atgoal). If none of these hold, the robot has no current goal to reach and so will wait until it receives a new goal.

The safe robot movement behaviour directs the robots to travel between adjacent goal markers. By using this behaviour-switch, a list of goal markers found by the A* search allows the robots to use the safe robot movement behaviour to reach a specific location. Repeating this process for multiple destinations allows the robots to navigate between different locations within the department.

8.3 Robot assistant behaviours

Now that the robots are able to navigate safely around the Department of Computing, they can be used to help visitors and staff members in tasks around the department. We describe the implementation of three such behaviours. The guide behaviour directs the robots to help visitors to the department find particular rooms, the deliver behaviour directs the robots to provide general fetching and carrying assistance to members of staff and the search behaviour directs the robots to assist
<table>
<thead>
<tr>
<th>Token</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>atgoal</td>
<td>The result of the atGoal perception method for the first goal marker in the robot’s destination list.</td>
</tr>
<tr>
<td>storedgoals</td>
<td>The destination list of the robot is not empty.</td>
</tr>
<tr>
<td>newgoal</td>
<td>The robot has a new goal destination to reach and has no goal markers in its goal marker list.</td>
</tr>
<tr>
<td>storedgoalmarkers</td>
<td>The goal marker list of the robot is not empty.</td>
</tr>
<tr>
<td>atgoalmarker</td>
<td>The result of the atGoal perception method for the first goal marker in the robot’s goal marker list.</td>
</tr>
<tr>
<td>wait</td>
<td>The wait action, where the robot performs no action for this time step.</td>
</tr>
<tr>
<td>nextgoal</td>
<td>The robot removes the first goal marker from its destination list. If there is at least one goal marker remaining in the destination list, then the newgoal perception will now hold.</td>
</tr>
<tr>
<td>astarsearch</td>
<td>The A* search method.</td>
</tr>
<tr>
<td>nextgoalmarker</td>
<td>The robot removes the first goal marker from its goal marker list. If there is at least one goal marker remaining in the goal marker list, then the robot will travel towards this next goal marker.</td>
</tr>
<tr>
<td>saferobotmovement</td>
<td>The safe robot movement behaviour.</td>
</tr>
</tbody>
</table>

Table 8.2: The dictionary of tokens for the robot movement behaviour-switch.

Security in searching for a lost child.

8.3.1 Visitor guides

Visitors to the department are able to request a robot to guide them to a room or location within the department. This behaviour is inspired by the SIGA robots used to guide visitors around the Santander Banking Group’s visitor centre [YDr11]. A request to guide a visitor to a particular room, along with the current location of the visitor, is sent to a robot that has identified itself as unoccupied. The college Wi-Fi is used to send this message and the recipient is selected from the database of robots, where an unoccupied robot is selected by the task manager.

When a robot receives a request to guide a visitor, the robot stores the goal markers representing the location of the visitor and the location of the destination room in its goal destination list. The robot then uses the safe robot movement behaviour to travel to the visitor’s start location.

\[
\text{tovisitor} \leftarrow
\]

The visitor waits for the robot that will act as his guide to arrive. Having reached the location of the visitor, the robot must wait until the visitor notifies the robot that he is ready to be taken to his destination. Therefore, the visitor must be able
to signal to the robot that he is ready. Something like a button or touch sensor could be provided on the robot for this purpose. Having collected its visitor, the robot travels to the visitor’s selected destination. By following the robot, the visitor is able to reach this destination.

\[
\text{todest} \leftarrow \text{collected}
\]

\[
\text{wait} \leftarrow \text{atvisitor}
\]

If, for whatever reason, the visitor leaves the robot before the robot has reached the visitor’s destination, the robot must stop and wait for the visitor to return. Therefore, the robot requires an additional perception capability to allow the robot to monitor whether the visitor is still following the robot. This perception capability will track the visitor as the robot moves.

When the user initially signals to the robot that he is ready, the robot’s sensors begin to track this person. As the visitor must touch the robot to signal that he is ready, the robot is able to start monitoring the nearest human that it can perceive. If the robot’s sensors are unable to find the visitor at any point, then the robot will stop and wait for the visitor to return and again signal to the robot that he is ready before continuing.

\[
\text{todest} \leftarrow \text{collected}
\]

\[
\text{wait} \leftarrow \text{lostvisitor}
\]

When the robot has reached the visitor’s destination, the robot sends a message to update the central database of robots that the robot is now unoccupied. This central database is automatically updated, allowing the robot to be assigned a new task when one is generated.

\[
\text{notifyunoccupied} \leftarrow \text{atdest}
\]

The \texttt{guide} behaviour, shown in Listing 8.3, implements this behaviour of the robot. The tokens used in the \texttt{guide} behaviour are defined in Table 8.3. The \texttt{guide} behaviour highlights how behaviours developed before deploying the robots can inform the physical capabilities required for these robots. In this case, additional perception capabilities and a method to interact with the user are required for robots to use the \texttt{guide} behaviour.

The default directive of the \texttt{guide} behaviour specifies that the robot must travel to the visitor’s start location. After this, however, the robot must not only travel to a specific location within the department but also ensure that the visitor is following this route. The remaining directives of the behaviour are ordered so that the robot is able to guide the visitor successfully.

The robot will only leave the visitor’s start location once the robot has \texttt{collected} and is able to perceive the visitor. Then, the robot will only travel towards the
notifyunoccupied :- atdestination.
todestination :- collected.
wait :- lostvisitor.
wait :- atvisitor.
tovisitor :- .

:- notifyunoccupied, todestination.
:- notifyunoccupied, wait.
:- notifyunoccupied, tovisitor.
:- todestination, wait.
:- todestination, tovisitor.
:- wait, tovisitor.

s: 0 \{lostvisitor, collected\} 1.
s: 0 \{atvisitor, atdestination\} 1.

in: atvisitor lostvisitor collected atdestination.

out: tovisitor todestination wait notifyunoccupied.

Listing 8.3: The guide behaviour input file.

<table>
<thead>
<tr>
<th>Token</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>atvisitor and atdestination</td>
<td>The result of the atGoal perception method, for the location of the visitor and the visitor’s destination respectively.</td>
</tr>
<tr>
<td>collected</td>
<td>The visitor has touched the button on the robot that informs the robot that he is ready to be taken to his destination. The robot has also been able to begin monitoring this person. This perception is only true if the robot is able to perceive its visitor.</td>
</tr>
<tr>
<td>lostvisitor</td>
<td>The robot is no longer able to perceive the person that it was monitoring.</td>
</tr>
<tr>
<td>tovisitor and todestination</td>
<td>The robot uses the A^ search method to plan a route to the visitor’s location and the visitor’s destination respectively. The safe robot movement behaviour is used by the robot to travel between each goal marker on this route.</td>
</tr>
<tr>
<td>wait</td>
<td>The wait action, where the robot performs no action for this time step.</td>
</tr>
<tr>
<td>notifyunoccupied</td>
<td>The robot must notify the central database that it has completed its guiding duty. The robot can then be assigned a new task when another request is made by a visitor or a member of staff.</td>
</tr>
</tbody>
</table>

Table 8.3: The dictionary of tokens for the guide behaviour.
visitor’s destination while the robot can perceive the visitor. Finally, however, the robot will report that it is unoccupied once it has reached the visitor’s destination, regardless of whether the robot can still perceive its visitor. This is to prevent situations where the robot is still responding to the presence or the absence of its visitor when it has already guided the visitor to his destination.

8.3.2 Helping staff

The robot assistants can be used to help reduce pressures on members of staff, by delivering post, collecting printing and bringing restorative cups of tea. However, while robots may be requested to assist any member of staff, the requests of some members of staff have a higher priority than others.

When the post has arrived at the department, a member of staff, acting as postmaster, sends a request for a robot to collect and deliver the post. A robot is selected by the task manager to respond to this request.

When a robot receives a request to deliver post, the robot first travels to the post room. Here the postmaster loads the robot with letters and sends a message to the robot containing a list of destinations identifying where the robot must deliver these items. The list of destinations are understood as goal markers by the robot and stored in the robot’s goal destination list. Each goal marker corresponds to an office where the robot must deliver the post.

Once the robot has collected the post, the robot proceeds to travel to these goal destinations and deliver the post. The robot uses an A* search to find a route to each destination and uses the safe robot movement behaviour to travel between the goal markers on this route.

\[
\begin{align*}
\text{collectpost} & \leftarrow \text{requestpost, atpost} \\
\text{deliverpost} & \leftarrow \text{havepost} \\
\text{topost} & \leftarrow \text{requestpost}
\end{align*}
\]

Members of staff can request that a robot collects their printing for them. When a robot receives a request to collect printing the robot first travels to the printer room, where the robot can pick up the printing from the printer. The robot then travels to the office of the member of staff that requested for their printing to be collected.

\[
\begin{align*}
\text{collectprinting} & \leftarrow \text{requestprinting, atprinter} \\
\text{deliverprinting} & \leftarrow \text{haveprinting} \\
\text{toprinter} & \leftarrow \text{requestprinting}
\end{align*}
\]

The robot assistants can also be used to bring a cup of tea to members of staff. The implementation of this process follows the same pattern for robots that are collecting and delivering printing. Therefore, the robot travels to the tea room,
collects a cup of tea and delivers the tea to the office of the member of staff who sent the request. However, the robot must also attempt to deliver the tea before it has gone cold.

\[
\begin{align*}
\text{collecttea} & \leftarrow \text{requesttea, attea} \\
\text{deliver tea} & \leftarrow \text{havetea} \\
\text{totea} & \leftarrow \text{requesttea}
\end{align*}
\]

The task manager does not necessarily send a request to assist a member of staff to an unoccupied robot. Instead, a robot that is already following the \text{deliver} behaviour may be selected. This means that a robot may be requested to deliver post, collect printing and to bring tea to different members of staff at the same time. Therefore, the robot must prioritise the different requests.

While the robot can be requested to assist any member of staff, the requests of some members of staff have a higher priority than others. For example, if a member of the academic staff requests a cup of tea, this request takes priority over a request from another member of staff, such as a research associate.

\[
\begin{align*}
\text{deliver tea} & \leftarrow \text{havetea, academic} \\
\text{totea} & \leftarrow \text{requesttea, academic}
\end{align*}
\]

Similarly, any request made by a professor is given highest priority.

\[
\begin{align*}
\text{deliver tea} & \leftarrow \text{havetea, prof} \\
\text{totea} & \leftarrow \text{requesttea, prof}
\end{align*}
\]

\[
\begin{align*}
\text{deliverprinting} & \leftarrow \text{haveprinting, prof} \\
\text{toprinter} & \leftarrow \text{requestprinting, prof}
\end{align*}
\]

To keep track of the destinations of the different requests received by the robot, the robots maintain three separate goal destination lists, which record the destinations for delivering post, printing and tea. The list of post destinations is specified by the postmaster when the robot collects the post. The lists of printing and tea destinations are updated automatically when the robot collects these items, based on the office of the person who sent the request.

Depending on what item the robot is currently delivering, the robot will use the \text{safe robot movement} behaviour to travel along a route to the first goal destination of the appropriate list. When all three destination lists are empty, the robot updates the central database of robots to specify that it is now unoccupied.

\[
\begin{align*}
\text{notifyunoccupied} & \leftarrow
\end{align*}
\]

The \text{deliver} behaviour, shown in Listing 8.3, implements this behaviour of the robot. The tokens used in the \text{deliver} behaviour are defined in Table 8.4. The
collecttea :- requesttea, prof, attea.
collectprinting :- requestprinting, prof, atprinter.

delivertea :- havetea, prof.
totea :- requesttea, prof.

deliverprinting :- haveprinting, prof.
toprinter :- requestprinting, prof.

collectpost :- requestpost, atpost.
deliverpost :- havepost.
topost :- requestpost.

collecttea :- requesttea, attea.
collectprinting :- requestprinting, atprinter.

delivertea :- havetea, academic.
totea :- requesttea, academic.

deliverprinting :- haveprinting.
toprinter :- requestprinting.

delivertea :- havetea.
totea :- requesttea.

notifyunoccupied :- .

:- notifyunoccupied, topost.
:- notifyunoccupied, collectpost.
:- notifyunoccupied, deliverpost.
:- notifyunoccupied, toprinter.
:- notifyunoccupied, collectprinting.
:- notifyunoccupied, deliverprinting.
:- notifyunoccupied, totea.
:- notifyunoccupied, collecttea.
:- notifyunoccupied, delivertea.
:- topost, collectpost.
:- topost, deliverpost.
:- topost, toprinter.
:- topost, collectprinting.
:- topost, deliverprinting.
:- topost, totea.
:- topost, collecttea.
:- topost, delivertea.

Listing 8.4: (part 1) The deliver behaviour input file.
:- collectpost, deliverpost.
:- collectpost, toprinter.
:- collectpost, collectprinting.
:- collectpost, deliverprinting.
:- collectpost, totea.
:- collectpost, collecttea.
:- collectpost, delivertea.
:- deliverpost, toprinter.
:- deliverpost, collectprinting.
:- deliverpost, deliverprinting.
:- deliverpost, totea.
:- deliverpost, collecttea.
:- deliverpost, delivertea.
:- toprinter, collectprinting.
:- toprinter, deliverprinting.
:- toprinter, totea.
:- toprinter, collecttea.
:- toprinter, delivertea.
:- collectprinting, deliverprinting.
:- collectprinting, totea.
:- collectprinting, collecttea.
:- collectprinting, delivertea.
:- deliverprinting, totea.
:- deliverprinting, collecttea.
:- deliverprinting, delivertea.
:- totea, collecttea.
:- totea, delivertea.
:- collecttea, delivertea.

s: 0 {requestpost, havepost} 2.
s: 0 {requestprinting, haveprinting} 2.
s: 0 {requesttea, havetea} 2.
s: 0 {atpost, atprinter, attea} 1.
s: 0 {academic, prof} 1.

in: requestpost requestprinting requesttea atpost atprinter attea havepost haveprinting havetea academic prof.

out: topost toprinter totea collectpost collectprinting collecttea deliverpost deliverprinting delivertea notifyunoccupied.

Listing 8.3: (part 2) The deliver behaviour input file.
The robot receives a request to deliver post, printing and tea respectively. If the request is for printing or tea, the request message will include the office of the person who made the request.

The result of the \texttt{atGoal} perception method, for the location of the post room, the printer room and the tea room respectively.

The robot has collected the post, printing and tea respectively. In the case of collecting post, the robot has also received a message specifying where to deliver the post.

The request was sent by a member of the academic staff and a professor respectively.

The robot uses the A* search method to plan a route to the post room, the printer room and the tea room respectively. The \texttt{safe robot movement} behaviour is used by the robot to travel between each goal marker on this route.

Therobot collects the post, printing and tea respectively. When collecting the post, the robot will wait while the postmaster loads the post onto the robot. The details of how the robot collects printing and tea are not discussed.

The robot uses the A* search method to plan a route to the first recipient of post, printing and tea respectively. The \texttt{safe robot movement} behaviour is used by the robot to travel between each goal marker on this route.

The robot must notify the central database that it has completed its deliveries. The robot can then be assigned a new task when another request is made by a visitor or a member of staff.

<table>
<thead>
<tr>
<th>Token</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestpost,</td>
<td>The robot receives a request to deliver post, printing and tea respectively. If the request is for printing or tea, the request message will include the office of the person who made the request.</td>
</tr>
<tr>
<td>requestprinting and requesttea</td>
<td></td>
</tr>
<tr>
<td>atpost, atprinter and attea</td>
<td>The result of the \texttt{atGoal} perception method, for the location of the post room, the printer room and the tea room respectively.</td>
</tr>
<tr>
<td>havepost, haveprinting and havetea</td>
<td>The robot has collected the post, printing and tea respectively. In the case of collecting post, the robot has also received a message specifying where to deliver the post.</td>
</tr>
<tr>
<td>academic and prof</td>
<td>The request was sent by a member of the academic staff and a professor respectively.</td>
</tr>
<tr>
<td>topost, toprinter and totea</td>
<td>The robot uses the A* search method to plan a route to the post room, the printer room and the tea room respectively. The \texttt{safe robot movement} behaviour is used by the robot to travel between each goal marker on this route.</td>
</tr>
<tr>
<td>collectpost, collectprinting and collecttea</td>
<td>The robot collects the post, printing and tea respectively. When collecting the post, the robot will wait while the postmaster loads the post onto the robot. The details of how the robot collects printing and tea are not discussed.</td>
</tr>
<tr>
<td>deliverpost, deliverprinting and delivertea</td>
<td>The robot uses the A* search method to plan a route to the first recipient of post, printing and tea respectively. The \texttt{safe robot movement} behaviour is used by the robot to travel between each goal marker on this route.</td>
</tr>
<tr>
<td>notifyunoccupied</td>
<td>The robot must notify the central database that it has completed its deliveries. The robot can then be assigned a new task when another request is made by a visitor or a member of staff.</td>
</tr>
</tbody>
</table>

Table 8.4: The dictionary of tokens for the \texttt{deliver} behaviour.

The \texttt{deliver} behaviour demonstrates how a priority order between directives can be used to implement a detailed relationship between agent states and prescribed actions.

The directives to deliver post are given a higher priority than the general directives to deliver printing and tea. This is because a robot will only be asked to deliver post once a day and so this request should be given priority when it occurs.

The directives to deliver tea and printing to professors are given the highest priority of all directives, meaning that these requests will be handled first. A request to bring tea is given priority over a request to bring printing so that the tea does not go cold. Similarly, the directives to deliver tea to members of the academic staff are given higher priority than requests to deliver printing and tea to other members of staff. Members of the research staff who request a cup of tea may have to wait a long time for this to be delivered.

264
The order of the directives for collecting and delivering each item are such that, if the robot receives multiple requests to deliver the same item, the robot will prefer to deliver the items that it has already collected before it travels to collect more. However, if the robot has received requests to deliver items and finds that it is in the correct location to collect these items, the robot will take the opportunity to pick up these items.

A robot may only perform one of the actions specified in the deliver behaviour per time step. Therefore, constraint formulae are required to specify that none of the actions can be performed concurrently. As the robot may receive a request while it is in the process of collecting and delivering any other item, it is not possible to make any assumptions about the possible states of the robot. Therefore, the full set of constraint formulae for all of the possible actions prescribed by this behaviour must be included in the deliver behaviour.

8.3.3 Lost child

A family visiting the department has reported to Security that their youngest child has wandered off and cannot be found. Security can use the Department of Computing robot assistants to help find the child. The robots are able to work as a team in order to search the department for the missing child.

This task is similar to the search performed by the seeking agents in the hide-and-seek domain (Section 7.3). The robots will coordinate their search by sharing information about the last time they searched a particular area of the department. However, there are some differences between the coordinated search behaviour used by the seeking agents and the search behaviour that will be implemented for the robot assistants.

In the hide-and-seek domain, zone markers were used to identify different areas of the environment that need to be searched. For the Department of Computing, goal markers are used to define the boundaries between areas of the department. Therefore, the robots keep track of the zones that have been searched by relating a time stamp to two goal markers, specifying that the robot has searched the area between these two markers.

To search the department, each robot identifies a zone that needs to be searched. In the hide-and-seek domain, each agent maintained its own search log, being the record of the time stamps when the agent last visited each zone. For the Department of Computing, the number of zones to be searched is much greater and there are more robots taking part in the search. Therefore, a centralised search log is maintained, to simplify the process whereby the robots update each other about their individual searches. Instead, the robots send messages to update the centralised search log to indicate the zones that they have searched.
When determining which zone to search next, a robot selects a zone from the search log based on the time stamp recorded for the last time this zone was searched and the distance of the agent from the closest of the two goal markers for this zone. A distance heuristic, similar to the heuristic used for the A* search, is required for the robot to estimate the distance to the goal markers from its current location. Therefore, when determining a zone to search, each robot must perform two calculations for each search log entry. Depending on the number of goal markers and the complexity of the distance heuristic, this may be a time consuming task. Optimisations can be introduced, if required; for example, where the robots only consider a subset of the zones or do not consider their distance from the zone.

Having selected a zone to search, the robot sends a message to associate the current time step with this zone. Similar to the seeking agents, this is to prevent other robots selecting to search this zone and so duplicating the search of this robot.

Having updated the centralised search log, the robot travels to the zone that it has selected to search. The robot uses an A* search to find a route of goal markers to the closest goal marker of this zone and then uses the safe robot movement behaviour to travel between these goal markers.

Once a robot has reached the zone, it must now search for the missing child. In the hide-and-seek domain, each zone was a regular shape and size, allowing a fixed pre-programmed search method to be used to search the zone. For the Department of Computing, the zones to be searched are not all the same size, as a zone may be a corridor, an atrium, an office or a lecture theatre.

To search a corridor, a robot only needs to travel along the appropriate highway for the length of the corridor. To search an atrium, however, or any other larger area, it is not sufficient for the robot to travel along a highway. Therefore, the robots use a search zone method to search larger areas more thoroughly. There are a number of ways that the search zone method could be implemented. For simplicity, let us supposed it is a simple left to right, up-down sweep of the area. Directives implementing this method are omitted.
While the robots search, they attempt to perceive the lost child. If the robot finds the child, the robot will stop its search and send a message to notify Security of its current location. This also notifies the other robots that they can stop searching. The robot then waits with the child until Security arrives.

\[
\begin{align*}
\text{waitwithchild} & \leftarrow \text{foundchild} \\
\text{notifysecurity} & \leftarrow \text{foundchild}
\end{align*}
\]

The search behaviour, shown in Listing 8.4, implements this behaviour of the robot. The tokens used in the search behaviour are defined in Table 8.5. The search behaviour demonstrates a behaviour that allows the robots to use directives to coordinate their actions in order to achieve a global team objective.

```
waitwithchild :- foundchild.
notifysecurity :- foundchild.

saferobotmovement :- corridor.
searchzone :- atrium.

sendupdate :- searchedzone.
selectnewzone :- searchedzone.
sendupdate :- newzone.
tozone :- notifiedsearch.

:- waitwithchild, saferobotmovement.
:- waitwithchild, searchzone.
:- waitwithchild, sendupdate.
:- waitwithchild, selectnewzone.
:- waitwithchild, tozone.
:- notifysecurity, saferobotmovement.
:- notifysecurity, searchzone.
:- notifysecurity, sendupdate.
:- notifysecurity, selectnewzone.
:- notifysecurity, tozone.
```

\[
\begin{align*}
s & : 1 \{\text{searchedzone, newzone, notifiedsearch, corridor, atrium}\} 1. \\
s & : 0 \{\text{foundchild}\} 1.
\end{align*}
\]

in: searchedzone newzone notifiedsearch corridor atrium foundchild.
out: saferobotmovement searchzone sendupdate selectnewzone tozone waitwithchild notifysecurity.

Listing 8.4: The search behaviour input file.

The different search states of the robot (searchedzone, newzone, notifiedsearch, corridor, atrium) specify that the robot has finished searching a zone, has selected a
The robot is within the zone that it intends to search and has identified this zone as a corridor and an atrium respectively. An atrium may also refer to any other larger area within the department.

The result of the atGoal perception method for the second of the two goal markers that identify a zone. When the robot selected this zone to search next (selectnewzone), this second goal marker was the more distant of the two goal markers. Specifies that the robot has completed the search of this zone.

The robot has selected a new zone to search.

The robot has sent an update to the centralised search log to associate the current time stamp with the zone that the robot intends to search and is travelling towards.

The robot has found the child.

The safe robot movement behaviour.

The search method used by a robot to search a large area.

The robot sends an update to the centralised search log to associate the current time stamp with the robot’s current selected zone. This therefore identifies a zone that the robot has just finished searching, or a zone that the robot intends to search next.

The robot selects a new zone to search based on the time stamp associated with the zone and the distance between the zone and the robot.

The robot uses the A* search method to plan a route to the selected zone. The safe robot movement behaviour is used by the robot to travel between each goal marker on this route.

The robot remains with the child until Security arrive.

The robot sends a message to Security specifying its current location.

---

Table 8.5: The dictionary of tokens for the search behaviour.

---

new zone to search, has notified the centralised search log of its intended search and is in the process of searching this zone respectively. The robot can only be in one of these states at a time and so the relative priority of these directives is unimportant.

At any point during the robot’s search, however, the robot may locate the child. Therefore, the foundchild directives must have the highest priority and constraint formulae must be used to ensure that these directives override the other search directives.

Using the search behaviour, the robots are able to maintain a shared record of when zones have been searched and the zones that the robots intend to search next. Therefore, the robots are able to coordinate their actions in order to search the
department more efficiently and to avoid duplication of effort where possible.

8.3.4 Robot assistant behaviour-switch

The robot movement behaviour-switch for the robot assistants in Listing 8.2 allows the robots to travel to a set of goal markers, using the safe robot movement behaviour to reach each location in turn. This behaviour-switch must now be updated to allow the robots to switch between using the guide behaviour, the deliver behaviour and the search behaviour.

The robots use the guide behaviour and the deliver behaviour when they were previously unoccupied and have received a request to carry out these tasks. Only one robot is assigned a task when it is created, although in the case of the deliver behaviour, a robot may receive multiple deliver requests while following this behaviour.

\[
\text{guide} \leftarrow \text{visitor} \\
\text{deliver} \leftarrow \text{request}
\]

The robots may switch to using the search behaviour when unoccupied or when following the deliver behaviour. As the search behaviour may be a time-critical task, as many robots as possible take part in the search in order to search the department faster. A robot that is currently guiding a visitor will finish following the guide behaviour before switching to follow the search behaviour.

\[
\text{guide} \leftarrow \text{visitor, collected} \\
\text{search} \leftarrow \text{lostchild}
\]

The guide behaviour, the deliver behaviour and the search behaviour all require for the robot to travel to different locations around the department. Therefore, the existing robot movement behaviour-switch directives that direct the robot to travel between a list of goal markers are used in conjunction with the new behaviours.

Unlike the robot movement behaviour-switch, however, the action when the robot selects a new goal location to travel to (nextgoal) is determined by the new behaviours, rather than being updated automatically by the behaviour-switch directive. Instead, the actions in the new behaviours where the robot travels, such as tovisitor, deliverprinting and tozone, will specify the new goal location that the agent must reach (newgoal). This will cause the behaviour-switch directives to direct the robot to carry out an A* search to find the best route to reach this location and then to use the safe robot movement behaviour to travel along this route.

Listing 8.5 shows the robot assistant behaviour-switch that allows the robots to use the guide behaviour, the deliver behaviour and the search behaviour. The additions to the robot movement behaviour-switch are highlighted in bold. The additional tokens used in the robot assistant behaviour-switch are defined in Table 8.6.
guide :- visitor, collected.
search :- lostchild.
guide :- visitor.
deriver :- request.

nextgoal :- atgoal.
nextgoalmarker :- storedgoalmarkers, atgoalmarker.
saferobotmovement :- storedgoalmarkers.
astarsearch :- newgoal.

wait :- .

:- search, guide.
:- search, deliver.
:- search, nextgoal.
:- search, wait.
:- guide, nextgoal.
:- guide, wait.
:- deliver, nextgoal.
:- deliver, wait.
:- nextgoal, wait.
:- nextgoalmarker, saferobotmovement.
:- nextgoalmarker, wait.
:- saferobotmovement, wait.
:- astarsearch, wait.

s:  0 {lostchild} 1.
s:  0 {visitor, request} 1.
s:  0 {collected} 1.
s:  0 {newgoal, storedgoalmarkers, atgoal} 1.
s:  0 {atgoalmarker} 1.

in:  atgoal newgoal storedgoalmarkers atgoalmarker lostchild visitor collected request.

out:  wait astarsearch nextgoal nextgoalmarker saferobotmovement search guide deliver.

Listing 8.5: The robot assistant behaviour-switch input file.
<table>
<thead>
<tr>
<th>Token</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>lostchild</td>
<td>Security have notified the robots that a child has been reported lost somewhere within the department.</td>
</tr>
<tr>
<td>visitor</td>
<td>The robot has received a request to guide a visitor to a particular location within the department.</td>
</tr>
<tr>
<td>collected</td>
<td>The robot is currently able to perceive its visitor and is in the process of guiding the visitor to his destination.</td>
</tr>
<tr>
<td>request</td>
<td>The robot has received a request to collect and deliver the post, printing or a cup of tea.</td>
</tr>
<tr>
<td>search</td>
<td>The search behaviour.</td>
</tr>
<tr>
<td>guide</td>
<td>The guide behaviour.</td>
</tr>
<tr>
<td>deliver</td>
<td>The deliver behaviour.</td>
</tr>
</tbody>
</table>

Table 8.6: The dictionary of tokens for the robot assistant behaviour-switch.

8.4 Discussion

We have shown the development of behaviours that would allow robot assistants to move safely around the Department of Computing and to perform tasks to assist visitors to the department and members of staff. We now consider these behaviours and the use of Sinatra to implement behaviours for this case study.

8.4.1 Behaviours

The behaviours that have been developed for the robot assistants have drawn on the results of our investigation of using directives to manage agent actions. In particular, the safe robot movement behaviour and the search behaviour have together used aspects of each part of this investigation.

The safe robot movement behaviour utilises existing behaviours and controller implementations that have already been developed throughout this thesis. By combining these existing features, the safe robot movement behaviour attempts to take into account and to resolve the issues and limitations that were identified when developing these behaviours and controllers in isolation.

The safe robot movement behaviour combines a behaviour to prevent stationary interactions and a behaviour to resolve stationary interactions. From our investigation of using directives, however, we have seen that directives are not always sufficient for achieving a desired global objective and that in some situations a centralised control mechanism is more appropriate.

The safe robot movement behaviour handles this limitation of directives by also utilising the stationary controller and the entrance controller. These controllers demonstrate two variations in how our centralised control mechanism can be implemented. The stationary controller is invoked by the robots, whereas the entrance
controller monitors the actions of the robots in a specific area and issues instructions to these robots without first being invoked. By combining these different elements, the safe robot movement behaviour is able to resolve the limitations of the individual parts and so implements a robust movement behaviour.

The guide behaviour is a relatively simple behaviour. However, this behaviour demonstrates how Sinatra can be used to highlight the required physical capabilities of the actual robot assistants that will use them.

By contrast, the deliver behaviour is significantly more complicated and is able to direct the agents how to act when performing multiple related tasks. The deliver behaviour demonstrates how directives can be used to define behaviours that take into account a detailed relationship between agent states and the actions that should be performed. The priorities between the directives of the deliver behaviour allow different requests from different members of staff to be handled in different ways, depending on the member of staff making the request and the other requests received by the robot.

The search behaviour implements a behaviour that allows robots to use directives to coordinate their actions, demonstrating the final part of our investigation of managing agent actions using directives. By allowing the robots to share information about their current and intended actions, the robots should be able to search the Department of Computing in a more efficient manner than when robots search independently.

Not all of the behaviours outlined for this case study have been tested in Sinatra. However, many of the details of the safe robot movement behaviour, as well as the stationary controller, have been implemented and tested in Sinatra. The results of our existing experiments were taken into account when developing the safe robot movement behaviour. For example, the conditions under which the existing behaviours may perform incorrectly were considered and potential solutions to these problems included in the safe robot movement behaviour.

The guide behaviour, deliver behaviour and search behaviour all utilise the safe robot movement behaviour to implement higher level behaviours of the robots. If the safe robot movement behaviour is able to function correctly, it is reasonable to believe that these higher level behaviours will also be able to direct a set of robots to act as intended in most situations.

Our investigations, however, have shown that the potential for unexpected emergent behaviours to occur must always be considered when using directives. In particular, emergent behaviours can occur when multiple sets of directives are combined in a single setting. For example, this was shown to result in repeated state interactions during our investigations (Section 4.3.2). Therefore, robots using directives must still be simulated in Sinatra to allow unexpected emergent behaviour of the robots to be identified.
8.4.2 Sinatra simulation

Sinatra is currently used to simulate a small fixed-size grid network of locations. Each area of the department, being a corridor, atrium, office or lecture theatre, can similarly be represented as a fixed network of locations. Therefore, using Sinatra, the actions of simulated robots using the behaviours that have been developed for this case study can be tested and observed in each area of the department. In particular, the location of the different highways within each area can be tested and the frequency and type of agent interactions that occur when robots use these highways can be observed.

Similar to the grid environment that is already simulated in Sinatra, the goal locations for the simulated robots will be found around the periphery of each area. These goal locations are the goal markers that represent intersections and doorways within the department. Unlike the existing Sinatra simulation, however, where goal locations can be found anywhere around the periphery of the environment, the goal locations used for this case study will be found in a few specific grid locations for each area. Therefore, the number of robots trying to reach the same location within the environment will be greater, which can be expected to lead to a higher frequency of undesirable agent interactions at these points.

The robots are able to move between adjacent areas of the department by travelling to the goal markers at the periphery of the areas. By combining two or three adjacent areas of the department in the simulated environment, the actions of simulated robots can be observed as they move within and between these areas. In this way, Sinatra could be used to confirm that the directives are operating as intended, allowing the robots to move around the department in a safe manner and allowing the robots to resolve undesirable agent interactions when they occur. In addition, by combining adjacent areas of the department in this way, the Sinatra simulation could be used to verify that the goal markers are correctly positioned to allow robots to move between these areas.

By using Sinatra to simulate sub-areas of the department, with simulated robots moving within a few adjacent areas at a time, potential problem areas within the department can be identified. For example, we have imagined that the entrance area to the department may be difficult for the robots to navigate safely due to the volume of traffic. More concretely, our simulations using Sinatra have shown that agents can have difficulties resolving stationary interactions that occur at the corners of the environment, for which we suggested the development of a corner controller (Section 5.4.3). Therefore, it is also reasonable to assume that there will be particular intersections within the department that the robots are often unable to navigate safely.

The implementation of centralised control mechanisms to handle undesirable agent
interactions can be carried out and tested in Sinatra. The controllers can be used to resolve a specific type of interaction or to manage agent interactions within a particular area. Having developed a controller to handle agent interactions in one problem area or situation, Sinatra can be used to determine whether the same controller implementation will be effective at handling undesirable agent interactions that occur in another area of the department.

In addition, if there is a particular area of the department that the simulated robots have difficulty navigating using their directives, goal markers can be removed such that the robots will have to find a different route between areas of the department. Removing goal markers can also be used to determine and verify the robustness of the implementation under changing conditions; for example, if a particular corridor becomes temporarily impassible due to maintenance work.

To test these situations, Sinatra may be required to simulate robots moving around an entire floor of the department, or even over multiple floors, so that the route planning abilities of the robots can be verified. It is also necessary to simulate the system as a whole in order to identify potential problem situations or areas that were not apparent when simulating subsections of the department in isolation.

While combining two or three adjacent areas of the department appears to be a straightforward progression from the small agent environments that are currently simulated using Sinatra, the ability to simulate and to visualise robots moving around whole floors is a much greater challenge. In principle, the simulation methods implemented in Sinatra could be extended to this much larger environment. However, the real difficulty is that the visualisation mechanism will need to be improved to allow the user to be able to observe and understand the simulation that is occurring. To allow whole floors to be visualised, the Sinatra GUI will need to use a much larger scale and may not be able to show the precise actions of each robot. Log files of the robots’ actions may be required to provide this information to the user.

In summary, Sinatra can be used to implement and to test the directives of the behaviours that are developed for this case study. Robots using these directives can be simulated in Sinatra and their actions and interactions observed using the Sinatra GUI. Due to the rapid development process offered by Sinatra, modifications to the robots, the directives and the environment could be carried out and their effects tested in an efficient manner. However, the potential for Sinatra, in its current state, to simulate a much larger environment is not clear.

8.5 Case study review

We have shown the development of behaviours that can be used to direct physical autonomous robots how to act in the Department of Computing robot assistants
case study. Starting from a basic set of robot capabilities, we show how a robust movement behaviour could be developed to allow these robots to travel safely around the department, while avoiding undesirable agent interactions.

A collection of more advanced behaviours for performing specific tasks have also been devised. These higher level behaviours utilise the safe robot movement behaviour, while achieving sophisticated control of the robots’ actions. The robots are able to use all of these behaviours to determine their actions by following the directives of the robot assistant behaviour-switch, which directs the robots to switch between the different behaviours, as appropriate.

The development of the behaviours in this case study have shown that directives are a potential means of implementing the reasoning processes of a set of agents within a real world scenario. By combining directives and centralised control, whilst also taking into account previously identified limitations, directives appear to be able to bring about complex control and coordination of autonomous agents.

Behaviours developed in this manner, however, can only be considered as successfully implementing the reasoning processes of agents in a real world scenario when agents have been observed using these behaviours. As the directives rely on a defined set of agent capabilities, the directives can be tested in both a real world setting using physical robots and in a simulated setting using the simulation test bed Sinatra.

We have discussed how Sinatra may be able to simulate robots moving around the Department of Computing for this case study scenario. By using Sinatra to simulate the robots, changes to the implementation of the behaviours, centralised control mechanisms and necessary requirements of the physical robots can be identified in a cost efficient and timely manner. However, the accuracy of the Sinatra simulation is a potential limiting factor on the conclusions that can be drawn about the behaviours that are developed.

For many real world situations, the current implementation of Sinatra may rely on an oversimplification of how robots are able to act in a physical environment. The grid network of locations used in Sinatra is an artificial construct that makes it significantly easier to implement the simulation test bed and to develop directives. However, when attempting to apply these directives in a real world scenario, the absence of these grid locations may undermine the conclusions drawn from the Sinatra simulations.

In scenarios where more rigid robot movement actions can be expected, however, the existing Sinatra implementation will be able to develop directives for these robots. Such scenarios occur where the environment of the robots is regimented in some way and where the robots perform a restricted set of actions. While there may be only a few such regimented real world environments, it appears that these environments correspond to existing applications of robots in industry.
One such regimented scenario is a warehouse, for which we have described examples of mobile robots that are already in use, albeit normally guided by a centralised control system (Section 2.2.2). In the context of warehouse robots, the existing Sinatra test bed is able to provide an accurate environment simulation. A warehouse environment also limits the interactions between humans and robots, meaning that a Sinatra implementation of warehouse robots can focus on managing the interactions between robots only. Therefore, this appears to be the most appropriate real world scenario for using directives developed in Sinatra to implement the reasoning processes of physical autonomous robots.

Similar to the Packet-World test bed (Section 2.2.1), Sinatra can be used to investigate the implementation of autonomous agents and strategies for realising the decentralised control of these agents in an automated warehouse transportation system. Packet-World has been used to explore a variety of coordination strategies for multi-agent systems; however, the use of norms has not received much attention. In contrast, Sinatra is specifically designed for the investigation of norms, implemented using directives, to coordinate the actions of the agents in a variety of multi-agent systems and the analysis of the interactions of agents using these norms. Sinatra is therefore suited to the development of directives for norm-governed agents in a realistic warehouse environment.
9 Conclusion

This thesis has aimed to address the absence of strategies for implementing norms in multi-agent system applications and to explore the extent to which norms can be used to manage agent actions in this setting. An implementation strategy has been developed for allowing autonomous agents to take norms into account when determining their actions. Using this implementation strategy, together with the simulation test bed Sinatra, the ability of directives to manage the actions and interactions of autonomous agents and strategies for when these rules are unsuccessful have been investigated through a series of examples.

We now consider how the work described in this thesis has addressed these original aims. We begin with our strategy for allowing agents to take norms into account when determining their actions and how this strategy is implemented for agents in Sinatra. We then compare the Sinatra test bed to the Packet-World test bed, which was one of the main influences on Sinatra. Next, we consider the results of our investigation into using directives to manage the actions of agents and allowing agents to coexist in a shared environment. Finally, we consider how a system that is based on norms can be implemented using Sinatra and directives.

9.1 Taking a set of norms into account using behaviours

The starting point for the implementation strategy proposed in this thesis is the classification of system norms and agent-specific norms described by Craven and Sergot [CS08, Ser08]. Given an agent-specific norm – defined in terms of an agent’s perception and action capabilities – the aim is to allow the agent to take this norm into account, in some manner, in the agent’s implementation or reasoning process. An additional basis for the implementation strategy is to address whether recent work on reasoning using priorities [Han08, Hor07, Hor12] can be applied to the development of rules that implement an agent-specific norm.

In Chapter 3 we presented a method by which a ‘behaviour’, being a named set of prioritised directives, can be used to allow an agent to comply with a norm. We also presented the simulation test bed Sinatra, which enables the actions of agents implemented using directives to be observed and analysed. The main results of this chapter were to demonstrate the strategy for allowing an agent to take a norm into account when determining its actions and to show that agents following this strategy
can be implemented, simulated and visualised using Sinatra.

We also showed that existing work on reasoning using priorities addresses a range of issues that do not arise for prioritised directives. This allowed the definitions for reasoning using prioritised directives to be simpler than those suggested for reasoning using other types of prioritised rules.

Based on these definitions, we developed an implementation method for allowing agents simulated in Sinatra to determine their actions using behaviours. A behaviour is encoded as a prioritised extended logic program. Using a translation framework, we convert this into an extended logic program without priorities, from which the answer set solver CLASP is used to generate the state-action table for the behaviour. The agents in Sinatra are implemented to follow the directives of a behaviour by consulting the entries in the associated state-action table.

The generation of a state-action table for each behaviour is a significant limitation of the presented work. The state-action table will grow exponentially when new variables are introduced to the agent’s state for the behaviour. This occurs specifically when controllers are being implemented, as the addition of the token to invoke the controller will cause the state-action table to double in size.

We chose to use state-action tables because we wanted to make minimal assumptions about the reasoning capabilities of the agents in Sinatra. However, it should be stressed that the modular nature of behaviours has kept the associated state-action tables relatively small and lookup operations on these tables can be performed efficiently. We have also described how a state-action table can be encoded as a hash map if performance during this lookup step becomes an issue, although this must be balanced against the necessary pre-processing time to construct the hash map.

It may be that an alternative implementation strategy would be more suitable for implementing agents that use directives to determine their actions. For example, agents can use CLASP to determine the actions associated with their current state at runtime. CLASP has been optimised for very efficient computation of answer sets. At its heart, however, CLASP uses techniques for boolean satisfiability checking (SAT), which means that CLASP may also struggle with exponential growth in the search space.

9.2 Comparison of Sinatra and Packet-World

The Packet-World test bed (Section 2.2.1), developed by Weyns et al. [WHH05, Wey09b], is used to investigate the implementation of autonomous agents in multi-agent systems, where the agents cooperate to achieve complex tasks. Specifically, Weyns et al. have used Packet-World to investigate the implementation of decentralised control for an automated warehouse transportation system (e.g. [Wey09a]). This domain and simulation purpose are similar to Sinatra, which we designed and
used to simulate norm-governed autonomous agents in a general abstract domain, based on a warehouse environment.

In fact, we have also used Sinatra to implement our own version of the Packet-World domain. The Sinatra implementation of the Packet-World domain, as described in [WHH05], is shown in Figure 9.1. The agents in our Packet-World implementation use simple behaviours to collect and to deliver packets.

![Figure 9.1: The Sinatra implementation of the Packet-World domain.](image)

The agents collect packets shown as coloured squares and attempt to deliver these to the destination locations, shown by the correspondingly coloured circles. Agents with coloured bodies are carrying a packet, attempting to find the correct destination, while agents with white bodies have just delivered or are moving to pick up a packet.

The properties of the domain and the environment shown in Figure 9.1 are the same or similar to those described for Packet-World [WHH05]. For example, an agent cannot enter a grid location containing a packet, a destination or another agent. Agents are able to pick up packets from and deliver packets to adjacent grid locations. Finally, each agent is only able to perceive a distance of two grid cells in any direction from their current location.

There are many clear similarities in the type of domain simulated by these two test beds and the agents that populate these domains. Nevertheless, to our knowledge, the implementation of norm-governed agents in Packet-World has not received much attention by Weyns et al. Instead, Weyns et al. have focussed on the use of the environment to manage the agents’ actions [WOO07, WH07]. This contrasts to Sinatra, where the main purpose of our test bed is to investigate the implementation of norms to manage agent actions.

We now compare the Sinatra and Packet-World test beds.
9.2.1 Simulation and visualisation

Sinatra and Packet-World are both written using Java; although Sinatra, being more recently developed, is written in Java 6, whereas Packet-World is written in Java 1.3. This difference allows the Sinatra simulation to benefit from the additions that have been made to the Java language during this time. These include

- the introduction of concurrency utilities (java.util.concurrent) – used to implement the synchronised execution of agent threads and to animate the movement of agents in the Sinatra GUI.

- the introduction of JAXB version 2.0 – used to implement the replay facility in the Sinatra GUI.

Both Sinatra and Packet-World are implemented with standard controls for viewing the simulation via the simulation GUI, such as play and pause. However, the Sinatra GUI is also able to visualise the movement of agents as the simulation is progressing and then to replay their actions in the GUI. These additional features proved useful in order to analyse the agent interactions that take place in Sinatra and to identify the behaviours that are used by each agent.

These features reflect the use of Sinatra to investigate the implementation of norms in multi-agent systems. Packet-World, in contrast, has focussed on the use of the environment to manage the actions of agents. Weyns et al. describe how the environment can be seen as an active entity in the multi-agent system and describe how such an environment is used to control agent perception, action and communication [WOO07, WH07]. This control of the environment is reflected in the Packet-World implementation [Wey09b], where the agents appear to reason based on their internal state and then to inform the environment of their chosen action, whereby the environment then handles the execution of the action for each agent.

9.2.2 Environment

Sinatra has been implemented to be a multi-purpose test bed, designed to allow a range of possible domains to be implemented. We have shown a generic grid domain, based on a warehouse environment, as well as a robot rugby domain and a hide-and-seek domain (Chapter 7). We have also considered how Sinatra can be used to simulate a much larger and more complex environment, such as the Department of Computing, demonstrated in the case study application (Chapter 8).

By contrast, the Packet-World test bed appears to have only been used to simulate variations on the Packet-World domain. However, within this domain, Packet-World is comparatively much more configurable than the domains considered in Sinatra. For example, Packet-World allows the user to select the size of the domain, the
number of agents and the number of packets. This allows a greater variety of domain variations to be considered than is available in Sinatra, allowing a more in-depth study of this domain.

Analogous variations in the domains used by Sinatra can be implemented. In general, however, we chose in Sinatra to support a fixed-size environment and have only considered variations in the number of agents.

Packet-World also contains a variation known as Wall-World, where impassible barriers are introduced into the environment. Such obstructions have not been implemented in Sinatra at this time, although they were present in an early prototype. Impassible grid locations were also described as part of the case study in Chapter 8 as a way to represent non-robot obstacles in the simulation, such as humans and pieces of furniture. The obstacle avoidance behaviour was designed to be a generic behaviour that would direct the agent around any obstruction, not just another agent.

9.2.3 Agents

In general, the agents in Sinatra and Packet-World can operate in a very similar manner. They use a general movement strategy and react to events, objects or agents in the portion of the environment that they are able to perceive. The agents in Sinatra are always restricted to a minimal field of view, being just the adjacent locations that the agent can move to; the only exception being for the Sinatra implementation of the Packet-World domain. This severely limits the perception capabilities of the agents, which must then be overcome by the behaviours that are developed. In contrast, the Packet-World agent field of view can range from the distance of two grid cells in each direction, to a maximum range encompassing the whole environment. This modification could be added easily to Sinatra.

Packet-World also considers the added complication of agents that run on battery power. Agents must collect and deliver packets while also ensuring that their batteries are kept charged. This is an important consideration for real world autonomous robots. However, the use of an agent battery has not been investigated using Sinatra.

The most significant difference between Sinatra agents and Packet-World agents are the methods of communication available to the agents. Sinatra agents have very limited communication capabilities, where agents are able to send text-based messages to known recipients. In contrast, Packet-World agents are able to communicate with nearby agents via a communication protocol. This protocol was developed to allow agents to attempt to set up chains of agents to deliver packets more efficiently. While both test beds use communication to allow the agents to coordinate their actions, the Packet-World protocol is much more advanced than the simple information exchange used by Sinatra agents.
Packet-World has also been used to investigate the effect of indirect agent communication abilities, based on placing markers in the environment. Flags are used to indicate that an agent has found no packets in an area, crumbs are used to mark the path an agent has taken, pheromones are used to allow agents to observe the paths taken by other agents and gradient fields are used to allow agents to find charging points. These communication methods set up swarm-style feedback between the agents, allowing the agents to search for or deliver packets more efficiently by indirectly coordinating their actions.

The communication methods, along with the environment in Packet-World, are used to manage the actions of the agents. Together, these features allow the agents to coexist and to function together effectively in order to achieve the desired global objective of collecting and delivering all packets. Sinatra uses norms, implemented using directives, to attempt to achieve the analogous global behaviour in each domain.

9.3 Using behaviours to manage the actions of agents

The starting point for the investigation carried out in this thesis is a commonly repeated suggestion that norms and social laws are a possible mechanism for allowing autonomous agents to coordinate their actions dynamically and so to coexist while achieving their individual goals (see e.g. [ST95]). Despite this proposal, there appears to have been very little consideration of the implementation of norms in physical multi-agent system applications. When we examine existing real world examples of multi-robot systems, it appears that the operation of these robots is normally managed by a centralised control system [KIV, Gra12]. Where a decentralised management system has been used to coordinate the actions of autonomous robots, swarm robotics appears to be the implementation method of choice [Fra12, YDr11].

The aim was to explore to what extent norms, expressed here in the form of directives, can be used to manage the actions of agents, implemented using the implementation strategy and simulation test bed presented in Chapter 3. The Sinatra test bed enabled this investigation by allowing the actions of agents using directives to be analysed and providing an efficient implementation and testing tool for the development of directives as behaviours.

The use of Sinatra to develop and to analyse the directives developed during this investigation, however, also presented a limitation on the behaviours, agent interactions and domains that could be investigated. It was difficult to display more information about the simulation that was occurring in the visualisation of the simulated agents. We were unable to devise a method for displaying information about when agents use certain directives or perceive certain events in a coherent manner. Therefore, we were limited to designing simulation domains and behaviours.
that would allow agents to interact in a manner that was easy to comprehend using the available visualisation features.

This in turn meant that we were only able to consider agent interactions that involved agents meeting in the grid environment and moving in relation to each other. This type of interaction is easy to follow and analyse using the Sinatra GUI and replay method. It would be interesting to be able to investigate agent interactions that focus on agent communication or other less physical interactions, but the visualisation component in Sinatra was not the main focus of the Sinatra implementation and further development would be a separate sub-project.

9.3.1 Using behaviours to manage agent interactions

In Chapter 4 we began our investigation by considering the use of norms to manage the agent interaction that we characterised as a stationary interaction. We were able to demonstrate that directives can be used to manage agent interactions but that using local rules to determine the actions of independently acting agents means that there will be some situations where the local rules are unable to achieve the desired global objective.

By implementing behaviours to prevent and to resolve stationary interactions, we demonstrated two strategies by which norms, implemented using directives, can be used to manage the interactions of agents and that these directives were an effective mechanism for managing stationary interactions in general situations.

For all of the example behaviours, however, we demonstrated that problem situations can arise or that emergent undesirable behaviour could be observed. We described how these problems would persist, in one form or another, even after additional directives and behaviours had been introduced to resolve the initial problems. Therefore, we suggested that problem situations are to be an expected outcome of using directives to manage agent actions and that eventually these situations require a global solution.

9.3.2 Using controllers when behaviours are unsuccessful

In Chapter 5 we investigated how directives can be used in conjunction with a centralised control mechanism, or controller, to manage agent interactions that are not successfully handled by the directives alone. The main result of this chapter was to demonstrate that controllers, using a global view of an interaction, are able to manage agent interactions that proved difficult for directives to resolve.

We described a general centralised control mechanism, where agents or the controller monitor the actions of agents to identify situations where an undesirable interaction can be prevented or needs to be resolved. The controller sends instructions to the agents in order to manage the interaction, based on the resolution
mechanism implemented for the controller. We discussed general issues related to
the use of a controller, including how multiple controllers can be used together and
what happens when an agent fails to act as instructed by a controller.

We demonstrated the centralised control mechanism by implementing controllers
to resolve what we called repeated state interactions and stationary interactions. We
showed how directives can be used to allow an agent to invoke and to respond to the
instructions of a controller. However, in both of these examples, a relatively simple
pre-programmed heuristic method was used to implement the resolution mechanisms
of the controllers.

In Chapter 6 we continued our investigation by exploring the implementation
of a more sophisticated resolution mechanism for a controller, using the universal
multi-agent planning framework UMOP. The main result of this chapter was that,
while UMOP itself does not appear to be a suitable implementation tool, it is pos-
sible in principle to use existing tools such as planners to implement the resolution
mechanism of a controller.

We demonstrated a controller implementation using UMOP and discussed pos-
sible strategies for using the multi-agent and adversarial planning capabilities of
UMOP to develop a better resolution mechanism. However, when generating and
simulating plans using UMOP, we encountered problems due to the time taken for
plan-generation and the size of the output plan. This meant that we were severely
restricted in the planning domain that could be used, in particular, in terms of the
number of agents in the domain and the complexity of the goal condition. Therefore,
we found that UMOP is only suitable as an implementation tool for the resolution
mechanism of a very specific and limited controller.

The example controllers that have been implemented are all designed to resolve
undesirable interactions when invoked by the agents that are participating in the
interaction. This is a limitation of the presented work. While these controllers
demonstrate the general centralised control mechanism and highlight how directives
are used to allow agents to interact with controllers, these examples only represent
one possible type of controller implementation.

We have discussed possible alternative implementation strategies for the cen-
tralised control mechanism but an important avenue of future investigation is to
implement examples of these controllers. In particular, the implementation of the
corner controller would be a valuable starting point.

The corner controller, initially described in Section 5.4.3, is conceived as a con-
troller that monitors the actions of the agents within a specific area of the grid. The
corner controller can be used to resolve stationary interactions that occur in this
area but also has the potential to prevent stationary interactions from occurring by
proactively sending instructions to the agents. In addition, the corner controller does
not have to send all instructions to the agents at once and can monitor the progress
of the agents as they follow these instructions. This allows us to use the corner controller to explore the effect of agents that fail to comply with the instructions of a controller. Therefore, this controller implementation will allow us to demonstrate and to explore many significant aspects of the centralised control mechanism that are not included in the current examples.

A key implementation point from the current example controllers, however, is that each controller only has access to information normally available to the interacting agents, but that the controller has the ability to combine these local perceptions in order to develop a global view of the interaction. Specifically, this allows the controller to identify which agents are interacting with which other agents. Based on this information, the controller is able to identify a set of actions for the agents to perform in order to resolve the undesirable interaction.

This property of the controllers that have been implemented presents an interesting avenue for future investigation. While it is unlikely that combining just the local perceptions of the agents will be sufficient to resolve undesirable interactions in all circumstances, the implementation of the example controllers show that it is possible. Therefore, if agents are able to communicate in order to share their perception information amongst themselves, as they do with the example controllers, the agents can be implemented to be able to resolve undesirable interactions autonomously. Such an implementation would allow directives to resolve the undesirable interactions, using a strategy similar to that suggested in Section 7.3 for allowing agents to coordinate their actions.

9.3.3 Using behaviours to coordinate agent actions

In Chapter 7 we considered the use of directives that not only allow agents to coexist but are also specifically designed to allow agents to coordinate their actions in order to achieve a desired global or team objective. We devised behaviours to allow agents to coordinate their actions in a simulated robot rugby domain and a hide-and-seek domain. The main result of this chapter was to demonstrate that directives can be used to allow agents to coordinate their actions by regulating the actions of agents and so managing the unpredictable nature of the other agents’ actions.

By implementing directives that used expected patterns of behaviour and agent communication to manage the unpredictable nature of agent actions, we demonstrated two techniques by which directives can be used to coordinate the actions of agents. Using Sinatra to observe and to analyse the actions of agents following these directives, we identified that the directives were able to allow the agents to coordinate their actions.

For interest, we ran repeated simulations in Sinatra of agents using these example behaviours. In these instances of the coordination domains, we determined whether the additional coordination between the agents provided any benefit to the agents
in terms of achieving their global objective. We found that the formation behaviour used in the robot rugby domain provided a clear benefit to a team of agents using this behaviour, even though the agents were not always able to achieve the desired formation exactly. For the coordinated search behaviour in the hide-and-seek domain, however, we found that there was no clear benefit to seeking agents using this behaviour in this small instance of the domain, despite an observable difference in their search strategies.

The behaviours developed in this chapter represent only a single example of two possible techniques for implementing agent coordination by managing the unpredictable nature of agent actions. Significant further work must be carried out to explore these techniques and other coordination strategies in more detail. Despite the mixed results for the effect of directives on the ability of agents to achieve a global objective, the work carried out in this chapter is the first step in this investigation. In particular, in future experiments we can use the robot rugby domain to compare agents using different coordination behaviours at the same time.

9.4 Developing behaviours for a norm-governed application

To complete the thesis, we presented a case study application, drawing on aspects of all parts of the investigation described in the preceding chapters. The aim was to show how the individual parts of this work – the method for implementing norms using directives, the Sinatra test bed, and our experience of using directives to manage agent actions – can be used together to achieve a functional set of behaviours for determining the actions of norm-governed agents.

In Chapter 8 we presented the autonomous robot assistants case study, where mobile robots are required to fulfil tasks assigned to them by visitors and members of staff. The main result of this chapter was to demonstrate ways in which Sinatra and directives can be used to implement norm-governed agents in a potential application.

We described behaviours to allow the robots to travel safely around the department, utilising behaviours to manage agent interactions, controllers that are invoked by the agents, controllers that monitor the agents’ actions, and runtime decision making by the agents in the form of an A* search. We then described three robot assistant behaviours, which included the use of agent communication to coordinate the actions of agents. These assistant behaviours demonstrated how behaviours can be used in a hierarchical manner, by using the safe robot movement behaviour within their implementation.

We considered how Sinatra could be used to simulate the physical layout of the Department of Computing and to test the behaviours that are developed. We described how these simulations could be used to learn about how the behaviours will
direct physical robots to operate, including highlighting potential problem areas within the department and verifying that the agents are able to navigate between areas.

A limitation of the presented work concerns the implementation and testing of the behaviours developed for the robot assistants in Sinatra. In this case study we have only described how the implementation could be carried out. Simulating an entire representation of the whole Department of Computing is beyond the capabilities of the current visualisation features in Sinatra. To verify the suitability of Sinatra for developing behaviours, an important avenue of future investigation is to demonstrate the use of Sinatra to test the safe robot movement behaviour.

While the improvement of the Sinatra visualisation capabilities to be able to represent a large physical domain such as the Department of Computing would be a significant project in itself, the existing Sinatra visualisation should be able to verify aspects of the safe robot movement behaviour. In particular, we can create a domain containing interconnecting rooms and corridors in order to test the navigation capabilities of agents using the safe robot movement behaviour. This will allow the navigation of agents via goal markers and the route planning capabilities of agents using these goal markers to be observed and analysed.

We ended the case study by discussing how Sinatra and the behaviours that have been developed could be used to assist in the development of actual assistant robots. Behaviours developed in Sinatra can help to determine the required capabilities of the physical robots and modifications to behaviours can be tested efficiently and safely before being applied to physical robots. However, we identified that the utility of the conclusions drawn from the Sinatra simulations of these behaviours relies on the accuracy of the simulated domain and agent capabilities implemented in Sinatra.

This presents a significant limitation for using Sinatra to simulate a realistic domain, due to the simplifying assumptions introduced to the Sinatra simulation. These include the use of a grid domain and restrictions on possible agent movement. A major but worthwhile modification to the Sinatra test bed would be to improve the accuracy of the basic simulated environment. To do this, the current grid environment must be replaced with a two-dimensional geometric space. This will allow agents in Sinatra to move freely between points in the environment, turning to face any angle and travelling along vectors.

As mentioned in Chapter 8, this modification will require significant changes to the Sinatra test bed. However, the resulting improved simulation tool will allow a more realistic simulation of agents that have comparable abilities to the existing Sinatra agents. The new Sinatra agents will be able to turn on the spot to face any direction and then to move forwards in this direction, while the agent perception methods will allow the agents to observe their environment within a certain radius.
Without these modifications, however, we have identified that Sinatra may simulate a sufficiently accurate environment for developing directives for autonomous warehouse robots. Therefore, a clear direction for future research is to use Sinatra to investigate the implementation of norm-governed agents in a warehouse transportation system. Similar to the work carried out by Weyns et al. to explore the implementation of autonomous warehouse agents using Packet-World, Sinatra will allow us to investigate the use of norms to implement these agents, where the norms are implemented using directives.

Using a simulation domain similar to that shown in Figure 9.1, we can develop directives to allow agents to coexist while collecting and delivering specific colours of packets. We can also develop directives to allow agents to coordinate their actions in order to collect and deliver all packets in an efficient manner. Controllers may be necessary to prevent undesirable agent interactions, specifically stationary interactions, in the vicinity of the destination locations; or we can experiment with the use of agent communication to allow agents to manage the delivery of packets themselves. The different behaviours developed through these experiments should be suitable for testing by physical warehouse robots, leading to an implementation of norm-governed agents in a potential real world domain.
Bibliography


